모범 사례 가이드

Amazon EKS



Copyright © 2026 2024. Review license at https://github.com/aws/aws-eks-best-practices/blob/master/LICENSE

Amazon EKS: 모범 사례 가이드

Copyright © 2026 2024. Review license at https://github.com/aws/aws-eks-best-practices/blob/master/LICENSE

Amazon의 상표 및 트레이드 드레스는 Amazon 외 제품 또는 서비스와 함께, Amazon 브랜드 이미지를 떨어뜨리거나 고객에게 혼동을 일으킬 수 있는 방식으로 사용할 수 없습니다. Amazon이 소유하지 않은 다른 모든 상표는 Amazon과 제휴, 연결 또는 후원할 수도 있고 그렇지 않을 수도 있는 해당 소유자의 자산입니다.

Table of Contents

소개 1
관련 가이드 1
제공 2
보안 3
이 설명서의 사용법 3
공동 책임 모델 이해 3
소개 1
Feedback 5
참고 문헌 6
도구 및 리소스 6
EKS Auto Mode - 보안 6
보안 아키텍처 7
자주 묻는 질문(FAQ) 16
자격 증명 및 액세스 관리 19
EKS 클러스터에 대한 액세스 제어 19
클러스터 액세스 권장 사항 26
EKS 포드의 자격 증명 및 자격 증명 32
EKS 포드에 대한 자격 증명 및 자격 증명 권장 사항 40
도구 및 리소스 48
포드 보안 48
Linux 기능 48
포드 보안 솔루션 50
권장 사항 58
도구 및 리소스 48
멀티테넌시 64
소프트 멀티테넌시 65
Kubernetes 구문 67
제어 완화 69
하드 멀티테넌시 76
향후 지침 76
다중 클러스터 관리 도구 및 리소스 77
탐지 제어 77
권장 사항 58
도구 및 리소스 48

네트워크 보안 85
트래픽 제어 85
네트워크 암호화 85
네트워크 정책 85
권장 사항 58
보안 그룹 90
포드에 네트워크 정책 대 보안 그룹을 사용해야 하는 경우는 언제입니까? 92
Service Mesh 정책 적용 또는 Kubernetes 네트워크 정책 94
ThirdParty 네트워크 정책 엔진 95
전송 중 암호화 96
도구 및 리소스 48
데이터 암호화 및 보안 암호 관리 105
저장 시 암호화 105
보안 암호 관리 107
도구 및 리소스 48
런타임 보안 110
보안 컨텍스트 및 내장 Kubernetes 제어 111
권장 사항 58
도구 및 리소스 114
인프라 보안 114
권장 사항 58
대안 118
도구 및 리소스 48
규정 준수 121
왼쪽으로 이동 123
도구 및 리소스 48
인시던트 대응 및 포렌식 124
샘플 인시던트 대응 계획 125
권장 사항 58
도구 및 리소스 48
이미지 보안 129
권장 사항 58
도구 및 리소스 48
다중 계정 전략 139
다중 테넌트 클러스터에 대한 다중 워크로드 계정 전략 계획 139
중앙 집중식 EKS 클러스터 140

분산형 EKS 클러스터 145
중앙 집중식 및 비중앙 집중식 EKS 클러스터 147
클러스터 액세스 관리 149
EKS 액세스 관리 옵션 149
옵션 1: 클러스터 액세스 관리(CAM) API를 사용하는 AWS IAM Identity Center 150
옵션 2: Kubernetes 그룹에 매핑된 AWS IAM 사용자/역할 152
옵션 3: OIDC 공급자 152
워크로드에 대한 AWS EKS Pod Identity와 IRSA 비교 153
권장 사항 154
클러스터 Autoscaling 155
EKS Auto Mode 155
Auto Mode를 사용하는 이유 156
FAQ 156
Karpenter 162
권장 사항 58
Karpenter 모범 사례 163
NodePools 생성 166
포드 예약 171
CoreDNS 권장 사항 173
Karpenter 청사진 174
추가 리소스 84
Cluster Autoscaler 174
개요 174
성능 및 확장성 최적화 178
비용 및 가용성 최적화 181
고급 사용 사례 183
추가 파라미터 185
추가 리소스 84
참조 76
신뢰성 189
이 설명서의 사용법 3
소개 1
Feedback 5
애플리케이션 192
권장 사항 58
Horizontal Pod Autoscaler(HPA) 196

Vertical Pod Autoscaler(VPA) 197
애플리케이션 업데이트 198
상태 확인 및 자가 복구 199
권장 사항 200
중단 처리 202
권장 사항 203
관찰성 204
권장 사항 205
제어 플레인 207
EKS 아키텍처 207
권장 사항 208
컨트롤 플레인 지표 모니터링 208
클러스터 인증 211
Admission Webhook 213
클러스터 업그레이드 처리 214
클러스터 엔드포인트 연결 214
대규모 클러스터 실행 215
추가 리소스: 216
데이터 영역 216
권장 사항 58
네트워킹 223
Kubernetes 네트워킹 모델 223
컨테이너 네트워킹 인터페이스(CNI) 224
Amazon Virtual Private Cloud(VPC) CNI 225
서브넷 계산기 226
VPC 및 서브넷 고려 사항 226
개요 174
권장 사항 58
Amazon VPC CNI 236
개요 174
권장 사항 58
IP 주소 사용률 최적화 248
노드 수준 IP 소비 최적화 249
IP 소모 완화 249
IPv6 클러스터 실행 255
개요 174

권장 사항 58
사용자 지정 네트워킹 264
구성의 예제 265
권장 사항 58
Linux용 접두사 모드 272
권장 사항 58
Windows용 접두사 모드 277
권장 사항 58
포드당 보안 그룹 281
권장 사항 58
로드 밸런싱 290
Load Balancer 유형 선택 290
로드 밸런서 프로비저닝 291
Load Balancer 대상 유형 선택 292
Load Balancer서 상태 확인 구성 294
가용성 및 포드 수명 주기 295
참조 76
부록 299
네트워크 성능 문제 모니터링 301
CoreDNS 트래픽에서 DNS 제한 문제 모니터링 301
Conntrack 지표를 사용하여 DNS 쿼리 지연 모니터링 301
기타 중요한 네트워크 성능 지표 302
지표를 캡처하여 워크로드의 네트워크 성능 문제 모니터링 302
IPVS 모드에서 kube-proxy 실행 310
개요 174
확장성 315
이 설명서의 사용법 315
크기 조정 차원 이해 315
엑스트라 라지 스케일링 316
제어 플레인 316
워크로드 및 노드 버스팅 제한 316
노드 및 포드를 안전하게 축소 317
Kubectl을 실행할 때 클라이언트 측 캐시 사용 317
kubectl 압축 비활성화 318
샤드 클러스터 오토스케일러 318
API 우선 순위 및 공정성 320

API 서버에서 리소스 검색 327
데이터 영역 333
자동 노드 Autoscaling 333
다양한 EC2 인스턴스 유형 사용 333
API 서버 로드를 줄이기 위해 더 큰 노드 선호 334
일관된 워크로드 성능을 위해 유사한 노드 크기 사용 335
컴퓨팅 리소스를 효율적으로 사용 336
Amazon Machine Image(AMI) 업데이트 자동화 336
컨테이너에 여러 EBS 볼륨 사용 337
워크로드가 EBS 볼륨을 사용하는 경우 EBS 연결 제한이 낮은 인스턴스 방지 338
디스크에 대한 불필요한 로깅 비활성화 339
OS 업데이트 속도가 필요한 경우 패치 인스턴스 적용 339
클러스터 서비스 340
CoreDNS 규모 조정 340
Kubernetes 지표 서버를 수직으로 확장 341
CoreDNS lameduck 기간 342
CoreDNS 준비 프로브 342
에이전트 로깅 및 모니터링 342
워크로드 343
포드 네트워킹에 IPv6 사용 344
네임스페이스당 서비스 수 제한 344
Elastic Load Balancer 할당량 이해 344
Route 53, Global Accelerator 또는 CloudFront 사용 345
엔드포인트 대신 EndpointSlices 사용 345
가능하면 변경 불가능한 보안 암호 및 외부 보안 암호 사용 346
배포 기록 제한 347
기본적으로 enableServiceLinks 비활성화 347
리소스당 동적 승인 웹후크 제한 347
여러 클러스터의 워크로드 비교 348
조정에 대한 이론 348
노드와 이탈률 비교 348
초당 쿼리로 생각하기 349
분산 구성 요소 조정 349
업스트림 및 다운스트림 병목 현상 350
지표별 규모 조정 352
컨트롤 플레인 모니터링 354

API 서버 354
문제는 어디에 있나요? 354
스케줄러 360
Kube 컨트롤러 관리자 361
ETCD 363
노드 효율성 및 크기 조정 364
노드 선택 364
노드 빈 패키징 366
사용률과 포화도 비교 373
CPU 제한 설정 381
Kubernetes SLOs 384
Kubernetes SLOs 384
Kubernetes SLI 지표 386
클러스터SLOs 390
알려진 제한 및 Service Quotas 391
기타 AWS Service Quotas 391
AWS 요청 조절 396
기타 알려진 제한 397
클러스터 업그레이드 398
개요 398
업그레이드 전 398
클러스터up-to-date 유지 399
EKS 릴리스 일정 검토 400
공동 책임 모델을 클러스터 업그레이드에 적용하는 방법 이해 400
클러스터 인플레이스 업그레이드 400
컨트롤 플레인 및 데이터 플레인을 순서대로 업그레이드 400
EKS 설명서를 사용하여 업그레이드 체크리스트 생성 401
Kubernetes API를 사용하여 추가 기능 및 구성 요소 업그레이드 402
업그레이드하기 전에 기본 EKS 요구 사항 확인 403
사용 가능한 IP 주소 확인 403
EKS IAM 역할 확인 404
EKS 추가 기능으로 마이그레이션 405
컨트롤 플레인을 업그레이드하기 전에 제거된 API 사용량 식별 및 해결 406
클러스터 인사이트 406
Kube-no-trouble 408
Pluto 409

리소스 409
Kubernetes 워크로드를 업데이트합니다. kubectl-convert를 사용하여 매니페스트 업데이트 410
데이터 영역이 업그레이드되는 동안 워크로드의 가용성을 보장하기 위해 PodDisruptionBudgets 및 topologySpreadConstraints 구성 411
관리형 노드 그룹 또는 Karpenter를 사용하여 데이터 영역 업그레이드 간소화 412
기존 노드 및 컨트롤 플레인과의 버전 호환성 확인 413
Karpenter 관리형 노드에 대한 노드 만료 활성화 413
Karpenter 관리형 노드에 드리프트 기능 사용 413
eksctl을 사용하여 자체 관리형 노드 그룹의 업그레이드 자동화 414
업그레이드하기 전에 클러스터 백업 414
컨트롤 플레인 업그레이드 후 Fargate 배포 다시 시작 415
인플레이스 클러스터 업그레이드의 대안으로 블루/그린 클러스터 평가 415
Kubernetes 프로젝트에서 계획된 주요 변경 사항 추적 - 미리 생각하기 416
기능 제거에 대한 특정 지침 416
1.25에서 Dockershim 제거 - Docker 소켓용 감지기(DDS) 사용 416
1.25에서 PodSecurityPolicy 제거 - 포드 보안 표준 또는 policy-as-code 솔루션으로 마이그레이션 416
1.23에서 인트리 스토리지 드라이버 사용 중단 - 컨테이너 스토리지 인터페이스(CSI) 드라이버로 마이그레이션 417
추가 리소스 417
ClowdHaus EKS 업그레이드 지침 417
GoNoGo 417
비용 최적화 418
일반 가이드라인 418
EKS 비용 최적화 모범 사례 418
이 설명서의 사용법 3
주요 AWS 서비스 및 Kubernetes 기능 419
Feedback 5
프레임워크 420
See 원칙: 측정 및 책임 420
절감 원칙: 비용 최적화 421
계획 원칙: 계획 및 예측 422
실행 원칙 422
참조 76
인식 423
권장 사항 58

기타 도구 430
컴퓨팅 431
워크로드 크기 조정 431
소비 감소 432
미사용 용량 감소 433
컴퓨팅 용량 유형 최적화 438
컴퓨팅 사용량 최적화 442
Network 443
포드 간 통신 443
Load Balancer서와 포드 통신 455
컨테이너 레지스트리에서 데이터 전송 457
인터넷 및 AWS 서비스로 데이터 전송 458
VPCs 간 데이터 전송 460
Service Mesh 사용 462
추가 리소스 84
스토리지 471
개요 174
임시 볼륨 471
영구 볼륨 472
기타 고려 사항 478
관찰성 479
소개 479
로깅 479
EKS 제어 플레인 479
EKS 데이터 영역 480
Metrics 484
추적 487
추가 리소스: 84
Windows 490
AMI 관리 490
자체 Amazon EKS 최적화 Windows AMI 관리 491
사용자 지정 EKS 최적화 AMIs 대한 더 빠른 시작 구성 492
사용자 지정 AMIs에서 Windows 기본 계층 캐싱 493
블로그 게시물 494
Windows 컨테이너용 gMSA 494
gMSA 계정이란? 494

Windows 컨테이너 및 gMSA 사용 사례 494
Windows Server 강화 496
Windows Server Core를 사용하여 공격 표면 줄이기 497
RDP 연결 방지 497
Amazon Inspector – 498
Amazon GuardDuty 499
Windows용 Amazon EC2의 보안 499
Windows 이미지 스캔 500
Windows 버전 및 라이선스 500
Windows Server 버전 500
라이선싱 501
로깅 501
로깅 권장 사항 501
Windows 컨테이너 모니터링 502
Windows 네트워킹 506
Windows 컨테이너 네트워킹 개요 506
IP 주소 관리 507
컨테이너 네트워크 인터페이스(CNI) 옵션 511
네트워크 정책 511
메모리 및 시스템 관리 511
시스템 및 kubelet 메모리 예약 511
Windows 컨테이너 메모리 요구 사항 512
결론 513
인프라 관리 513
Windows 이미지 푸시 및 가져오기 513
레퍼런스 515
일정 예약 515
노드PODs 할당 모범 사례 515
OS별 워크로드가 적절한 컨테이너 호스트에 도달하도록 보장 516
동일한 클러스터에서 여러 Windows 빌드 처리 517
RuntimeClass를 사용하여 포드 매니페스트에서 NodeSelector 및 허용치 간소화 518
관리형 노드 그룹 지원 519
추가 설명서 520
Windows 컨테이너용 포드 보안 520
스토리지 옵션 522
트리 내 볼륨 플러그인과 out-of-tree 볼륨 플러그인이란 무엇입니까? 522

Windows용 인트리 볼륨 플러그인 523
Windows용 Out-of-tree 527
Amazon FSx for Windows File Server 527
Windows 컨테이너 이미지 강화 528
1. 로컬 보안 정책 및 레지스트리를 사용하여 계정 정책(암호 또는 잠금) 구성 529
2. 감사 정책 531
3. Windows 컨테이너에 대한 IIS 보안 모범 사례 533
4. 최소 권한의 원칙 538
최종 생각: 오늘날의 위협 환경에서 Windows 컨테이너를 보호해야 하는 이유 541
하이브리드 542
네트워크 연결 해제 542
모범 사례 542
Kubernetes 포드 장애 조치 547
애플리케이션 네트워크 트래픽 553
호스트 자격 증명 558
AI/ML 560
Feedback 560
컴퓨팅 560
GPU 리소스 최적화 및 비용 관리 560
노드 복원력 및 훈련 작업 관리 573
애플리케이션 조정 및 성능 575
고급 GPU 관리를 위한 동적 리소스 할당 577
네트워킹 623
노드 간 통신이 많은 애플리케이션의 경우 더 높은 네트워크 대역폭 또는 탄력적 패브릭 어댑터 고려 623
사용 가능한 IP 주소 수를 늘려 포드 시작 시간 단축 623
보안 624
보안 및 규정 준수 624
스토리지 625
데이터 관리 및 스토리지 625
관찰성 637
모니터링 및 관찰성 637
관찰성 및 지표 638
성능 652
ML 아티팩트, 서비스 프레임워크 및 시작 최적화 관리 652
컨테이너 이미지 풀 타임 최적화 654

kubelet 및 컨테이너형 스토리지용 NVMe 고려 657
기여 658
기존 기여자에 대한 요약 658
로컬 편집 환경 설정 658
리포지토리 포크 및 복제 658
VS Code Workspace 열기 658
파일 편집 659
풀 요청 제출 659
github.dev 웹 기반 편집기 사용 659
단일 페이지 편집 660
페이지 ID 보기 및 설정 660
페이지 ID 설정 660
새 페이지 생성 661
페이지 메타데이터 생성 661
목차에 추가 662
이미지 삽입 662
Vale을 사용하여 스타일 확인 662
로컬 미리 보기 빌드 663
AsciiDoc 치트 시트 663
기본 형식 지정 663
헤더 663
Lists 664
링크 664
이미지 664
코드 블록 664
테이블 664
주의 문구 665
포함 파일 665
..... dclxvi

Amazon EKS 모범 사례 가이드

EKS 모범 사례 가이드에 오신 것을 환영합니다. 이 프로젝트의 주요 목표는 Amazon EKS의 2일차 작업에 대한 일련의 모범 사례를 제공하는 것입니다. 이 지침을 GitHub에 게시하여 빠르게 반복하고, 다양한 문제에 대한 시기 적절하고 효과적인 권장 사항을 제공하고, 더 광범위한 커뮤니티의 제안을 쉽게 통합할 수 있도록 하기로 결정했습니다.

현재 다음 주제에 대한 가이드를 게시했습니다.

- 보안 모범 사례
- 신뢰성 모범 사례
- 클러스터 오토 스케일링 모범 사례: Karpenter
- 클러스터 오토 스케일링 모범 사례: cluster-autoscaler
- 클러스터 Autoscaling 모범 사례: EKS Auto Mode
- 네트워킹 모범 사례
- 확장성 모범 사례
- 클러스터 업그레이드 모범 사례
- 비용 최적화 모범 사례
- Windows 컨테이너 실행 모범 사례
- 하이브리드 배포 모범 사례
- AI/ML 워크로드 실행 모범 사례

또한이 가이드의 몇 가지 권장 사항을 확인하기 위해 hardeneks라는 Python 기반 CLI(명령줄 인터페이스)를 오픈 소스로 제공했습니다.

향후 성능, 비용 최적화 및 운영 우수성에 대한 모범 사례 지침을 게시할 예정입니다.

관련 가이드

AWS는 EKS 사용 설명서 외에도 EKS 구현에 도움이 될 수 있는 몇 가지 다른 가이드를 게시했습니다.

- EMR 컨테이너 모범 사례 가이드
- EKS의 데이터
- AWS 관찰성 모범 사례

- Terraform용 Amazon EKS 블루프린트
- Amazon EKS 블루프린트 빠른 시작

제공

이 가이드에 기여하는 것이 좋습니다. 효과적인 것으로 검증된 사례를 구현한 경우 문제 또는 풀 요청을 열어 공유하십시오. 마찬가지로 이미 게시한 지침에서 오류나 결함이 발견되면 PR을 제출하여 수정하십시오. PRs 제출 지침은 기여 지침에서 확인할 수 있습니다.

보안 모범 사례

이 가이드에서는 위험 평가 및 완화 전략을 통해 비즈니스 가치를 제공하면서 EKS에 의존하는 정보, 시스템 및 자산을 보호하는 방법에 대한 조언을 제공합니다. 여기의 지침은 고객이 모범 사례에 따라 EKS를 구현하는 데 도움이 되도록 AWS가 게시하는 일련의 모범 사례 가이드의 일부입니다. 성능, 운영 우수성, 비용 최적화 및 신뢰성에 대한 가이드는 향후 몇 달 내에 제공될 예정입니다.

이 설명서의 사용법

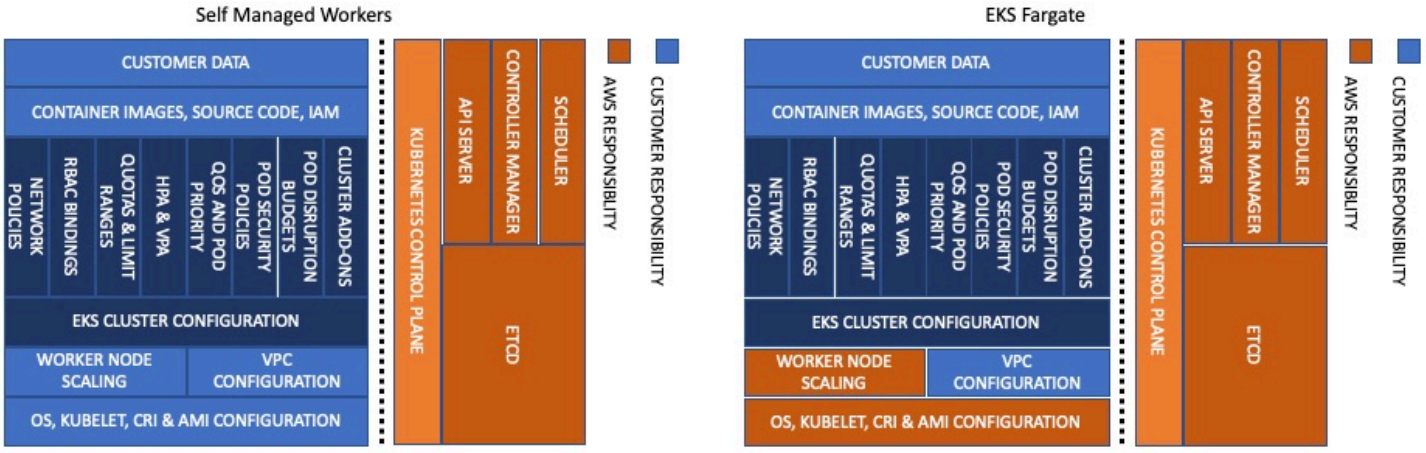
이 가이드는 EKS 클러스터 및 지원하는 워크로드에 대한 보안 제어의 효과를 구현하고 모니터링하는 보안 전문가를 대상으로 합니다. 이 가이드는 더 쉽게 사용할 수 있도록 다양한 주제 영역으로 구성되어 있습니다. 각 주제는 간략한 개요로 시작하여 EKS 클러스터 보안을 위한 권장 사항 및 모범 사례 목록으로 이어집니다. 주제를 특정 순서로 읽을 필요는 없습니다.

공동 책임 모델 이해

보안 및 규정 준수는 EKS와 같은 관리형 서비스를 사용할 때 공동 책임으로 간주됩니다. 일반적으로 AWS는 클라우드의 보안에 대한 책임이 있는 반면, 고객인 사용자는 클라우드의 보안에 대한 책임이 있습니다. EKS를 통해 AWS는 EKS 관리형 Kubernetes 컨트롤 플레인을 관리할 책임이 있습니다. 여기에는 Kubernetes 컨트롤 플레인 노드, ETCD 데이터베이스 및 AWS가 안전하고 신뢰할 수 있는 서비스를 제공하는 데 필요한 기타 인프라가 포함됩니다. EKS 소비자는 IAM, 포드 보안, 런타임 보안, 네트워크 보안 등과 같은이 가이드의 주제에 대해 주로 책임을 집니다.

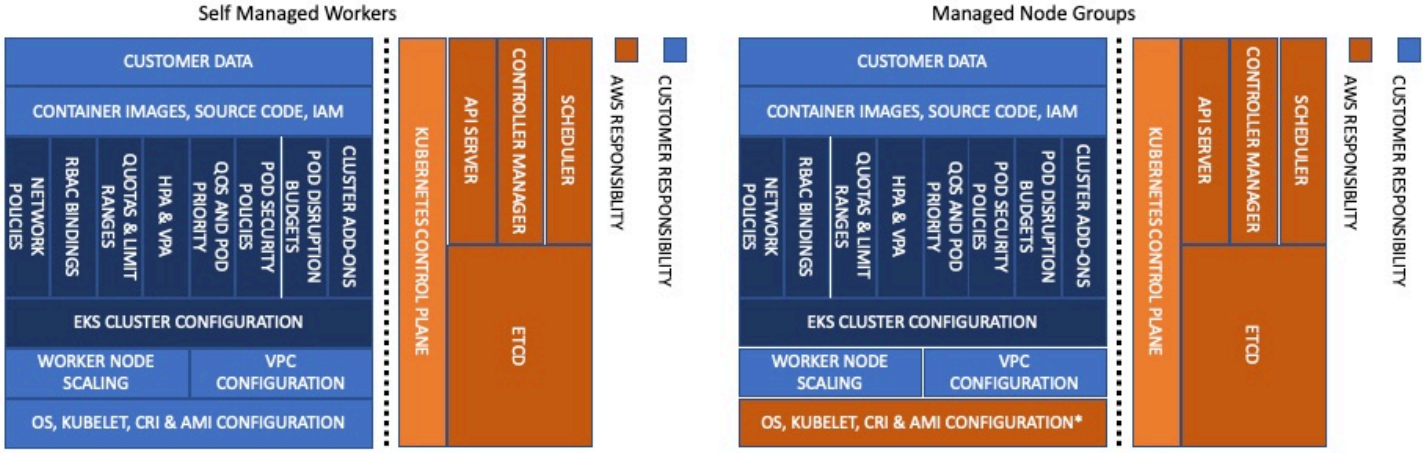
인프라 보안과 관련하여 AWS는 사용자가 자체 관리형 작업자에서 관리형 노드 그룹으로 Fargate로 이동할 때 추가 책임을 집니다. 예를 들어 Fargate를 사용하면 AWS는 포드를 실행하는 데 사용되는 기본 인스턴스/런타임을 보호할 책임이 있습니다.

공동 책임 모델 - Fargate



또한 AWS는 Kubernetes 패치 버전 및 보안 패치를 사용하여 EKS 최적화 AMI를 최신 상태로 유지할 책임이 있습니다. 관리형 노드 그룹(MNG)을 사용하는 고객은 EKS API, CLI, Cloudformation 또는 AWS 콘솔을 통해 노드 그룹을 최신 AMI로 업그레이드할 책임이 있습니다. 또한 Fargate와 달리 MNGs는 인프라/클러스터를 자동으로 조정하지 않습니다. 이는 클러스터-오토스케일러 또는 Karpenter, 네이티브 AWS Autoscaling, SpotInst's Ocean 또는 Atlassian's Escalator와 같은 기타 기술에서 처리할 수 있습니다.

공동 책임 모델 - MNG



시스템을 설계하기 전에 책임과 서비스 공급자(AWS) 간의 경계선을 파악하는 것이 중요합니다.

공동 책임 모델에 대한 자세한 내용은 https://aws.amazon.com/compliance/shared-responsibility-model/ 참조하십시오.

소개

EKS와 같은 관리형 Kubernetes 서비스를 사용할 때 관련된 몇 가지 보안 모범 사례 영역이 있습니다.

- 자격 증명 및 액세스 관리
- 포드 보안
- 런타임 보안
- 네트워크 보안
- 멀티테넌시
- 다중 테넌시를 위한 다중 계정
- 탐지 제어
- 인프라 보안
- 데이터 암호화 및 보안 암호 관리
- 규정 준수
- 인시던트 대응 및 포렌식
- 이미지 보안

시스템 설계의 일환으로 보안 영향과 보안 태세에 영향을 미칠 수 있는 사례를 고려해야 합니다. 예를 들어 리소스 세트에 대해 작업을 수행할 수 있는 사용자를 제어해야 합니다. 또한 보안 인시던트를 신속하게 식별하고, 시스템 및 서비스를 무단 액세스로부터 보호하고, 데이터 보호를 통해 데이터의 기밀성과 무결성을 유지할 수 있는 기능이 필요합니다. 보안 인시던트에 대응하기 위한 잘 정의되고 리허스된 프로세스 세트를 갖추면 보안 태세도 향상됩니다. 이는 금전적 손해 방지 또는 규제 의무 준수 등 목표 달성을 뒷받침하는 중요한 도구이자 기법입니다.

AWS는 광범위한 보안 인식 고객의 피드백을 기반으로 발전한 풍부한 보안 서비스 세트를 제공하여 조직이 보안 및 규정 준수 목표를 달성할 수 있도록 지원합니다. 고객은 매우 안전한 기반을 제공함으로써 " 차별화되지 않은 과중한 업무 처리"에 소요되는 시간을 줄이고 비즈니스 목표를 달성하는 데 더 많은 시간을 할애할 수 있습니다.

Feedback

이 가이드는 광범위한 EKS/Kubernetes 커뮤니티에서 직접적인 피드백과 제안을 수집하기 위해 GitHub에서 릴리스됩니다. 가이드에 포함해야 한다고 생각되는 모범 사례가 있는 경우 GitHub 리포지토리에 문제를 제기하거나 PR을 제출하세요. 새로운 기능이 서비스에 추가되거나 새로운 모범 사례가 발전할 때 가이드를 정기적으로 업데이트하는 것이 목적입니다.

참고 문헌

Security Audit Working Group에서 후원하는 Kubernetes 보안 백서는 보안 실무자가 건전한 설계 및 구현 결정을 내릴 수 있도록 지원하기 위해 Kubernetes 공격 표면 및 보안 아키텍처의 주요 측면을 설명합니다.

CNCF는 클라우드 네이티브 보안에 대한 백서도 게시했습니다. 이 백서에서는 기술 환경이 어떻게 발전했는지 살펴보고 DevOps 프로세스 및 애자일 방법론에 맞는 보안 사례 채택을 지지합니다.

도구 및 리소스

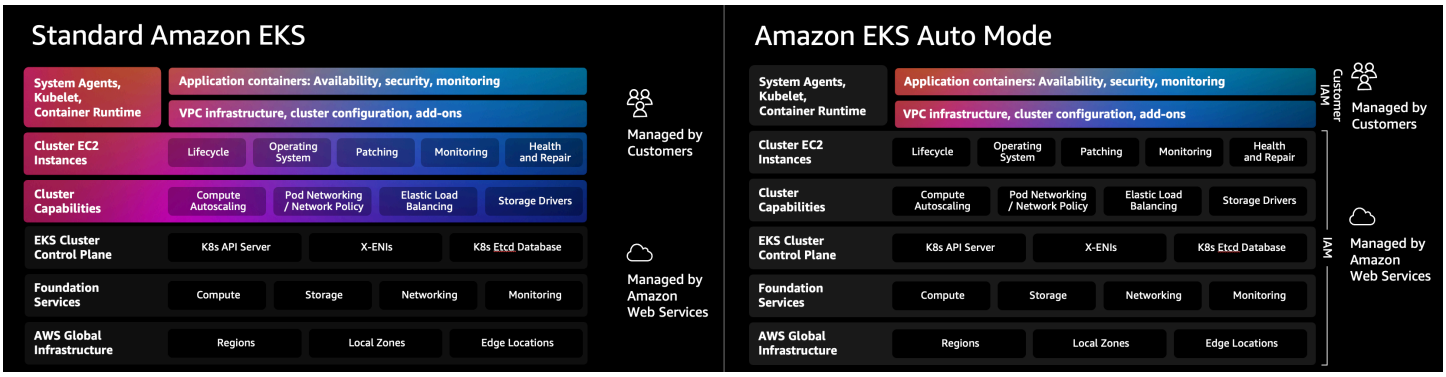
Amazon EKS Security Immersion 워크숍

EKS Auto Mode - 보안

> **Tip**
>
> Amazon EKS Auto Mode에 대한 실습 경험을 얻고 싶으신가요? AWS Connected Community에 가입하여 AWS 전문가가 진행하는 예정된 가상 Kubernetes 워크숍에 참여하세요.

Amazon EKS Auto Mode는 작업자 노드와 코어 클러스터 구성 요소를 포함하도록 AWS의 보안 관리를 컨트롤 플레인 이상으로 확장하여 향상된 보안 기능을 도입합니다. 이 포괄적인 보안 모델은 조직이 강력한 보안 태세를 유지하면서 운영 오버헤드를 줄이는 데 도움이 됩니다.

공동 책임 모델 - EKS Auto Mode



EKS Auto Mode의 주요 보안 개선 사항은 다음과 같습니다.

- 공격 표면이 감소하여 컨테이너 최적화 OS 최소화
- EC2 관리형 인스턴스를 통한 보안 모범 사례 적용
- 필수 노드 교체를 통한 자동 보안 패치 관리
- 기본 제공 노드 격리 및 보안 경계
- 최소한의 필수 권한으로 간소화된 IAM 통합

EKS Auto Mode는 기본적으로 이러한 보안 제어를 적용하므로 조직이 보안 및 규정 준수 요구 사항을 충족하는 동시에 클러스터 작업을 간소화할 수 있습니다. 이 접근 방식은 defense-in-depth 원칙에 부합하여 인프라, 노드 및 워크로드 수준에서 여러 계층의 보안 제어를 제공합니다.

> **Important**
>
> EKS Auto Mode는 향상된 보안 기능을 제공하지만 조직은 여전히 애플리케이션 계층에서 적절한 보안 제어를 구현하고 클러스터에서 실행되는 워크로드에 대한 보안 모범 사례를 따라야 합니다.

보안 아키텍처

EKS Auto Mode는 컨트롤 플레인에서 개별 노드에 이르기까지 EKS 인프라의 여러 계층에 걸쳐 보안 제어를 구현합니다. 이 아키텍처를 이해하는 것은 EKS 클러스터를 효과적으로 관리하고 보호하는 데 매우 중요합니다.

컨트롤 플레인 보안

EKS Auto Mode의 EKS 컨트롤 플레인은 새로운 보안 기능을 추가하면서 기존 EKS 클러스터와 동일한 높은 보안 표준을 유지합니다.

- 봉투 암호화: 모든 Kubernetes API 데이터는 봉투 암호화를 사용하여 자동으로 암호화됩니다.
- KMS 통합: AWS 소유 키 또는 고객 관리형 키(CMK) 옵션과 함께 Kubernetes KMS 공급자 v2와 함께 AWS KMS를 사용합니다.
- 향상된 구성 요소 관리: Auto Scaling, ENI 관리 및 EBS 컨트롤러와 같은 중요한 구성 요소는 클러스터 외부로 이동하여 AWS에서 관리합니다.
- 보안 제어 및 감사 기능 개선: 표준 EKS 클러스터 이외의 EKS Auto Mode 필수 권한은 개별 노드 역할이 아닌 클러스터 IAM 역할을 통해 완전히 관리됩니다.

IAM 통합 및 액세스 관리

EKS Auto Mode는 EKS 액세스 항목 및 EKS Pod Identity를 통해 AWS Identity and Access Management(IAM)와의 향상된 통합을 제공합니다.

클러스터 액세스 관리

EKS Auto Mode는 클러스터 액세스 관리(CAM) API를 통해 클러스터 액세스 관리를 개선했습니다.

- 를 통한 표준화된 인증 모드 `EKS_API`
- API 기반 액세스 관리를 통한 보안 강화
- 액세스 항목 및 액세스 정책을 사용한 간소화된 액세스 제어

액세스 항목을 생성하여 클러스터 액세스를 관리할 수 있습니다.

```
aws eks create-access-entry \
    --cluster-name ${EKS_CLUSTER_NAME} \
    --principal-arn arn:aws:iam::${ACCOUNT_ID}:role/${IAM_ROLE_NAME} \
    --type STANDARD
```

> **Important**
>
> `CONFIG_MAP_AND_API` 인증 모드로 EKS Auto Mode 클러스터를 생성하는 것은 여전히 가능하지만 이는 표준 접근 방식이 아니며 새 클러스터에 기본 `API` 인증 모드를 사용하는 것이 좋습니다. `API` 기반 인증은 레거시 ConfigMap 기반 접근 방식에 비해 향상된 보안과 간소화된 액세스 관리를 제공합니다.

EKS Pod Identity

EKS Auto Mode에는 Pod Identity Agent가 이미 배포되어 있으므로 포드에 AWS IAM 권한을 부여하는 간소화된 방법이 가능합니다.

- OIDC 공급자 구성 없이 간소화된 IAM 권한 관리
- IRSA에 비해 운영 오버헤드 감소
- 세션 태그 지정 및 ABAC 지원을 통한 보안 강화

```
aws eks create-pod-identity-association \
```

```
  --cluster-name ${EKS_CLUSTER_NAME} \
  --role-arn arn:aws:iam::${AWS_ACCOUNT_ID}:role/${IAM_ROLE_NAME} \
  --namespace ${NAMESPACE} \
  --service-account ${SERVICE_ACCOUNT_NAME}
```

> **Important**
>
> 포드 자격 증명은 IRSA에 비해 향상된 보안 기능과 간소화된 관리를 제공하므로 EKS Auto Mode에서 IAM 권한에 권장되는 접근 방식입니다.

노드 IAM 역할

EKS Auto Mode는 EKS Auto Mode 노드가 작동하는 데 필요한 권한만 `AmazonEKSWorkerNodeMinimalPolicy` 제공하는 새를 사용합니다. 이러한 권한은 다음과 같습니다.

- 기존 노드 정책에 비해 감소된 권한 세트 제공
- 최소 권한 원칙 준수
- AWS에서 자동으로 관리 및 업데이트

이 최소 정책 접근 방식은 노드와 해당 워크로드에 사용할 수 있는 권한을 제한하여 보안 태세를 개선하는 데 도움이 됩니다.

노드 보안

EKS Auto Mode는 노드 수준에서 몇 가지 중요한 보안 개선을 도입합니다.

EC2 관리형 인스턴스 보안

EKS Auto Mode 노드는 향상된 보안 속성과 함께 Amazon EC2 관리형 인스턴스를 사용합니다.

- AWS의 노드 운영 능력을 손상시킬 수 있는 작업을 방지하는 IAM 적용 제한
- 구성 변경에 노드 교체가 필요한 변경 불가능한 인프라 패턴
- 정기적인 보안 업데이트를 보장하기 위해 21일 이내에 노드 교체 필수
- 제어된 홉 제한이 있는 IMDSv2를 사용하여 인스턴스 메타데이터에 대한 액세스 제한

운영 체제 보안

운영 체제는 다음을 포함하여 EKS Auto Mode에 최적화된 Bottlerocket의 사용자 지정 변형입니다.

- 읽기 전용 루트 파일 시스템
- SELinux는 필수 액세스 제어를 통해 기본적으로 활성화됨
- 고유한 SELinux MCS 레이블을 사용한 자동 포드 격리
- SSH 액세스 및 불필요한 서비스 제거 비활성화
- 노드 교체를 통한 자동화된 보안 패치

노드 구성 요소 보안

노드 구성 요소는 보안 모범 사례로 구성됩니다.

- 보안 기본값으로 구성된 Kubelet
- 컨테이너 런타임 강화 구성
- 자동화된 인증서 관리 및 교체
- 제한된 node-to-control-plane 통신

네트워크 보안

EKS Auto Mode는 클러스터 내부 및 외부 리소스와의 보안 통신을 보장하기 위해 여러 네트워크 보안 기능을 구현합니다.

VPC CNI 네트워크 정책

EKS Auto Mode는 Amazon VPC CNI 플러그인의 기본 Kubernetes 네트워크 정책 지원을 활용합니다.

- 업스트림 Kubernetes 네트워크 정책 API와 통합
- pod-to-pod
- 수신 및 송신 규칙 모두 지원

EKS Auto Mode에서 네트워크 정책 지원을 활성화하려면 `configMap` 매니페스트를 사용하여 VPC CNI 추가 기능을 구성해야 합니다. 다음 예를 참고하세요

```
apiVersion: v1
```

```
kind: ConfigMap
metadata:
  name: amazon-vpc-cni
  namespace: kube-system
data:
  enable-network-policy: "true"
```

또한 다음 그림과 같이 노드 클래스에서 구성된 네트워크 정책 지원을 정의해야 합니다.

```
apiVersion: eks.amazonaws.com/v1
kind: NodeClass
metadata:
  name: example-node-class
spec:
  networkPolicy: DefaultAllow
  networkPolicyEventLogs: Enabled
```

활성화되면 트래픽을 제어하는 네트워크 정책을 생성할 수 있습니다.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

향상된 ENI 관리

EKS Auto Mode는 탄력적 네트워크 인터페이스(ENI) 관리를 위한 향상된 보안을 제공합니다.

- AWS 관리형 ENI 연결 및 구성
- 데이터 트래픽에서 제어 트래픽 분리
- 노드에 필요한 권한이 감소된 자동화된 IP 주소 관리

스토리지 보안

EKS Auto Mode는 임시 스토리지와 영구 스토리지 모두에 향상된 보안 기능을 제공합니다.

임시 스토리지

- 임시 볼륨에 기록된 모든 데이터는 자동으로 암호화됩니다.
- 업계 표준 AES-256 암호화 알고리즘 사용
- 서비스에서 원활하게 처리되는 암호화 및 복호화

EBS 볼륨

- 루트 및 데이터 EBS 볼륨은 항상 암호화됩니다.
- 볼륨은 인스턴스 종료 시 삭제되도록 구성됩니다.
- 암호화를 위한 사용자 지정 KMS 키를 지정하는 옵션이 있습니다.

EFS 통합

- EFS를 사용한 전송 중 암호화 지원
- EFS 파일 시스템의 저장 시 자동 암호화
- 향상된 액세스 제어를 위해 EFS 액세스 포인트와 통합

> **Important**
>
> EFS를 EKS Auto Mode와 함께 사용하는 경우 EKS Auto Mode가 EFS 암호화를 직접 관리하지 않으므로 EFS 파일 시스템 수준에서 적절한 암호화 설정이 구성되어 있는지 확인합니다.

모니터링 및 로깅

EKS Auto Mode는 클러스터의 보안 태세 및 운영 상태에 대한 가시성을 유지하는 데 도움이 되는 향상된 모니터링 및 로깅 기능을 제공합니다.

제어 플레인 로깅

EKS Auto Mode는 표준 EKS와 동일한 컨트롤 플레인 로깅 기능을 유지하지만 향상된 모니터링을 위해 기본적으로 모든 로그를 활성화합니다.

- 로그는 Amazon CloudWatch Logs로 전송됩니다.
- 기본적으로 EKS Auto Mode는 API 서버, 감사, 인증자, 컨트롤러 관리자 및 스케줄러와 같은 모든 컨트롤 플레인 로그를 활성화합니다.

- EKS Auto Mode를 사용하면 클러스터 작업 및 보안 이벤트를 자세히 파악할 수 있습니다.

> **Important**
>
> 컨트롤 플레인 로깅에는 CloudWatch의 로그 스토리지에 대한 추가 비용이 발생합니다. 로깅 전략을 신중하게 고려하여 보안 요구 사항과 비용 관리의 균형을 맞춥니다.

노드 수준 로깅

EKS Auto Mode는 노드 수준 로깅을 개선합니다.

- 시스템 로그는 자동으로 수집되며 CloudWatch Logs를 통해 액세스할 수 있습니다.
- 노드 로그는 노드 종료 후에도 유지되므로 인시던트 후 분석에 도움이 됩니다.
- 노드 수준 보안 이벤트 및 운영 문제에 대한 가시성 향상

Amazon GuardDuty 통합

EKS Auto Mode 클러스터는 향상된 위협 탐지를 위해 Amazon GuardDuty와 원활하게 통합됩니다. 기능은 다음과 같습니다.

- 컨트롤 플레인 감사 로그 자동 스캔
- 워크로드 모니터링에 대해 활성화할 수 있는 런타임 모니터링
- 기존 GuardDuty 조사 결과 및 알림 메커니즘과 통합

Amazon GuardDuty for Kubernetes 감사 로그에서 EKS Auto Mode 보호를 활성화하려면 다음 명령을 실행할 수 있습니다.

```
aws guardduty update-detector \
    --detector-id 12abc34d567e8fa901bc2d34e56789f0 \
    --data-sources '{"Kubernetes":{"AuditLogs":{"Enable":true}}}'
```

런타임 보안을 위한 Amazon GuardDuty 통합

Amazon GuardDuty는 EKS Auto Mode 클러스터에 대한 필수 런타임 보안 모니터링을 제공하여 포괄적인 위협 탐지 및 보안 모니터링 기능을 제공합니다. 이 통합은 잠재적 보안 위협과 악의적인 활동을 실시간으로 식별하는 데 도움이 되므로 특히 중요합니다.

EKS Auto Mode의 주요 GuardDuty 기능

- 런타임 모니터링:
  - 런타임 동작의 지속적인 모니터링
  - 악의적이거나 의심스러운 활동 탐지
  - 잠재적 컨테이너 이스케이프 시도 식별
  - 비정상적인 프로세스 실행 또는 네트워크 연결 모니터링
- Kubernetes별 위협 탐지:
  - 의심스러운 포드 배포 시도 식별
  - 손상된 컨테이너 감지
  - 권한 있는 컨테이너 시작 모니터링
  - 의심스러운 서비스 계정 사용 식별
- 포괄적인 결과 유형:
  - 정책: Kubernetes/* - 보안 모범 사례 위반을 감지합니다.
  - 영향: Kubernetes/* - 잠재적으로 영향을 받는 리소스 식별
  - Discovery:Kubernetes/* - 정찰 활동을 탐지합니다.
  - 실행: Kubernetes/* - 의심스러운 실행 패턴 식별
  - 지속성: Kubernetes/* - 잠재적 영구 위협을 탐지합니다.

Amazon GuardDuty for Kubernetes 감사 로그 및 런타임 모니터링에서 EKS Auto Mode 보호를 활성화하려면 다음 명령을 실행할 수 있습니다.

```
aws guardduty update-detector \
    --detector-id 12abc34d567e8fa901bc2d34e56789f0 \
    --data-sources '{
        "Kubernetes": {
            "AuditLogs": {"Enable": true},
            "RuntimeMonitoring": {"Enable": true}
        }
    }'
```

> **Important**
>
> GuardDuty 런타임 모니터링은 EKS Auto Mode 클러스터에서 자동으로 지원되므로 노드 수준에서 추가 구성 없이 향상된 보안 가시성을 제공합니다.

GuardDuty 결과 통합

GuardDuty 조사 결과는 자동 응답을 위해 다른 AWS 서비스와 통합할 수 있습니다.

- EventBridge 규칙:

```
{
  "source": ["aws.guardduty"],
  "detail-type": ["GuardDuty Finding"],
  "detail": {
    "type": ["Runtime:Container/*", "Runtime:Kubernetes/*"],
    "severity": [4, 5, 6, 7, 8]
  }
}
```

- Security Hub 통합:

```
# Enable Security Hub integration
aws securityhub enable-security-hub \
    --enable-default-standards \
    --tags '{"Environment":"Production"}' \
    --region us-west-2
```

EKS Auto Mode를 사용하는 GuardDuty 모범 사례

1. 모든 결과 유형 활성화:
   - Kubernetes 감사 로그 모니터링 및 런타임 모니터링 모두 활성화
   - 모든 심각도 수준에 대한 조사 결과 구성
2. 자동 응답 구현:
   - 심각도가 높은 조사 결과에 대한 EventBridge 규칙 생성
   - 중앙 집중식 보안 관리를 위해 AWS Security Hub와 통합

- 적절한 경우 자동 문제 해결 작업 설정

3. 정기 검토 및 튜닝:
   - GuardDuty 조사 결과를 정기적으로 검토
   - 환경에 따라 감지 임계값 조정
   - 새 결과 유형에 따라 응답 절차 업데이트
4. 교차 계정 관리:
   - 중앙 집중식 관리를 위해 GuardDuty 관리자 계정 사용 고려
   - 여러 계정에서 조사 결과 집계 활성화

> **Warning**
>
> GuardDuty는 포괄적인 보안 모니터링을 제공하지만 네트워크 정책, 포드 보안 표준 및 적절한 RBAC 구성과 같은 다른 보안 제어를 포함하는 defense-in-depth 전략의 일부여야 합니다.

자주 묻는 질문(FAQ)

Q: 보안 측면에서 EKS Auto Mode는 표준 EKS와 어떻게 다릅니까? A: EKS Auto Mode는 EC2 관리형 인스턴스, 자동 패치 적용, 필수 노드 교체 및 내장 보안 제어를 통해 향상된 보안을 제공합니다. AWS가 더 많은 보안 측면을 관리하도록 하여 강력한 보안 태세를 유지하면서 운영 오버헤드를 줄입니다.

Q: EKS Auto Mode에서 기존 보안 도구 및 정책을 계속 사용할 수 있나요? A: 예, EKS Auto Mode는 대부분의 기존 보안 도구 및 정책과 호환됩니다. 그러나 일부 노드 수준 보안 도구는 EKS Auto Mode 노드의 관리형 특성으로 인해 적응이 필요할 수 있습니다.

Q: EKS Auto Mode에서 보안 에이전트 및 모니터링 도구를 배포하려면 어떻게 해야 하나요? A: EKS Auto Mode에서는 보안 에이전트 및 모니터링 도구를 노드 OS에 직접 설치하지 않고 Kubernetes 워크로드(일반적으로 모든 노드에 기본적으로 포드의 인스턴스 하나를 배포하는 DaemonSets)로 배포해야 합니다. 이 접근 방식은 EKS Auto Mode의 변경 불가능한 인프라 모델과 일치합니다. 예제:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: security-agent
  namespace: security
spec:
  selector:
```

```
    matchLabels:
      app: security-agent
  template:
    metadata:
      labels:
        app: security-agent
    spec:
      containers:
      - name: security-agent
        image: security-vendor/agent:latest
        securityContext:
          privileged: false
          # Use specific capabilities instead of privileged mode
          capabilities:
            add: ["NET_ADMIN", "SYS_ADMIN"]
```

Q: 타사 보안 솔루션이 EKS Auto Mode와 호환되나요? A: 널리 사용되는 여러 타사 보안 솔루션이 EKS Auto Mode를 지원하도록 업데이트되었지만 EKS Auto Mode를 지원하려면 업데이트된 버전 또는 특정 배포 구성이 필요할 수 있으므로 항상 보안 공급업체에 특정 버전 및 배포 요구 사항을 확인하는 것이 좋습니다.

Q: EKS Auto Mode에서 보안 에이전트의 제한 사항은 무엇입니까? A: 주요 제한 사항은 다음과 같습니다.

- 노드의 운영 체제를 수정할 수 있는 직접 액세스 권한 없음
- 노드 교체 간 지속성 없음
- 컨테이너 기반 배포와 호환되어야 합니다.
- 노드 불변성을 준수해야 함
- 다른 권한 구성이 필요할 수 있습니다.
- 노드에 대한 모든 지속적인 변경은 `NodePools` 및 `NodeClasses` 리소스를 통해 수행해야 합니다.

> **Note**
>
> EKS Auto Mode는 보안 도구 배포 전략을 조정해야 할 수 있지만 이러한 변경으로 인해 클라우드 네이티브 모범 사례에 맞는 보다 유지 관리 가능하고 안전한 구성이 발생하는 경우가 많습니다. EKS Auto Mode는 관리하는 대부분의 기능을 완전히 맡을 것으로 예상합니다. 따라서 이러한 기능에 대한 수동 변경 사항을 가져올 수 있는 경우 EKS Auto Mode에서 이를 덮어쓰거나 삭제할 수 있습니다.

Q: EKS Auto Mode에서 사용자 지정 AMIs 사용할 수 있나요? A: 현재 EKS Auto Mode는 사용자 지정 AMIs 지원하지 않습니다. 이는 AWS가 공동 책임 모델의 일부로 노드의 보안, 패치 적용 및 유지 관리를 관리하므로 설계되었습니다. EKS Auto Mode 노드는 AWS에서 최적화하고 유지 관리하는 Bottlerocket의 특수 변형을 사용합니다.

Q: EKS Auto Mode에서 노드는 얼마나 자주 자동으로 교체되나요? A: EKS Auto Mode의 노드 수명은 최대 21일입니다. 이 제한 전에 자동으로 교체되므로 정기적인 보안 업데이트 및 패치 애플리케이션이 보장됩니다.

Q: 문제 해결을 위해 EKS Auto Mode 노드로 SSH할 수 있습니까? A: 아니요, EKS Auto Mode에서는 직접 SSH 액세스를 사용할 수 없습니다. 대신 NodeDiagnostic Custom Resource Definition(CRD)을 사용하여 시스템 로그를 수집하고 정보를 디버깅할 수 있습니다.

Q: EKS Auto Mode에서 네트워크 정책 지원이 기본적으로 활성화되어 있나요? A: 지금은 VPC CNI 추가 기능 구성을 통해 네트워크 정책 지원을 명시적으로 활성화해야 합니다. 활성화되면 표준 Kubernetes 네트워크 정책을 사용할 수 있습니다.

Q: EKS Auto Mode에서 IRSA 또는 Pod Identity를 사용해야 합니까? A: 둘 다 지원되지만 포드 자격 증명은 이미 포드 자격 증명 보안 에이전트 추가 기능을 포함하고 향상된 보안 기능과 간소화된 관리를 제공하므로 EKS Auto Mode에서 권장되는 접근 방식입니다.

Q: EKS Auto Mode에서 aws-auth ConfigMap을 계속 사용할 수 있나요? A: `aws-auth` ConfigMap은 더 이상 사용되지 않는 기능입니다. 보안 강화 및 액세스 관리 간소화를 위해 API 기반 인증의 기본 접근 방식을 사용하는 것이 좋습니다.

Q: EKS Auto Mode에서 보안 이벤트를 모니터링하려면 어떻게 해야 하나요? A: EKS Auto Mode는 GuardDuty, CloudWatch 및 CloudTrail을 포함한 여러 모니터링 솔루션과 통합됩니다. GuardDuty는 특히 EKS 워크로드를 위한 향상된 런타임 보안 모니터링을 제공합니다.

Q: EKS Auto Mode 노드에서 로그를 수집하려면 어떻게 해야 합니까? A: 로그를 S3 버킷에 자동으로 업로드하는 NodeDiagnostic CRD를 사용합니다. CloudWatch Container Insights 및 AWS Distro for OpenTelemetry를 사용할 수도 있습니다.

> **Note**
>
> 이 FAQ 섹션은 새로운 기능이 EKS Auto Mode에 추가되고 고객으로부터 일반적인 질문을 받으면 정기적으로 업데이트됩니다.

자격 증명 및 액세스 관리

Identity and Access Management(IAM)는 인증 및 권한 부여라는 두 가지 필수 기능을 수행하는 AWS 서비스입니다. 인증에는 자격 증명 확인이 수반되는 반면, 권한 부여는 AWS 리소스에서 수행할 수 있는 작업을 관리합니다. AWS 내에서 리소스는 EC2와 같은 다른 AWS 서비스이거나 IAM 사용자 또는 역할과 같은 AWS 보안 주체일 수 있습니다. 리소스가 수행할 수 있는 작업을 관리하는 규칙은 IAM 정책으로 표시됩니다.

EKS 클러스터에 대한 액세스 제어

Kubernetes 프로젝트는 베어러 토큰, X.509 인증서, OIDC 등 kube-apiserver 서비스에 대한 요청을 인증하기 위한 다양한 전략을 지원합니다. EKS는 현재 웹후크 토큰 인증, 서비스 계정 토큰 및 2021년 2월 21일부터 OIDC 인증을 기본적으로 지원합니다.

웹후크 인증 전략은 보유자 토큰을 확인하는 웹후크를 호출합니다. EKS에서 이러한 보유자 토큰은 `kubectl` 명령을 실행할 때 AWS CLI 또는 aws-iam-authenticator 클라이언트에 의해 생성됩니다. 명령을 실행하면 토큰이 kube-apiserver로 전달되어 인증 웹후크로 전달됩니다. 요청이 잘 구성된 경우 Webhook는 토큰 본문에 포함된 미리 서명된 URL을 호출합니다. 이 URL은 요청의 서명을 검증하고 사용자 계정, Arn 및 UserId와 같은 사용자에 대한 정보를 kube-apiserver에 반환합니다.

인증 토큰을 수동으로 생성하려면 터미널 창에 다음 명령을 입력합니다.

```
aws eks get-token --cluster-name <cluster_name> --region <region>
```

출력은 다음과 유사해야 합니다.

```
{
    "kind": "ExecCredential",
    "apiVersion": "client.authentication.k8s.io/v1alpha1",
    "spec": {},
    "status": {
        "expirationTimestamp": "2024-12-20T17:38:48Z",
        "token": "k8s-aws-
v1.aHR0cHM6Ly9zdHMudXMtd2VzdC0yLmFtYXpvbmF3cy5jb20vP0FjdGlvbj1HZ...."
    }
}
```

프로그래밍 방식으로 토큰을 가져올 수도 있습니다. 다음은 Go로 작성된 예제입니다.

```
package main
```

```
import (
  "fmt"
  "log"
  "sigs.k8s.io/aws-iam-authenticator/pkg/token"
)

func main()  {
  g, _ := token.NewGenerator(false, false)
  tk, err := g.Get("<cluster_name>")
  if err != nil {
    log.Fatal(err)
  }
  fmt.Println(tk)
}
```

출력은 다음과 유사해야 합니다.

```
{
  "kind": "ExecCredential",
  "apiVersion": "client.authentication.k8s.io/v1alpha1",
  "spec": {},
  "status": {
    "expirationTimestamp": "2020-02-19T16:08:27Z",
    "token": "k8s-aws-
v1.aHR0cHM6Ly9zdHMuYW1hem9uYXdzLmNvbS8_QWN0aW9uPUdldENhbGxlcklkZW50aXR5JlZlcnNpb249MjAxMS0wNi0x
  }
}
```

각 토큰은 로 시작하고 그 `k8s-aws-v1.` 뒤에 base64 인코딩 문자열이 옵니다. 디코딩할 때 문자열은 다음과 비슷한 것과 유사해야 합니다.

```
https://sts.amazonaws.com/?Action=GetCallerIdentity&Version=2011-06-15&X-
Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=XXXXJPFRILKNSRC2W5QA
%2F20200219%2Fus-xxxx-1%2Fsts%2Faws4_request&X-Amz-Date=20200219T155427Z&X-
Amz-Expires=60&X-Amz-SignedHeaders=host%3Bx-k8s-aws-id&X-Amz-
Signature=XXXf8f3285e320ddb5e683a5c9a405301ad76546f24f28111fdad09cf648a393
```

토큰은 Amazon 자격 증명과 서명을 포함하는 미리 서명된 URL로 구성됩니다. 자세한 내용은 https://docs.aws.amazon.com/STS/latest/APIReference/API_GetCallerIdentity.html 참조하십시오.

토큰의 TTL(Time To Live)은 15분이며, 그 이후에는 새 토큰을 생성해야 합니다. 이는와 같은 클라이언트를 사용할 때 자동으로 처리kubectl되지만 Kubernetes 대시보드를 사용하는 경우 새 토큰을 생성하고 토큰이 만료될 때마다 다시 인증해야 합니다.

사용자의 자격 증명이 AWS IAM 서비스에 의해 인증되면 kube-apiserver는 `kube-system` 네임스페이스의 `aws-auth` ConfigMap을 읽어 사용자와 연결할 RBAC 그룹을 결정합니다. `aws-auth` ConfigMap은 IAM 보안 주체, 즉 IAM 사용자 및 역할과 Kubernetes RBAC 그룹 간에 정적 매핑을 생성하는 데 사용됩니다. RBAC 그룹은 Kubernetes RoleBindings 또는 ClusterRoleBindings. Kubernetes 리소스(객체) 모음에 대해 수행할 수 있는 일련의 작업(동사)을 정의한다는 점에서 IAM 역할과 유사합니다.

사용자가 글로벌 STS 엔드포인트로 요청을 보내는 클라이언트를 식별하는 데 도움이 되는 CloudWatch 쿼리

아래에서 CloudWatch 쿼리를 실행하여 sts 엔드포인트를 가져옵니다. stsendpoint가 "sts.amazonaws.com"과 같으면 글로벌 STS 엔드포인트입니다. stsendpoint가 "sts.<region>.amazonaws.com"과 같으면 리전 STS 엔드포인트입니다.

```
fields @timestamp, @message, @logStream, @log,stsendpoint
| filter @logStream like /authenticator/
| filter @message like /stsendpoint/
| sort @timestamp desc
| limit 10000
```

클러스터 액세스 관리자

이제 Amazon EKS 클러스터에 대한 AWS IAM 보안 주체의 액세스를 관리하는 기본 방법인 Cluster Access Manager는 AWS API의 기능이며 EKS v1.23 이상 클러스터(신규 또는 기존)에 대한 옵트인 기능입니다. AWS IAM과 Kubernetes RBACs 간의 자격 증명 매핑을 간소화하여 AWS와 Kubernetes APIs 간에 전환하거나 액세스 관리를 위해 `aws-auth` ConfigMap을 편집할 필요가 없으며 운영 오버헤드를 줄이고 잘못된 구성을 해결하는 데 도움이 됩니다. 또한이 도구를 사용하면 클러스터 관리자가 클러스터를 생성하는 데 사용되는 AWS IAM 보안 주체에 자동으로 부여된 `cluster-admin` 권한을 취소하거나 세분화할 수 있습니다.

이 API는 다음 두 가지 개념에 의존합니다.

- 액세스 항목: Amazon EKS 클러스터에 인증할 수 있는 AWS IAM 보안 주체(사용자 또는 역할)에 직접 연결된 클러스터 자격 증명입니다.

- 액세스 정책: 액세스 항목이 Amazon EKS 클러스터에서 작업을 수행할 수 있는 권한을 제공하는 Amazon EKS별 정책입니다.

시작 시 Amazon EKS는 사전 정의된 및 AWS 관리형 정책만 지원합니다. 액세스 정책은 IAM 엔터티가 아니며 Amazon EKS에서 정의 및 관리합니다.

Cluster Access Manager는 업스트림 RBAC와 API 서버 요청과 관련된 Kubernetes AuthZ 결정의 허용 및 전달(거부 아님)을 지원하는 액세스 정책의 조합을 허용합니다. 업스트림 RBAC 및 Amazon EKS 권한 부여자 모두 요청 평가 결과를 확인할 수 없는 경우 거부 결정이 발생합니다.

이 기능을 통해 Amazon EKS는 세 가지 인증 모드를 지원합니다.

1. CONFIG_MAP는 aws-auth configMap만 계속 사용합니다.
2. API_AND_CONFIG_MAP EKS 액세스 항목 APIs와 aws-auth configMap 모두에서 인증된 IAM 보안 주체를 소싱하여 액세스 항목의 우선순위를 지정합니다. 기존 aws-auth 권한을 액세스 항목으로 마이그레이션하는 데 적합합니다.
3. API는 EKS 액세스 항목 APIs. 이것이 새로운 권장 접근 방식입니다.

시작하기 위해 클러스터 관리자는 Amazon EKS 클러스터를 생성 또는 업데이트하여 기본 인증을 API_AND_CONFIG_MAP 또는 API 메서드로 설정하고 액세스 항목을 정의하여 원하는 AWS IAM 보안 주체에 액세스 권한을 부여할 수 있습니다.

```
$ aws eks create-cluster \
    --name <CLUSTER_NAME> \
    --role-arn <CLUSTER_ROLE_ARN> \
    --resources-vpc-config
 subnetIds=<value>,endpointPublicAccess=true,endpointPrivateAccess=true \
    --logging '{"clusterLogging":[{"types":
["api","audit","authenticator","controllerManager","scheduler"],"enabled":true}]}' \
    --access-config
 authenticationMode=API_AND_CONFIG_MAP,bootstrapClusterCreatorAdminPermissions=false
```

위 명령은 클러스터 생성자의 관리자 권한 없이 이미 Amazon EKS 클러스터를 생성하는 예제입니다.

update-cluster-config 명령을 사용하여 API authenticationMode를 활성화하도록 Amazon EKS 클러스터 구성을 업데이트하고,를 사용하여 기존 클러스터에서 업데이트하려면 먼저 로 업데이트API_AND_CONFIG_MAP한 다음 로 업데이트CONFIG_MAP해야 합니다API. 이러한 작업은 되돌릴 수 없습니다. 즉,에서 API_AND_CONFIG_MAP 또는 API로 CONFIG_MAP전환하거나에서 로 전환할 API_AND_CONFIG_MAP 수 없습니다CONFIG_MAP.

```
$ aws eks update-cluster-config \
    --name <CLUSTER_NAME> \
    --access-config authenticationMode=API
```

API는 클러스터에 대한 액세스를 추가 및 취소하고 지정된 클러스터에 대한 기존 액세스 정책 및 액세스 항목을 검증하는 명령을 지원합니다. 기본 정책은 다음과 같이 Kubernetes RBACs 일치하도록 생성됩니다.

EKS 액세스 정책	Kubernetes RBAC
AmazonEKSClusterAdminPolicy	cluster-admin
AmazonEKSAdminPolicy	admin
AmazonEKSEditPolicy	edit
AmazonEKSViewPolicy	보기

```
$ aws eks list-access-policies
{
    "accessPolicies": [
        {
            "name": "AmazonEKSAdminPolicy",
            "arn": "arn:aws:eks::aws:cluster-access-policy/AmazonEKSAdminPolicy"
        },
        {
            "name": "AmazonEKSClusterAdminPolicy",
            "arn": "arn:aws:eks::aws:cluster-access-policy/AmazonEKSClusterAdminPolicy"
        },
        {
            "name": "AmazonEKSEditPolicy",
            "arn": "arn:aws:eks::aws:cluster-access-policy/AmazonEKSEditPolicy"
        },
        {
            "name": "AmazonEKSViewPolicy",
            "arn": "arn:aws:eks::aws:cluster-access-policy/AmazonEKSViewPolicy"
        }
    ]
}
```

```
$ aws eks list-access-entries --cluster-name <CLUSTER_NAME>

{
    "accessEntries": []
}
```

기본적으로 생성되는 유일한 항목인 클러스터 생성자 관리자 권한 없이 클러스터를 생성할 때는 액세스 항목을 사용할 수 없습니다.

aws-auth ConfigMap(사용되지 않음)

Kubernetes와 AWS 인증을 통합하는 한 가지 방법은 kube-system 네임스페이스에 있는 aws-auth ConfigMap을 사용하는 것입니다. AWS IAM 자격 증명(사용자, 그룹 및 역할) 인증을 Kubernetes 역할 기반 액세스 제어(RBAC) 권한 부여에 매핑할 책임이 있습니다. aws-auth ConfigMap은 프로비저닝 단계에서 Amazon EKS 클러스터에 자동으로 생성됩니다. 처음에는 노드가 클러스터에 조인할 수 있도록 생성되었지만, 언급한 대로이 ConfigMap을 사용하여 IAM 보안 주체에 RBACs 액세스 권한을 추가할 수도 있습니다.

클러스터의 aws-auth ConfigMap을 확인하려면 다음 명령을 사용할 수 있습니다.

```
kubectl -n kube-system get configmap aws-auth -o yaml
```

다음은 aws-auth ConfigMap의 기본 구성 샘플입니다.

```
apiVersion: v1
data:
  mapRoles: |
    - groups:
      - system:bootstrappers
      - system:nodes
      - system:node-proxier
      rolearn: arn:aws:iam::<AWS_ACCOUNT_ID>:role/kube-system-<SELF_GENERATED_UUID>
      username: system:node:{{SessionName}}
kind: ConfigMap
metadata:
  creationTimestamp: "2023-10-22T18:19:30Z"
  name: aws-auth
  namespace: kube-system
```

이 ConfigMap의 기본 세션은 기본적으로 3개의 파라미터로 구성된 mapRoles 블록data의 아래에 있습니다.

- 그룹: IAM 역할을 매핑할 Kubernetes 그룹/그룹입니다. 기본 그룹이거나 `clusterrolebinding` 또는에 지정된 사용자 지정 그룹일 수 있습니다`rolebinding`. 위 예제에서는 시스템 그룹만 선언했습니다.
- rolearn: 다음 형식을 사용하여 Kubernetes 그룹/그룹이 추가하는에 매핑될 AWS IAM 역할의 ARN입니다`arn:<PARTITION>:iam::<AWS_ACCOUNT_ID>:role/role-name`.
- username: AWS IAM 역할에 매핑할 Kubernetes 내 사용자 이름입니다. 사용자 지정 이름일 수 있습니다.

AWS IAM 사용자에 대한 권한을 매핑하여 `aws-auth`ConfigMap의 `data` `mapUsers`에서에 대한 새 구성 블록을 정의하고 userarn에 대한 rolearn 파라미터를 대체할 수도 있지만, 모범 사례로 사용자는 `mapRoles` 항상 대신 사용하는 것이 좋습니다.

권한을 관리하기 위해 Amazon EKS 클러스터에 대한 액세스 권한을 추가하거나 제거하는 `aws-auth` ConfigMap을 편집할 수 있습니다. `aws-auth` ConfigMap을 수동으로 편집할 수 있지만, 이는 매우 민감한 구성이며 부정확한 구성은 Amazon EKS 클러스터 외부에서 사용자를 잠글 수 `eksctl`있으므로와 같은 도구를 사용하는 것이 좋습니다. 자세한 내용은 아래 aws-auth ConfigMap을 변경하려면 도구 사용 하위 섹션을 참조하세요.

ConfigMap 기반 액세스 관리의 이점

1. 잘못된 구성의 위험 감소: 직접 API 기반 관리는 수동 ConfigMap 편집과 관련된 일반적인 오류를 제거합니다. 이렇게 하면 클러스터에서 사용자를 잠글 수 있는 실수로 인한 삭제 또는 구문 오류를 방지하는 데 도움이 됩니다.
2. 향상된 최소 권한 원칙: 클러스터 생성자 자격 증명에서 클러스터 관리자 권한의 필요성을 제거하고 보다 세분화되고 적절한 권한 할당을 허용합니다. 휴지통 사용 사례에 대해이 권한을 추가하도록 선택할 수 있습니다.
3. 향상된 보안 모델: 액세스 항목이 적용되기 전에 액세스 항목에 대한 기본 제공 검증을 제공합니다. 또한는 인증을 위해 AWS IAM과의 통합을 강화합니다.
4. 간소화된 작업: AWS 네이티브 도구를 통해 권한을 관리하는 보다 직관적인 방법을 제공합니다.

클러스터 액세스 권장 사항

IAM Identity Center와 CAM API 결합

- 간소화된 관리: 클러스터 액세스 관리 API를 IAM Identity Center와 함께 사용하면 관리자가 다른 AWS 서비스와 함께 EKS 클러스터 액세스를 관리할 수 있으므로 다른 인터페이스 간에 전환하거나 ConfigMaps 수동으로 편집할 필요가 줄어듭니다.
- 액세스 항목을 사용하여 클러스터 외부에서 IAM 위탁자의 Kubernetes 권한을 관리합니다. EKS API, AWS 명령줄 인터페이스, AWS SDKs, AWS CloudFormation 및 AWS Management Console을 사용하여 클러스터에 대한 액세스를 추가하고 관리할 수 있습니다. 즉, 클러스터를 생성할 때 사용한 것과 동일한 도구로 사용자를 관리할 수 있습니다.
- 이 예제에서 설명한 대로 자동화를 활용하여 CAM API를 진입점으로 사용하는 AWS IAM Identity Center를 IdP로 사용하는 클러스터를 배포합니다.
- 세분화된 Kubernetes 권한은 액세스 항목 및 액세스 정책을 통해 Kubernetes 사용자 또는 그룹을 SSO 자격 증명과 연결된 IAM 보안 주체와 매핑하여 적용할 수 있습니다.
- 시작하려면 인증 모드를 액세스 항목을 사용하도록 변경에 따라 기존 aws-auth ConfigMap 항목을 액세스 항목으로 마이그레이션합니다.

EKS 클러스터 엔드포인트를 프라이빗으로 설정

기본적으로 EKS 클러스터를 프로비저닝할 때 API 클러스터 엔드포인트는 퍼블릭으로 설정됩니다. 즉, 인터넷에서 액세스할 수 있습니다. 인터넷에서 액세스할 수 있지만 모든 API 요청을 IAM에서 인증한 다음 Kubernetes RBAC에서 승인해야 하므로 엔드포인트는 여전히 안전한 것으로 간주됩니다. 즉, 기업 보안 정책에서 인터넷에서 API에 대한 액세스를 제한하도록 요구하거나 클러스터 VPC 외부로 트래픽을 라우팅하지 못하도록 하는 경우 다음을 수행할 수 있습니다.

- EKS 클러스터 엔드포인트를 프라이빗으로 구성합니다. 이 주제에 대한 자세한 내용은 클러스터 엔드포인트 액세스 수정을 참조하세요.
- 클러스터 엔드포인트를 퍼블릭으로 두고 클러스터 엔드포인트와 통신할 수 있는 CIDR 블록을 지정합니다. 블록은 클러스터 엔드포인트에 액세스할 수 있는 화이트리스트에 등록된 퍼블릭 IP 주소 집합입니다.
- 화이트리스트 CIDR 블록 세트를 사용하여 퍼블릭 액세스를 구성하고 프라이빗 엔드포인트 액세스를 활성화됨으로 설정합니다. 이렇게 하면 컨트롤 플레인이 프로비저닝될 때 클러스터 VPC에 프로비저닝되는 교차 계정 ENIs를 통해 kubelet(작업자)과 Kubernetes API 간의 모든 네트워크 트래픽을 강제로 적용하면서 특정 범위의 퍼블릭 IPs에서 퍼블릭 액세스를 허용할 수 있습니다.

인증에 서비스 계정 토큰을 사용하지 마십시오.

서비스 계정 토큰은 수명이 긴 정적 자격 증명입니다. 손상, 분실 또는 도난된 경우 공격자는 서비스 계정이 삭제될 때까지 해당 토큰과 연결된 모든 작업을 수행할 수 있습니다. 경우에 따라 CI/CD 파이프라인 애플리케이션과 같이 클러스터 외부에서 Kubernetes API를 사용해야 하는 애플리케이션에 예외를 부여해야 할 수 있습니다. 이러한 애플리케이션이 EC2 인스턴스와 같은 AWS 인프라에서 실행되는 경우 인스턴스 프로파일을 사용하고 이를 Kubernetes RBAC 역할에 매핑하는 것이 좋습니다.

AWS 리소스에 대한 최소 권한 액세스 사용

IAM 사용자는 Kubernetes API에 액세스하기 위해 AWS 리소스에 권한을 할당받을 필요가 없습니다. IAM 사용자에게 EKS 클러스터에 대한 액세스 권한을 부여해야 하는 경우 해당 사용자의 `aws-auth` ConfigMap에서 특정 Kubernetes RBAC 그룹에 매핑되는 항목을 생성합니다.

클러스터 생성자 보안 주체에서 클러스터 관리자 권한 제거

기본적으로 Amazon EKS 클러스터는 클러스터 생성자 보안 주체에 바인딩된 영구 `cluster-admin` 권한으로 생성됩니다. Cluster Access Manager API를 사용하면 `API_AND_CONFIG_MAP` 또는 `API` 인증 모드를 사용할 `false`때를 로 설정하는이 권한 없이 클러스터`--access-config bootstrapClusterCreatorAdminPermissions`를 생성할 수 있습니다. 클러스터 구성에 대한 원치 않는 변경을 방지하기 위한 모범 사례로 간주되는 액세스를 취소합니다. 이 액세스를 취소하는 프로세스는 동일한 프로세스를 따라 클러스터에 대한 다른 액세스를 취소합니다.

API는 액세스 정책에서 IAM 보안 주체의 연결만 해제할 수 있는 유연성을 제공합니다.이 경우 입니다`AmazonEKSClusterAdminPolicy`.

```
$ aws eks list-associated-access-policies \
    --cluster-name <CLUSTER_NAME> \
    --principal-arn <IAM_PRINCIPAL_ARN>

$ aws eks disassociate-access-policy --cluster-name <CLUSTER_NAME> \
    --principal-arn <IAM_PRINCIPAL_ARN. \
    --policy-arn arn:aws:eks::aws:cluster-access-policy/AmazonEKSClusterAdminPolicy
```

또는 `cluster-admin` 권한과 연결된 액세스 항목을 완전히 제거합니다.

```
$ aws eks list-access-entries --cluster-name <CLUSTER_NAME>

{
```

```
    "accessEntries": []
}

$ aws eks delete-access-entry --cluster-name <CLUSTER_NAME> \
  --principal-arn <IAM_PRINCIPAL_ARN>
```

클러스터에 액세스할 수 없는 인시던트, 긴급 또는 휴지통 시나리오 중에 필요한 경우이 액세스 권한을 다시 부여할 수 있습니다.

클러스터가 여전히 CONFIG_MAP 인증 방법으로 구성된 경우 모든 추가 사용자에게 aws-auth ConfigMap을 통해 클러스터에 대한 액세스 권한을 부여해야 하며, aws-auth ConfigMap이 구성된 후에는 클러스터를 생성한 엔터티에 할당된 역할을 삭제하고 인시던트, 긴급 또는 휴지통 시나리오가 발생하거나 aws-authConfigMap이 손상되어 클러스터에 액세스할 수 없는 경우에만 다시 생성할 수 있습니다. 이는 프로덕션 클러스터에서 특히 유용할 수 있습니다.

여러 사용자가 클러스터에 대해 동일한 액세스 권한이 필요한 경우 IAM 역할 사용

각 개별 IAM 사용자에 대한 항목을 생성하는 대신 해당 사용자가 IAM 역할을 수임하고 해당 역할을 Kubernetes RBAC 그룹에 매핑하도록 허용합니다. 특히 액세스 권한이 필요한 사용자 수가 증가함에 따라 유지 관리가 더 쉬워집니다.

> **Important**
>
> ConfigMap에 의해 매핑된 IAM aws-auth 엔터티를 사용하여 EKS 클러스터에 액세스할 때 설명된 사용자 이름은 Kubernetes 감사 로그의 사용자 필드에 기록됩니다. IAM 역할을 사용하는 경우 해당 역할을 수임하는 실제 사용자는 기록되지 않으며 감사할 수 없습니다.

여전히 aws-auth configMap을 인증 방법으로 사용하는 경우 IAM 역할에 K8s RBAC 권한을 할당할 때 사용자 이름에 \{{SessionName}}을 포함해야 합니다. 이렇게 하면 감사 로그에 세션 이름이 기록되므로 CloudTrail 로그와 함께 실제 사용자가이 역할을 수임하는 사람을 추적할 수 있습니다.

```
- rolearn: arn:aws:iam::XXXXXXXXXXXX:role/testRole
  username: testRole:{{SessionName}}
  groups:
    - system:masters
```

Kubernetes 1.20 이상에서는이 Kubernetes 감사 로그에 user.extra.sessionName.0 추가되었으므로이 변경이 더 이상 필요하지 않습니다.

RoleBindings 및 ClusterRoleBindings를 생성할 때 최소 권한 액세스 사용

AWS 리소스에 대한 액세스 권한 부여와 마찬가지로 RoleBindings 및 ClusterRoleBindings에는 특정 함수를 수행하는 데 필요한 권한 집합만 포함되어야 합니다. 반드시 필요한 경우가 아니면 역할 및 ClusterRoles["*"]에서를 사용하지 마세요. 할당할 권한이 확실하지 않은 경우 audit2rbac과 같은 도구를 사용하여 역할을 자동으로 생성하고 Kubernetes 감사 로그에서 관찰된 API 호출을 기반으로 바인딩하는 것이 좋습니다.

자동화된 프로세스를 사용하여 클러스터 생성

이전 단계에서 볼 수 있듯이 Amazon EKS 클러스터를 생성할 때 API_AND_CONFIG_MAP 또는 API 인증 모드를 사용하지 않고 클러스터 생성자에게 cluster-admin 권한을 위임하도록 옵트아웃하지 않으면 클러스터를 생성하는 페더레이션 사용자와 같은 IAM 엔터티 사용자 또는 역할에 클러스터의 RBAC 구성에 대한 system:masters 권한이 자동으로 부여됩니다. CONFIG_MAP 인증 방법을 사용하고 aws-authConfigMap을 사용하는 경우 여기에 설명된 대로이 권한을 제거하는 것이 모범 사례이지만이 액세스는 취소할 수 없습니다. 따라서 다른 사용자 또는 엔터티가 수임할 권한 없이 전용 IAM 역할에 연결된 인프라 자동화 파이프라인을 사용하여 클러스터를 생성하고 파이프라인을 트리거할 수 있는 액세스 권한이 있는이 역할의 권한, 정책을 정기적으로 감사하는 것이 좋습니다. 또한이 역할은 클러스터에서 일상적인 작업을 수행하는 데 사용해서는 안 되며, 예를 들어 SCM 코드 변경을 통해 파이프라인에 의해 트리거된 클러스터 수준 작업에만 사용해야 합니다.

전용 IAM 역할을 사용하여 클러스터 생성

Amazon EKS 클러스터를 생성하면 클러스터를 생성하는 페더레이션 사용자와 같은 IAM 엔터티 사용자 또는 역할에 클러스터의 RBAC 구성에 대한 system:masters 권한이 자동으로 부여됩니다. 이 액세스는 제거할 수 없으며 aws-auth ConfigMap을 통해 관리되지 않습니다. 따라서 전용 IAM 역할로 클러스터를 생성하고이 역할을 수임할 수 있는 사람을 정기적으로 감사하는 것이 좋습니다. 이 역할은 클러스터에서 일상적인 작업을 수행하는 데 사용해서는 안 되며, 대신 이러한 목적으로 aws-auth ConfigMap을 통해 클러스터에 대한 액세스 권한을 추가 사용자에게 부여해야 합니다. aws-auth ConfigMap을 구성한 후에는 역할을 보호해야 하며 클러스터에 액세스할 수 없는 시나리오의 경우 임시 승격된 권한 모드/중단 창에서만 사용해야 합니다. 이는 직접 사용자 액세스가 구성되지 않은 클러스터에서 특히 유용할 수 있습니다.

클러스터에 대한 액세스를 정기적으로 감사

액세스가 필요한 사람은 시간이 지남에 따라 변경될 수 있습니다. aws-auth ConfigMap을 정기적으로 감사하여 액세스 권한이 부여된 사람과 할당된 권한을 확인할 계획입니다. kubectl-who-can 또는 rbac-lookup과 같은 오픈 소스 도구를 사용하여 특정 서비스 계정, 사용자 또는 그룹에 바인딩된 역할을 검

사할 수도 있습니다. 감사에 대한 섹션으로 이동하면이 주제를 자세히 살펴보겠습니다. 추가 아이디어는 NCC 그룹의이 문서에서 확인할 수 있습니다.

`aws-auth` configMap에 의존하는 경우 도구를 사용하여 변경합니다.

형식이 잘못된 aws-auth ConfigMap으로 인해 클러스터에 대한 액세스 권한이 손실될 수 있습니다. ConfigMap을 변경해야 하는 경우 도구를 사용합니다.

eksctl `eksctl` CLI에는 aws-auth ConfigMap에 자격 증명 매핑을 추가하는 명령이 포함되어 있습니다.

CLI 도움말 보기:

```
$ eksctl create iamidentitymapping --help
...
```

Amazon EKS 클러스터에 매핑된 자격 증명을 확인합니다.

```
$ eksctl get iamidentitymapping --cluster $CLUSTER_NAME --region $AWS_REGION
ARN                                                                     USERNAME
               GROUPS                                                    ACCOUNT
arn:aws:iam::788355785855:role/kube-system-<SELF_GENERATED_UUID>      system:node:
{{SessionName}}     system:bootstrappers,system:nodes,system:node-proxier
```

IAM 역할을 클러스터 관리자로 설정:

```
$ eksctl create iamidentitymapping --cluster  <CLUSTER_NAME> --region=<region> --arn
 arn:aws:iam::123456:role/testing --group system:masters --username admin
...
```

자세한 내용은 `eksctl` 문서를 검토하세요.

keikoproj의 aws-auth

`aws-auth` by keikoproj에는 cli와 go 라이브러리가 모두 포함되어 있습니다.

도움말 다운로드 및 보기 CLI 도움말:

```
$ go get github.com/keikoproj/aws-auth
...
$ aws-auth help
...
```

또는 kubectl용 krew 플러그인 관리자를 aws-auth 사용하여를 설치합니다.

```
$ kubectl krew install aws-auth
...
$ kubectl aws-auth
...
```

go 라이브러리를 포함한 자세한 내용은 GitHub의 aws-auth 문서를 검토하세요.

AWS IAM Authenticator CLI

aws-iam-authenticator 프로젝트에는 ConfigMap을 업데이트하기 위한 CLI가 포함되어 있습니다.

GitHub에서 릴리스를 다운로드합니다.

IAM 역할에 클러스터 권한 추가:

```
$ ./aws-iam-authenticator add role --rolearn arn:aws:iam::185309785115:role/lil-dev-
role-cluster --username lil-dev-user --groups system:masters --kubeconfig ~/.kube/
config
...
```

인증 및 액세스 관리에 대한 대체 접근 방식

IAM은 EKS 클러스터에 액세스해야 하는 사용자를 인증하는 기본 방법이지만 인증 프록시 및 Kubernetes 위장을 사용하여 GitHub와 같은 OIDC 자격 증명 공급자를 사용할 수 있습니다. https://kubernetes.io/docs/reference/access-authn-authz/authentication/#user-impersonation 이러한 두 솔루션의 게시물은 AWS 오픈 소스 블로그에 게시되었습니다.

- 텔레포트와 함께 GitHub 자격 증명을 사용하여 EKS에 인증
- kube-oidc-proxy를 사용하여 여러 EKS 클러스터에서 일관된 OIDC 인증

> **Important**
>
> EKS는 기본적으로 프록시를 사용하지 않고 OIDC 인증을 지원합니다. 자세한 내용은 시작 블로그인 Amazon EKS에 대한 OIDC 자격 증명 공급자 인증 소개를 참조하세요. 다양한 인증 방법에 대한 커넥터가 있는 인기 있는 오픈 소스 OIDC 공급자인 Dex로 EKS를 구성하는 방법을 보여주는 예제는 Dex & dex-k8s-authenticator를 사용하여 Amazon EKS에 인증하기를 참조하

세요. 블로그에 설명된 대로 OIDC 공급자가 인증한 사용자 이름/사용자 그룹이 Kubernetes 감사 로그에 표시됩니다.

AWS SSO를 사용하여 Azure AD와 같은 외부 자격 증명 공급자와 AWS를 페더레이션할 수도 있습니다. 이를 사용하기로 결정한 경우 AWS CLI v2.0에는 SSO 세션을 현재 CLI 세션과 쉽게 연결하고 IAM 역할을 수임할 수 있는 명명된 프로파일을 생성하는 옵션이 포함되어 있습니다. IAM 역할이 사용자의 Kubernetes RBAC 그룹을 결정하는 데 사용`kubectl`되므로를 실행하기 전에 역할을 수임해야 합니다.

EKS 포드의 자격 증명 및 자격 증명

Kubernetes 클러스터 내에서 실행되는 특정 애플리케이션은 Kubernetes API를 호출하여 제대로 작동할 수 있는 권한이 필요합니다. 예를 들어 AWS Load Balancer Controller는 서비스의 엔드포인트를 나열할 수 있어야 합니다. 또한 컨트롤러는 AWS APIs를 호출하여 ALB를 프로비저닝하고 구성할 수 있어야 합니다. 이 섹션에서는 포드에 권한과 권한을 할당하는 모범 사례를 살펴봅니다.

Kubernetes 서비스 계정

서비스 계정은 포드에 Kubernetes RBAC 역할을 할당할 수 있는 특수한 유형의 객체입니다. 클러스터 내의 각 네임스페이스에 대해 기본 서비스 계정이 자동으로 생성됩니다. 특정 서비스 계정을 참조하지 않고 포드를 네임스페이스에 배포하면 해당 네임스페이스의 기본 서비스 계정이 포드에 자동으로 할당되고 보안 암호, 즉 해당 서비스 계정의 서비스 계정(JWT) 토큰이 포드에의 볼륨으로 마운트됩니다/`var/run/secrets/kubernetes.io/serviceaccount`. 해당 디렉터리에서 서비스 계정 토큰을 디코딩하면 다음 메타데이터가 표시됩니다.

```
{
  "iss": "kubernetes/serviceaccount",
  "kubernetes.io/serviceaccount/namespace": "default",
  "kubernetes.io/serviceaccount/secret.name": "default-token-5pv4z",
  "kubernetes.io/serviceaccount/service-account.name": "default",
  "kubernetes.io/serviceaccount/service-account.uid":
 "3b36ddb5-438c-11ea-9438-063a49b60fba",
  "sub": "system:serviceaccount:default:default"
}
```

기본 서비스 계정에는 Kubernetes API에 대한 다음과 같은 권한이 있습니다.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
```

```
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: "2020-01-30T18:13:25Z"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:discovery
  resourceVersion: "43"
  selfLink: /apis/rbac.authorization.k8s.io/v1/clusterroles/system%3Adiscovery
  uid: 350d2ab8-438c-11ea-9438-063a49b60fba
rules:
- nonResourceURLs:
  - /api
  - /api/*
  - /apis
  - /apis/*
  - /healthz
  - /openapi
  - /openapi/*
  - /version
  - /version/
  verbs:
  - get
```

이 역할은 인증되지 않고 인증된 사용자에게 API 정보를 읽을 수 있는 권한을 부여하며 공개적으로 액세스할 수 있는 안전한 것으로 간주됩니다.

포드 내에서 실행되는 애플리케이션이 Kubernetes APIs를 호출할 때 해당 API를 호출할 수 있는 권한을 명시적으로 부여하는 서비스 계정을 포드에 할당해야 APIs. 사용자 액세스 지침과 마찬가지로 서비스 계정에 바인딩된 역할 또는 ClusterRole은 애플리케이션이 작동하는 데 필요한 API 리소스 및 메서드로 제한되어야 합니다. 기본이 아닌 서비스 계정을 사용하려면 포드의 `spec.serviceAccountName` 필드를 사용하려는 서비스 계정의 이름으로 설정하면 됩니다. 서비스 계정 생성에 대한 자세한 내용은 https://kubernetes.io/docs/reference/access-authn-authz/rbac/#service-account-permissions 참조하십시오.

> **Note**
>
> Kubernetes 1.24 이전에는 Kubernetes가 각 서비스 계정에 대한 보안 암호를 자동으로 생성합니다. 이 보안 암호는 /var/run/secrets/kubernetes.io/serviceaccount 포드에 탑재되었으며 포드에서 Kubernetes API 서버에 인증하는 데 사용됩니다. Kubernetes 1.24에서는 포드가 실행될 때 서비스 계정 토큰이 동적으로 생성되며 기본적으로 1시간 동안만 유효합니

다. 서비스 계정의 보안 암호는 생성되지 않습니다. Jenkins와 같이 Kubernetes API에 인증해야 하는 클러스터 외부에서 실행되는 애플리케이션이 있는 경우와 같은 서비스 계정을 참조하는 주석과 `kubernetes.io/service-account-token` 함께 유형의 보안 암호를 생성해야 합니다`metadata.annotations.kubernetes.io/service-account.name: <SERVICE_ACCOUNT_NAME>`. 이러한 방식으로 생성된 보안 암호는 만료되지 않습니다.

서비스 계정에 대한 IAM 역할(IRSA)

IRSA는 Kubernetes 서비스 계정에 IAM 역할을 할당할 수 있는 기능입니다. 서비스 계정 토큰 볼륨 프로젝션이라는 Kubernetes 기능을 활용하여 작동합니다. IAM 역할을 참조하는 서비스 계정으로 포드를 구성하면 Kubernetes API 서버가 시작 시 클러스터에 대한 퍼블릭 OIDC 검색 엔드포인트를 호출합니다. 엔드포인트는 Kubernetes에서 발급한 OIDC 토큰과 볼륨으로 마운트된 결과 토큰에 암호화 방식으로 서명합니다. 이 서명된 토큰을 사용하면 포드가 AWS APIs 관련 IAM 역할을 호출할 수 있습니다. AWS API가 호출되면 AWS SDKs 호출합니다`sts:AssumeRoleWithWebIdentity`. 토큰의 서명을 검증한 후 IAM은 Kubernetes에서 발급한 토큰을 임시 AWS 역할 자격 증명으로 교환합니다.

IRSA를 사용할 때는 AWS SDK 세션을 재사용하여 AWS STS에 대한 불필요한 호출을 방지하는 것이 중요합니다.

IRSA용 (JWT) 토큰을 디코딩하면 아래 표시된 예제와 유사한 출력이 생성됩니다.

```
{
  "aud": [
    "sts.amazonaws.com"
  ],
  "exp": 1582306514,
  "iat": 1582220114,
  "iss": "https://oidc.eks.us-west-2.amazonaws.com/id/
D43CF17C27A865933144EA99A26FB128",
  "kubernetes.io": {
    "namespace": "default",
    "pod": {
      "name": "alpine-57b5664646-rf966",
      "uid": "5a20f883-5407-11ea-a85c-0e62b7a4a436"
    },
    "serviceaccount": {
      "name": "s3-read-only",
      "uid": "a720ba5c-5406-11ea-9438-063a49b60fba"
    }
  },
```

```
  "nbf": 1582220114,
  "sub": "system:serviceaccount:default:s3-read-only"
}
```

이 특정 토큰은 IAM 역할을 수임하여 포드 보기 전용 권한을 S3에 부여합니다. 애플리케이션이 S3에서 읽기를 시도하면 토큰이 다음과 유사한 임시 IAM 자격 증명 세트로 교환됩니다.

```
{
    "AssumedRoleUser": {
        "AssumedRoleId": "AROA36C6WWEJULFUYMPB6:abc",
        "Arn": "arn:aws:sts::123456789012:assumed-role/eksctl-winterfell-addon-
iamserviceaccount-de-Role1-1D61LT75JH3MB/abc"
    },
    "Audience": "sts.amazonaws.com",
    "Provider": "arn:aws:iam::123456789012:oidc-provider/oidc.eks.us-
west-2.amazonaws.com/id/D43CF17C27A865933144EA99A26FB128",
    "SubjectFromWebIdentityToken": "system:serviceaccount:default:s3-read-only",
    "Credentials": {
        "SecretAccessKey": "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY",
        "SessionToken": "FwoGZXIvYXdzEGMaDMLxAZkuLpmSwYXShiL9A1S0X87VBC1mHCrRe/
pB2oesl1eXxUYnPJyC9ayOoXMvqXQsomq0xs6OqZ3vaa5Iw1HIyA4Cv1suLaOCoU3hNvOIJ6C94H1vU0siQYk7DIq9Av5RZ
Zv2A6zp5xGz9cWj2f0aD9v66vX4bexOs5t/YYhwuwAvkkJPSIGvxja0xRThnceHyFHKtj0Hbi/
PWAtlI8YJcDX69cM30JAHDdQHltm/4scFptW1hlvMaPWReCAaCrsHrATyka7ttw5YlUyvZ8EPogj6fwHlxmrXM9h1Bqdiko
LgkWsCTGlYcY8z3zkigJMbYn07ewTL5Ss7LazTJJa758I7PZan/
v3xQHd5DEc5WBneiV3iOznDFgup0VAMkIviVjVCkszaPSVEdK2NU7jtrh6Jfm7bU/3P6ZGCkyDLIa8MBn9KPXeJd/
yjTk5IifIwO/mDpGNUribg6TPxhzZ8b/XdZO1kS1gVgqjXyVCM+BRBh6C4H21w/eMzjCtDIpoxt5rGKL6Nu/
IFMipoC4fgx6LIIHwtGYMG7SWQi7OsMAkiwZRg0n68/RqWgLzBt/4pfjSRYuk=",
        "Expiration": "2020-02-20T18:49:50Z",
        "AccessKeyId": "ASIAIOSFODNN7EXAMPLE"
    }
}
```

EKS 컨트롤 플레인의 일부로 실행되는 변형 웹후크는 AWS 역할 ARN과 웹 자격 증명 토큰 파일의 경로를 환경 변수로 포드에 주입합니다. 이러한 값은 수동으로 제공할 수도 있습니다.

```
AWS_ROLE_ARN=arn:aws:iam::AWS_ACCOUNT_ID:role/IAM_ROLE_NAME
AWS_WEB_IDENTITY_TOKEN_FILE=/var/run/secrets/eks.amazonaws.com/serviceaccount/token
```

kubelet은 총 TTL의 80%보다 오래된 경우 또는 24시간 후에 프로젝션된 토큰을 자동으로 교체합니다. AWS SDKs는 교체 시 토큰을 다시 로드할 책임이 있습니다. IRSA에 대한 자세한 내용은 https://docs.aws.amazon.com/eks/latest/userguide/iam-roles-for-service-accounts-technical-overview.html 참조하십시오.

EKS Pod Identity

EKS Pod Identity는 re:Invent 2023에서 시작된 기능으로, AWS 계정의 각 클러스터에 대해 Open Id Connect(OIDC) ID 제공업체(IDP)를 구성할 필요 없이 kubernetes 서비스 계정에 IAM 역할을 할당할 수 있습니다. EKS Pod Identity를 사용하려면 모든 적격 작업자 노드에서 DaemonSet 포드로 실행되는 에이전트를 배포해야 합니다. 이 에이전트는 EKS 추가 기능으로 사용할 수 있으며 EKS Pod Identity 기능을 사용하기 위한 사전 조건입니다. 이 기능을 사용하려면 애플리케이션에서 지원되는 버전의 AWS SDK를 사용해야 합니다.

포드에 대해 EKS 포드 ID가 구성되면 EKS는에서 포드 ID 토큰을 탑재하고 새로 고칩니다`/var/run/secrets/pods.eks.amazonaws.com/serviceaccount/eks-pod-identity-token`. 이 토큰은 AWS SDK에서 EKS Pod Identity Agent와 통신하는 데 사용됩니다.이 에이전트는 포드 자격 증명 토큰과 에이전트의 IAM 역할을 사용하여 AssumeRoleForPodIdentity API를 호출하여 포드에 대한 임시 자격 증명을 생성합니다. 포드에 전달되는 포드 자격 증명 토큰은 EKS 클러스터에서 발급되고 암호화 방식으로 서명된 JWT이며 EKS 포드 자격 증명과 함께 사용할 수 있는 적절한 JWT 클레임을 포함합니다.

EKS Pod Identity에 대한 자세한 내용은 이 블로그를 참조하세요.

EKS Pod Identity를 사용하기 위해 애플리케이션 코드를 수정할 필요는 없습니다. 지원되는 AWS SDK 버전은 자격 증명 공급자 체인을 사용하여 EKS Pod Identity에서 사용할 수 있는 자격 증명을 자동으로 검색합니다. IRSA와 마찬가지로 EKS 포드 자격 증명은 포드 내에서 변수를 설정하여 AWS 자격 증명을 찾는 방법을 지시합니다.

EKS Pod Identity에 대한 IAM 역할 작업

- EKS Pod Identity는 EKS 클러스터와 동일한 AWS 계정에 속한 IAM 역할만 직접 수임할 수 있습니다. 다른 AWS 계정의 IAM 역할에 액세스하려면 SDK 구성 또는 애플리케이션 코드에서 프로필을 구성하여 해당 역할을 수임해야 합니다. https://docs.aws.amazon.com/IAM/latest/UserGuide/sts_example_sts_AssumeRole_section.html
- 서비스 계정에 대해 EKS Pod Identity를 구성하는 경우 Pod Identity Association을 구성하는 사람 또는 프로세스에 해당 역할에 대한 `iam:PassRole` 권한이 있어야 합니다.
- 각 서비스 계정에는 EKS Pod Identity를 통해 연결된 IAM 역할이 하나만 있을 수 있지만 동일한 IAM 역할을 여러 서비스 계정과 연결할 수 있습니다.
- EKS Pod Identity와 함께 사용되는 IAM 역할은 `pods.eks.amazonaws.com` 서비스 보안 주체가 이를 수임하고 세션 태그를 설정할 수 있도록 허용해야 합니다. 다음은 EKS Pod Identity가 IAM 역할을 사용하도록 허용하는 역할 신뢰 정책의 예입니다.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "pods.eks.amazonaws.com"
      },
      "Action": [
        "sts:AssumeRole",
        "sts:TagSession"
      ],
      "Condition": {
        "StringEquals": {
          "aws:SourceOrgId": "${aws:ResourceOrgId}"
        }
      }
    }
  ]
}
```

AWS는 교차 서비스 혼동된 대리자 문제를 방지하기 `aws:SourceOrgId` 위해와 같은 조건 키를 사용할 것을 권장합니다. 위 예제 역할 신뢰 정책에서 `ResourceOrgId`는 AWS 계정이 속한 AWS Organization의 AWS Organizations Organization ID와 동일한 변수입니다. EKS는 EKS Pod Identity로 역할을 수임할 때와 `aws:SourceOrgId` 동일한 값을 전달합니다.

ABAC 및 EKS 포드 ID

EKS Pod Identity는 IAM 역할을 수임할 때 다음 세션 태그를 설정합니다.

EKS Pod Identity 세션 태그	값
kubernetes-네임스페이스	EKS Pod Identity와 연결된 포드가 실행되는 네임스페이스입니다.
kubernetes-service-account	EKS Pod Identity와 연결된 kubernetes 서비스 계정의 이름입니다.
eks-cluster-arn	EKS 클러스터의 ARN, 예: . `arn:${Partition}:eks:${Region}:${Account}:cluster/${ClusterName}` 클

EKS Pod Identity 세션 태그	값
	러스터 ARN은 고유하지만 동일한 AWS 계정 내에서 동일한 이름의 동일한 리전에서 클러스터가 삭제되고 다시 생성되면 동일한 ARN을 갖게 됩니다.
eks-cluster-name	EKS 클러스터의 이름입니다. EKS 클러스터 이름은 AWS 계정과 다른 AWS 계정의 EKS 클러스터 내에서 동일할 수 있습니다.
kubernetes-pod-name	EKS의 포드 이름입니다.
kubernetes-pod-uid	EKS에 있는 포드의 UID입니다.

이러한 세션 태그를 사용하면 속성 기반 액세스 제어(ABAC)를 사용하여 AWS 리소스에 대한 액세스 권한을 특정 kubernetes 서비스 계정에만 부여할 수 있습니다. 이때 kubernetes 서비스 계정은 네임스페이스 내에서만 고유하고 kubernetes 네임스페이스는 EKS 클러스터 내에서만 고유하다는 점을 이해하는 것이 매우 중요합니다. 이러한 세션 태그는 다음과 같은 `aws:PrincipalTag/<tag-key>` 전역 조건 키를 사용하여 AWS 정책에서 액세스할 수 있습니다. `aws:PrincipalTag/eks-cluster-arn`

예를 들어 IAM 또는 리소스 정책을 사용하여 계정의 AWS 리소스에 액세스할 수 있도록 특정 서비스 계정에만 액세스 권한을 부여하려면 다른 클러스터`eks-cluster-arn`가 및를 가질 수 있으므로 의도한 클러스터의 해당 서비스 계정만 해당 리소스에 액세스할 수 `kubernetes-service-account` 있도록 `kubernetes-service-accounts` 및 `kubernetes-namespace` 태그를 확인해야 합니다`kubernetes-namespaces`.

이 예제 S3 버킷 정책은 , `kubernetes-service-account`가 `kubernetes-namespace` `eks-cluster-arn` 모두 예상 값을 충족하는 경우에만 연결된 S3 버킷의 객체에 대한 액세스 권한을 부여합니다. 여기서 EKS 클러스터는 AWS 계정에서 호스팅됩니다`111122223333`.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:root"
            },
```

```
            "Action": "s3:*",
            "Resource":            [
                "arn:aws:s3:::ExampleBucket/*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:PrincipalTag/kubernetes-service-account": "s3objectservice",
                    "aws:PrincipalTag/eks-cluster-arn": "arn:aws:eks:us-
west-2:111122223333:cluster/ProductionCluster",
                    "aws:PrincipalTag/kubernetes-namespace": "s3datanamespace"
                }
            }
        }
    ]
}
```

EKS Pod Identity와 IRSA 비교

EKS Pod Identity와 IRSA 모두 EKS 포드에 임시 AWS 자격 증명을 제공하는 데 선호되는 방법입니다. IRSA에 대한 특정 사용 사례가 없는 한 EKS를 사용할 때는 EKS Pod Identity를 사용하는 것이 좋습니다. 이 표는 두 기능을 비교하는 데 도움이 됩니다.

#	EKS Pod Identity	IRSA
AWS 계정에서 OIDC IDP를 생성할 수 있는 권한이 필요합니까?	아니요	예
클러스터당 고유한 IDP 설정 필요	아니요	예
ABAC와 함께 사용할 관련 세션 태그를 설정합니다.	예	아니요
iam:PassRole 검사가 필요합니까?	예	아니요
AWS 계정에서 AWS STS 할당량을 사용하나요?	아니요	예

#	EKS Pod Identity	IRSA
다른 AWS 계정에 액세스할 수 있음	역할 체인을 사용하여 간접적으로	sts:AssumeRoleWithWebIdentity를 사용하여 직접
AWS SDKs와 호환	예	예
노드에 Pod Identity Agent Daemonset이 필요합니까?	예	아니요

EKS 포드에 대한 자격 증명 및 자격 증명 권장 사항

IRSA를 사용하도록 aws-node 데몬 세트 업데이트

현재 aws-node 데몬 세트는 EC2 인스턴스에 할당된 역할을 사용하여 포드에 IPs 할당하도록 구성됩니다. 이 역할에는 AmazonEKS_CNI_Policy 및 EC2ContainerRegistryReadOnly와 같은 여러 AWS 관리형 정책이 포함되어 있어 노드에서 실행되는 모든 포드가 ENIs를 연결/분리하거나, IP 주소를 할당/할당 해제하거나, ECR에서 이미지를 가져올 수 있습니다. 이는 클러스터에 위험을 초래하므로 IRSA를 사용하도록 aws-node 데몬 세트를 업데이트하는 것이 좋습니다. 이를 위한 스크립트는이 가이드의 리포지토리에서 찾을 수 있습니다.

aws-node 데몬 세트는 버전 v1.15.5 이상에서 EKS Pod Identity를 지원합니다.

작업자 노드에 할당된 인스턴스 프로파일에 대한 액세스 제한

IRSA 또는 EKS Pod Identity를 사용하는 경우 먼저 IRSA 또는 EKS Pod Identity를 사용하도록 포드의 자격 증명 체인을 업데이트하지만 포드는 여전히 작업자 노드에 할당된 인스턴스 프로파일의 권한을 상속할 수 있습니다. 이러한 권한이 필요하지 않은 포드의 경우 인스턴스 메타데이터에 대한 액세스를 차단하여 애플리케이션이 노드가 아닌 필요한 권한만 갖도록 할 수 있습니다.

> **Warning**
>
> 인스턴스 메타데이터에 대한 액세스를 차단하면 IRSA 또는 EKS Pod Identity를 사용하지 않는 포드가 작업자 노드에 할당된 역할을 상속하지 못합니다.

아래 예제와 같이 인스턴스가 IMDSv2만 사용하도록 하고 홉 수를 1로 업데이트하여 인스턴스 메타데이터에 대한 액세스를 차단할 수 있습니다. 노드 그룹의 시작 템플릿에 이러한 설정을 포함할 수도 있

습니다. 인스턴스 메타데이터를 비활성화하면 노드 종료 핸들러 및 인스턴스 메타데이터에 의존하는 기타 사물과 같은 구성 요소가 제대로 작동하지 않습니다.

```
$ aws ec2 modify-instance-metadata-options --instance-id <value> --http-tokens required
 --http-put-response-hop-limit 1
...
```

Terraform을 사용하여 관리형 노드 그룹에 사용할 시작 템플릿을 생성하는 경우 메타데이터 블록을 추가하여이 코드 조각에 표시된 대로 홉 수를 구성합니다.

```
tf hl_lines="7" resource "aws_launch_template" "foo" { name = "foo" …
 metadata_options { http_endpoint = "enabled" http_tokens = "required"
http_put_response_hop_limit = 1 instance_metadata_tags = "enabled" } …
```

노드에서 iptable을 조작하여 EC2 메타데이터에 대한 포드의 액세스를 차단할 수도 있습니다. 이 방법에 대한 자세한 내용은 인스턴스 메타데이터 서비스에 대한 액세스 제한을 참조하세요.

IRSA 또는 EKS Pod Identity를 지원하지 않는 이전 버전의 AWS SDK를 사용하는 애플리케이션이 있는 경우 SDK 버전을 업데이트해야 합니다.

IRSA 역할에 대한 IAM 역할 신뢰 정책의 범위를 서비스 계정 이름, 네임스페이스 및 클러스터로 지정합니다.

신뢰 정책의 범위는 네임스페이스 또는 네임스페이스 내의 특정 서비스 계정으로 지정할 수 있습니다. IRSA를 사용할 때는 서비스 계정 이름을 포함하여 역할 신뢰 정책을 최대한 명시적으로 만드는 것이 가장 좋습니다. 이렇게 하면 동일한 네임스페이스 내의 다른 포드가 역할을 수임하는 것을 효과적으로 방지할 수 있습니다. CLI는 이를 사용하여 서비스 계정/IAM 역할을 생성할 때 이를 자동으로 `eksctl` 수행합니다. 자세한 내용은 https://eksctl.io/usage/iamserviceaccounts/ 참조하십시오.

IAM으로 직접 작업할 때 조건을 사용하는 역할의 신뢰 정책에 조건을 추가하여 `:sub` 클레임이 예상한 네임스페이스 및 서비스 계정인지 확인합니다. 예를 들어, 이전에는 하위 클레임이 "system:serviceaccount:default:s3-read-only"인 IRSA 토큰이 있었습니다. 이는 `default` 네임스페이스이고 서비스 계정은 입니다`s3-read-only`. 다음과 같은 조건을 사용하여 클러스터의 지정된 네임스페이스에 있는 서비스 계정만 해당 역할을 수임할 수 있도록 합니다.

```
  "Condition": {
      "StringEquals": {
          "oidc.eks.us-west-2.amazonaws.com/id/D43CF17C27A865933144EA99A26FB128:aud":
 "sts.amazonaws.com",
```

```
          "oidc.eks.us-west-2.amazonaws.com/id/D43CF17C27A865933144EA99A26FB128:sub":
"system:serviceaccount:default:s3-read-only"
      }
 }
```

애플리케이션당 하나의 IAM 역할 사용

IRSA와 EKS Pod Identity를 모두 사용하면 각 애플리케이션에 고유한 IAM 역할을 부여하는 것이 가장 좋습니다. 이렇게 하면 다른 애플리케이션에 영향을 주지 않고 한 애플리케이션을 수정할 수 있으므로 격리가 개선되고 애플리케이션에 필요한 권한만 부여하여 최소 권한의 보안 주체를 적용할 수 있습니다.

EKS Pod Identity와 함께 ABAC를 사용하는 경우 여러 서비스 계정에서 공통 IAM 역할을 사용하고 액세스 제어를 위해 세션 속성에 의존할 수 있습니다. 이는 대규모로 운영할 때 특히 유용합니다. ABAC를 사용하면 더 적은 IAM 역할로 운영할 수 있기 때문입니다.

애플리케이션에서 IMDS에 액세스해야 하는 경우 IMDSv2를 사용하고 EC2 인스턴스의 홉 제한을 2로 늘립니다.

IMDSv2에서는 PUT 요청을 사용하여 세션 토큰을 가져와야 합니다. 초기 PUT 요청에는 세션 토큰에 대한 TTL이 포함되어야 합니다. 최신 버전의 AWS SDKs 이러한 토큰과 해당 토큰의 갱신을 자동으로 처리합니다. 또한 IP 전달을 방지하기 위해 EC2 인스턴스의 기본 홉 제한이 의도적으로 1로 설정되어 있다는 점에 유의해야 합니다. 결과적으로 EC2 인스턴스에서 실행되는 세션 토큰을 요청하는 포드는 결국 시간이 초과되어 IMDSv1 데이터 흐름 사용으로 대체될 수 있습니다. EKS는 v1과 v2를 모두 활성화하고 eksctl 또는 공식 CloudFormation 템플릿을 사용하여 프로비저닝된 노드에서 홉 제한을 2로 변경하여 IMDSv2 지원을 추가합니다. CloudFormation

서비스 계정 토큰의 자동 탑재 비활성화

애플리케이션이 Kubernetes API를 호출할 필요가 없는 경우 `automountServiceAccountToken` 속성을 애플리케이션의 PodSpec`false`에서 로 설정하거나 포드에 더 이상 자동으로 마운트되지 않도록 각 네임스페이스의 기본 서비스 계정을 패치합니다. 예제:

```
kubectl patch serviceaccount default -p $'automountServiceAccountToken: false'
```

각 애플리케이션에 전용 서비스 계정 사용

각 애플리케이션에는 자체 전용 서비스 계정이 있어야 합니다. 이는 Kubernetes API의 서비스 계정과 IRSA 및 EKS Pod Identity에 적용됩니다.

> **Important**
>
> IRSA를 사용할 때 현재 위치 클러스터 업그레이드를 수행하는 대신 클러스터 업그레이드에 블루/그린 접근 방식을 사용하는 경우 각 IRSA IAM 역할의 신뢰 정책을 새 클러스터의 OIDC 엔드포인트로 업데이트해야 합니다. 블루/그린 클러스터 업그레이드에서는 이전 클러스터와 함께 최신 버전의 Kubernetes를 실행하는 클러스터를 생성하고 로드 밸런서 또는 서비스 메시를 사용하여 이전 클러스터에서 실행되는 서비스에서 새 클러스터로 트래픽을 원활하게 이동할 수 있습니다. EKS Pod Identity와 함께 블루/그린 클러스터 업그레이드를 사용하는 경우 새 클러스터의 IAM 역할과 서비스 계정 간에 포드 ID 연결을 생성합니다. 그리고 `sourceArn` 조건이 있는 경우 IAM 역할 신뢰 정책을 업데이트합니다.

루트가 아닌 사용자로 애플리케이션 실행

컨테이너는 기본적으로 루트로 실행됩니다. 이렇게 하면 웹 자격 증명 토큰 파일을 읽을 수 있지만 컨테이너를 루트로 실행하는 것은 모범 사례로 간주되지 않습니다. 또는 PodSpec에 `spec.securityContext.runAsUser` 속성을 추가하는 것이 좋습니다. 의 값은 임의 값`runAsUser`입니다.

다음 예제에서는 포드 내의 모든 프로세스가 `runAsUser` 필드에 지정된 사용자 ID로 실행됩니다.

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
  containers:
  - name: sec-ctx-demo
    image: busybox
    command: [ "sh", "-c", "sleep 1h" ]
```

루트가 아닌 사용자로 컨테이너를 실행하면 토큰에 기본적으로 0600 [루트] 권한이 할당되므로 컨테이너가 IRSA 서비스 계정 토큰을 읽지 못합니다. fsgroup=65534 [Nobody]를 포함하도록 컨테이너의 securityContext를 업데이트하면 컨테이너가 토큰을 읽을 수 있습니다.

```
spec:
  securityContext:
```

```
    fsGroup: 65534
```

Kubernetes 1.19 이상에서는이 변경이 더 이상 필요하지 않으며 애플리케이션은 아무도 없음 그룹에 추가하지 않고 IRSA 서비스 계정 토큰을 읽을 수 있습니다.

애플리케이션에 대한 최소 권한 액세스 권한 부여

Action Hero는 애플리케이션과 함께 실행하여 애플리케이션이 제대로 작동하는 데 필요한 AWS API 호출 및 해당 IAM 권한을 식별할 수 있는 유틸리티입니다. 애플리케이션에 할당된 IAM 역할의 범위를 점진적으로 제한하는 데 도움이 된다는 점에서 IAM Access Advisor와 유사합니다. 자세한 내용은 AWS 리소스에 대한 최소 권한 액세스 권한 부여에 대한 설명서를 참조하세요.

IRSA 및 포드 ID와 함께 사용되는 IAM 역할에 권한 경계를 설정하는 것이 좋습니다. 권한 경계를 사용하여 IRSA 또는 포드 자격 증명에서 사용하는 역할이 최대 권한 수준을 초과하지 않도록 할 수 있습니다. 권한 경계 정책 예제를 사용하여 권한 경계를 시작하는 방법에 대한 예제 가이드는이 github 리포지토리를 참조하세요.

EKS 클러스터에 대한 불필요한 익명 액세스 검토 및 취소

모든 API 작업에 대해 이상적으로는 익명 액세스를 비활성화해야 합니다. 익명 액세스는 Kubernetes 기본 제공 사용자 시스템:익명에 대한 RoleBinding 또는 ClusterRoleBinding을 생성하여 부여됩니다. rbac-lookup 도구를 사용하여 system:anonymous user가 클러스터에 대해 갖는 권한을 식별할 수 있습니다.

```
./rbac-lookup | grep -P 'system:(anonymous)|(unauthenticated)'
system:anonymous               cluster-wide        ClusterRole/system:discovery
system:unauthenticated         cluster-wide        ClusterRole/system:discovery
system:unauthenticated         cluster-wide        ClusterRole/system:public-info-
viewer
```

system:public-info-viewer 이외의 역할 또는 ClusterRole은 system:anonymous user 또는 system:unauthenticated 그룹에 바인딩되어서는 안 됩니다.

특정 APIs. 클러스터의 경우 익명 사용자가 특정 APIs에만 액세스할 수 있도록 하고 인증 없이 해당 APIs 노출해도 클러스터가 취약하지 않도록 합니다.

Kubernetes/EKS 버전 1.14 이전에는 system:unauthenticated 그룹이 기본적으로 system:discovery 및 system:basic-user ClusterRoles에 연결되었습니다. 클러스터를 버전 1.14 이상으로 업데이트한 경우에도 클러스터 업데이트가 이러한 권한을 취소하지 않으므로 클러스터에서 이러한 권한이 계속 활성화될 수 있습니다. system:public-info-viewer를 제외한 "system:unauthenticated"가 있는 ClusterRoles를 확인하려면 다음 명령을 실행할 수 있습니다(jq 유틸리티 필요).

```
kubectl get ClusterRoleBinding -o json | jq -r '.items[] | select(.subjects[]?.name
 =="system:unauthenticated") | select(.metadata.name != "system:public-info-viewer")
 | .metadata.name'
```

또한 다음을 사용하여 "system:unuthenticated"를 제외한 모든 역할에서 "system:public-info-viewer"를 제거할 수 있습니다.

```
kubectl get ClusterRoleBinding -o json | jq -r '.items[] | select(.subjects[]?.name
 =="system:unauthenticated") | select(.metadata.name != "system:public-info-viewer") |
 del(.subjects[] | select(.name =="system:unauthenticated"))' | kubectl apply -f -
```

또는 kubectl describe 및 kubectl edit을 사용하여 수동으로 확인하고 제거할 수 있습니다. system:unauthenticated 그룹에 클러스터에 대한 system:discovery 권한이 있는지 확인하려면 다음 명령을 실행합니다.

```
kubectl describe clusterrolebindings system:discovery

Name:         system:discovery
Labels:       kubernetes.io/bootstrapping=rbac-defaults
Annotations:  rbac.authorization.kubernetes.io/autoupdate: true
Role:
  Kind:  ClusterRole
  Name:  system:discovery
Subjects:
  Kind   Name                    Namespace
  ----   ----                    ---------
  Group  system:authenticated
  Group  system:unauthenticated
```

system:unauthenticated 그룹에 클러스터에 대한 system:basic-user 권한이 있는지 확인하려면 다음 명령을 실행합니다.

```
kubectl describe clusterrolebindings system:basic-user

Name:         system:basic-user
Labels:       kubernetes.io/bootstrapping=rbac-defaults
Annotations:  rbac.authorization.kubernetes.io/autoupdate: true
Role:
  Kind:  ClusterRole
  Name:  system:basic-user
```

```
Subjects:
  Kind   Name                     Namespace
  ----   ----                     ---------
  Group  system:authenticated
  Group  system:unauthenticated
```

system:unauthenticated 그룹이 클러스터의 system:discovery 및/또는 system:basic-user ClusterRoles에 바인딩된 경우 system:unauthenticated 그룹에서 이러한 역할을 연결 해제해야 합니다. 다음 명령을 사용하여 system:discovery ClusterRoleBinding을 편집합니다.

```
kubectl edit clusterrolebindings system:discovery
```

위 명령은 아래와 같이 편집기에서 system:discovery ClusterRoleBinding의 현재 정의를 엽니다.

```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file will
 be
# reopened with the relevant failures.
#
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: "2021-06-17T20:50:49Z"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:discovery
  resourceVersion: "24502985"
  selfLink: /apis/rbac.authorization.k8s.io/v1/clusterrolebindings/system%3Adiscovery
  uid: b7936268-5043-431a-a0e1-171a423abeb6
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:discovery
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:authenticated
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:unauthenticated
```

위의 편집기 화면의 "제목" 섹션에서 system:unuthenticated 그룹에 대한 항목을 삭제합니다.

system:basic-user ClusterRoleBinding에 대해 동일한 단계를 반복합니다.

IRSA에서 AWS SDK 세션 재사용

IRSA를 사용하는 경우 AWS SDK를 사용하여 작성된 애플리케이션은 포드에 전달된 토큰을 사용하여를 호출`sts:AssumeRoleWithWebIdentity`하여 임시 AWS 자격 증명을 생성합니다. 이는 컴퓨팅 서비스가 임시 AWS 자격 증명을 Lambda 함수와 같은 AWS 컴퓨팅 리소스에 직접 제공하는 다른 AWS 컴퓨팅 서비스와 다릅니다. 즉, AWS SDK 세션이 초기화될 때마다에 대한 AWS STS 호출`AssumeRoleWithWebIdentity`이 이루어집니다. 애플리케이션이 빠르게 확장되고 많은 AWS SDK 세션을 초기화하는 경우 코드가를 많이 호출하므로 AWS STS에서 제한이 발생할 수 있습니다`AssumeRoleWithWebIdentity`.

이 시나리오를 방지하려면에 대한 불필요한 호출`AssumeRoleWithWebIdentity`이 이루어지지 않도록 애플리케이션 내에서 AWS SDK 세션을 재사용하는 것이 좋습니다.

다음 예제 코드에서 세션은 boto3 python SDK를 사용하여 생성되며 동일한 세션은 클라이언트를 생성하고 Amazon S3 및 Amazon SQS와 상호 작용하는 데 사용됩니다. `AssumeRoleWithWebIdentity`는 한 번만 호출되며 AWS SDK는 자동으로 만료될 `my_session` 때의 자격 증명을 새로 고칩니다.

```
import boto3

= Create your own session

my_session = boto3.session.Session()

= Now we can create low-level clients from our session

sqs = my_session.client(``sqs`') s3 = my_session.client(``s3`')

s3response = s3.list_buckets() sqsresponse = sqs.list_queues()

#print the response from the S3 and SQS APIs print("`s3 response:`")
print(s3response) print("`—`") print("`sqs response:`")
print(sqsresponse)
```

EC2와 같은 다른 AWS 컴퓨팅 서비스에서 IRSA를 사용하는 EKS로 애플리케이션을 마이그레이션하는 경우 이는 특히 중요한 세부 정보입니다. AWS SDK 세션을 초기화하는 다른 컴퓨팅 서비스에서는 사용자가 지시하지 않는 한 AWS STS를 호출하지 않습니다.

대체 접근 방식

IRSA 및 EKS Pod Identity는 포드에 AWS 자격 증명을 할당하는 데 선호되는 방법이지만 애플리케이션에 최신 버전의 AWS SDKs를 포함해야 합니다. 현재 IRSA를 지원하는 SDKs의 전체 목록은 https://docs.aws.amazon.com/eks/latest/userguide/iam-roles-for-service-accounts-minimum-sdk.html, EKS Pod Identity는 https://docs.aws.amazon.com/eks/latest/userguide/pod-id-minimum-sdk.html 참조하십시오. 호환되는 SDK로 즉시 업데이트할 수 없는 애플리케이션이 있는 경우 kube2iam 및 kiam을 포함하여 Kubernetes 포드에 IAM 역할을 할당하는 데 사용할 수 있는 몇 가지 커뮤니티 구축 솔루션이 있습니다. AWS는 이러한 솔루션을 승인, 허용 또는 지원하지 않지만 커뮤니티에서는 IRSA 및 EKS Pod Identity와 유사한 결과를 얻기 위해 이러한 솔루션을 자주 사용합니다.

aws에서 제공하지 않는 이러한 솔루션 중 하나를 사용해야 하는 경우 실사를 수행하고 그렇게 하면 보안에 미치는 영향을 이해해야 합니다.

도구 및 리소스

- Amazon EKS Security Immersion 워크숍 - 자격 증명 및 액세스 관리
- Terraform EKS 청사진 패턴 - 완전 프라이빗 Amazon EKS 클러스터
- Terraform EKS 블루프린트 패턴 - Amazon EKS 클러스터용 IAM Identity Center Single Sign-On
- Terraform EKS 블루프린트 패턴 - Amazon EKS 클러스터용 Okta Single Sign-On
- audit2rbac
- rbac.dev Kubernetes RBAC에 대한 블로그 및 도구를 포함한 추가 리소스 목록
- 액션 히어로
- kube2iam
- 키암

포드 보안

포드 사양에는 전반적인 보안 태세를 강화하거나 약화시킬 수 있는 다양한 속성이 포함되어 있습니다. Kubernetes 실무자는 컨테이너에서 실행 중인 프로세스가 컨테이너 런타임의 격리 경계를 벗어나 기본 호스트에 액세스하는 것을 방지해야 합니다.

Linux 기능

컨테이너 내에서 실행되는 프로세스는 기본적으로 [Linux] 루트 사용자의 컨텍스트에서 실행됩니다. 컨테이너 내의 루트 작업은 컨테이너 런타임이 컨테이너에 할당하는 Linux 기능 세트에 의해 부분적으

로 제한되지만, 이러한 기본 권한을 통해 공격자는 자신의 권한을 에스컬레이션하고/하거나 Secrets 및 ConfigMaps를 포함하여 호스트에 바인딩된 민감한 정보에 액세스할 수 있습니다. 다음은 컨테이너에 할당된 기본 기능 목록입니다. 각 기능에 대한 자세한 내용은 http://man7.org/linux/man-pages/man7/capabilities.7.html 참조하십시오.

```
CAP_AUDIT_WRITE, CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_FOWNER, CAP_FSETID,
CAP_KILL, CAP_MKNOD, CAP_NET_BIND_SERVICE, CAP_NET_RAW, CAP_SETGID,
CAP_SETUID, CAP_SETFCAP, CAP_SETPCAP, CAP_SYS_CHROOT
```

Example

EC2 및 Fargate 포드에는 기본적으로 앞서 언급한 기능이 할당됩니다. 또한 Linux 기능은 Fargate 포드에서만 삭제할 수 있습니다.

권한이 있는 상태로 실행되는 포드는 호스트의 루트와 연결된 모든 Linux 기능을 상속합니다. 가능하면 이를 피해야 합니다.

노드 권한 부여

모든 Kubernetes 작업자 노드는 노드 권한 부여라는 권한 부여 모드를 사용합니다. 노드 권한 부여는 kubelet에서 시작된 모든 API 요청을 승인하고 노드가 다음 작업을 수행하도록 허용합니다.

읽기 작업:

- 서비스
- 엔드포인트
- 노드
- 포드
- 보안 암호, 구성 맵, 영구 볼륨 클레임 및 kubelet 노드에 바인딩된 포드와 관련된 영구 볼륨

쓰기 작업:

- 노드 및 노드 상태(`NodeRestriction`인증 플러그인을 활성화하여 kubelet이 자체 노드를 수정하도록 제한)
- 포드 및 포드 상태(`NodeRestriction`인증 플러그인을 활성화하여 kubelet이 자신에게 바인딩된 포드를 수정하도록 제한)
- 이벤트

인증 관련 작업:

- TLS 부트스트래핑을 위한 CertificateSigningRequest(CSR) API에 대한 읽기/쓰기 액세스
- 위임된 인증/권한 부여 검사를 위해 TokenReview 및 SubjectAccessReview를 생성하는 기능

EKS는 노드가 노드에 바인딩된 제한된 노드 속성 및 포드 객체 집합만 수정할 수 있도록 허용하는 노드 제한 승인 컨트롤러를 사용합니다. 그러나 호스트에 액세스하도록 관리하는 공격자는 여전히 Kubernetes API에서 환경에 대한 민감한 정보를 수집하여 클러스터 내에서 측면으로 이동할 수 있습니다.

포드 보안 솔루션

포드 보안 정책(PSP)

과거에는 포드 보안 정책(PSP) 리소스를 사용하여 포드가 생성되기 전에 충족해야 하는 요구 사항 세트를 지정했습니다. Kubernetes 버전 1.21부터 PSP는 더 이상 사용되지 않습니다. Kubernetes 버전 1.25에서 제거될 예정입니다.

> **Important**
>
> PSPs는 Kubernetes 버전 1.21에서 더 이상 사용되지 않습니다. 버전 1.25까지 또는 약 2년 동안 대체 버전으로 전환해야 합니다. 이 문서에서는 이러한 사용 중단의 동기를 설명합니다.

새 포드 보안 솔루션으로 마이그레이션

Kubernetes v1.25부터 PSPs 제거되었으므로 클러스터 관리자와 운영자는 이러한 보안 제어를 교체해야 합니다. 이 요구 사항을 충족할 수 있는 두 가지 솔루션은 다음과 같습니다.

- Kubernetes 에코시스템의 Policy-as-code(PAC) 솔루션
- Kubernetes 포드 보안 표준(PSS)

PAC 및 PSS 솔루션은 모두 PSP와 공존할 수 있으며, PSP가 제거되기 전에 클러스터에서 사용할 수 있습니다. 이렇게 하면 PSP에서 마이그레이션할 때 쉽게 채택할 수 있습니다. PSP에서 PSS로 마이그레이션을 고려할 때는이 문서를 참조하세요.

아래에 설명된 PAC 솔루션 중 하나인 Kyverno는 PSPs에서 솔루션으로 마이그레이션할 때 유사한 정책, 기능 비교 및 마이그레이션 절차를 포함하여 블로그 게시물에 설명된 특정 지침을 제공합니다. 포

드 보안 승인(PSA)과 관련하여 Kyverno로 마이그레이션하는 방법에 대한 추가 정보와 지침은 여기 AWS 블로그에 게시되어 있습니다.

Policy-as-code(PAC)

Policy-as-code(PAC) 솔루션은 규정된 자동 제어를 통해 클러스터 사용자를 안내하고 원치 않는 동작을 방지하는 가드레일을 제공합니다. PAC는 Kubernetes Dynamic Admission Controllers를 사용하여 웹후크 호출을 통해 Kubernetes API 서버 요청 흐름을 가로채고 코드로 작성 및 저장된 정책에 따라 요청 페이로드를 변경하고 검증합니다. 변형 및 검증은 API 서버 요청으로 인해 클러스터가 변경되기 전에 발생합니다. PAC 솔루션은 정책을 사용하여 분류 체계 및 값을 기반으로 API 서버 요청 페이로드를 일치시키고 이에 따라 조치를 취합니다.

Kubernetes에는 몇 가지 오픈 소스 PAC 솔루션을 사용할 수 있습니다. 이러한 솔루션은 Kubernetes 프로젝트의 일부가 아니며 Kubernetes 에코시스템에서 소싱됩니다. 일부 PAC 솔루션은 아래에 나열되어 있습니다.

- OPA/게이트키퍼
- 정책 에이전트(OPA) 열기
- 키버노
- Kubewarden
- jsPolicy

PAC 솔루션과 필요에 적합한 솔루션을 선택하는 방법에 대한 자세한 내용은 아래 링크를 참조하세요.

- Kubernetes에 대한 정책 기반 대응 - 1부
- Kubernetes에 대한 정책 기반 대응 - 2부

포드 보안 표준(PSS) 및 포드 보안 승인(PSA)

PSP 사용 중단 및 기본 제공 Kubernetes 솔루션을 사용하여 포드 보안을 out-of-the-box 제어해야 하는 지속적인 필요성에 대응하여 Kubernetes 인증 특별 관심 그룹은 포드 보안 표준(PSS) 및 포드 보안 승인(PSA)을 생성했습니다. PSA 작업에는 PSS에 정의된 제어를 구현하는 승인 컨트롤러 웹후크 프로젝트가 포함됩니다. 이 승인 컨트롤러 접근 방식은 PAC 솔루션에 사용된 것과 유사합니다.

Kubernetes 설명서에 따르면 PSS "`는 보안 스펙트럼을 광범위하게 다루기 위해 세 가지 정책을 정의합니다. 이러한 정책은 누적되며 매우 허용적인 정책부터 매우 제한적인 정책까지 다양합니다."

이러한 정책은 다음과 같이 정의됩니다.

- 권한: 무제한(보안되지 않음) 정책으로 가능한 가장 광범위한 수준의 권한을 제공합니다. 이 정책은 알려진 권한 에스컬레이션을 허용합니다. 정책이 없습니다. 이는 로깅 에이전트, CNIs, 스토리지 드라이버 및 권한 있는 액세스가 필요한 기타 시스템 전체 애플리케이션과 같은 애플리케이션에 적합합니다.
- 기준: 알려진 권한 에스컬레이션을 방지하는 최소 제한 정책입니다. 기본(최소 지정) 포드 구성을 허용합니다. 기본 정책은 기타 여러 제한 사항과 함께 hostNetwork, hostPID, hostIPC, hostPath, hostPort, Linux 기능을 추가할 수 없는 사용을 금지합니다.
- 제한됨: 현재 포드 강화 모범 사례에 따라 정책이 엄격하게 제한되었습니다. 이 정책은 기준에서 상속되며 루트 또는 루트 그룹으로 실행할 수 없는 것과 같은 추가 제한을 추가합니다. 제한된 정책은 애플리케이션의 기능 기능에 영향을 미칠 수 있습니다. 주로 보안에 중요한 애플리케이션을 실행하는 것을 대상으로 합니다.

이러한 정책은 포드 실행을 위한 프로파일을 정의하며, 세 가지 수준의 권한 있는 액세스와 제한된 액세스로 구성됩니다.

PSS에서 정의한 제어를 구현하기 위해 PSA는 세 가지 모드로 작동합니다.

- 적용: 정책 위반으로 인해 포드가 거부됩니다.
- 감사: 정책 위반으로 인해 감사 로그에 기록된 이벤트에 감사 주석이 추가되지만, 그렇지 않으면 허용됩니다.
- 경고: 정책 위반은 사용자 대면 경고를 트리거하지만 그렇지 않으면 허용됩니다.

이러한 모드와 프로파일(제한) 수준은 아래 예제와 같이 레이블을 사용하여 Kubernetes 네임스페이스 수준에서 구성됩니다.

```
apiVersion: v1
kind: Namespace
metadata:
  name: policy-test
  labels:
    pod-security.kubernetes.io/enforce: restricted
```

이러한 운영 모드를 독립적으로 사용하면 응답이 다르기 때문에 사용자 경험이 달라집니다. 적용 모드는 각 podSpecs가 생성되지 않도록 합니다. 그러나이 모드에서는 배포와 같이 포드를 생성하는 포드가 아닌 Kubernetes 객체가 클러스터에 적용되는 PSS를 podSpec 위반하더라도 클러스터에 적용되지 않습니다. 이 경우 배포가 적용되고 포드(들)가 적용되지 않습니다.

성공적으로 적용된 배포 객체가 포드 생성에 실패했다는 즉각적인 표시가 없으므로 이는 어려운 사용자 경험입니다. 문제가 되는 podSpecs는 포드를 생성하지 않습니다. 를 사용하여 배포 리소스를 검사kubectl get deploy <DEPLOYMENT_NAME> -oyaml하면 아래와 같이 실패한 포드(들) .status.conditions 요소의 메시지가 표시됩니다.

```
...
status:
  conditions:
    - lastTransitionTime: "2022-01-20T01:02:08Z"
      lastUpdateTime: "2022-01-20T01:02:08Z"
      message: 'pods "test-688f68dc87-tw587" is forbidden: violates PodSecurity
 "restricted:latest":
        allowPrivilegeEscalation != false (container "test" must set
 securityContext.allowPrivilegeEscalation=false),
        unrestricted capabilities (container "test" must set
 securityContext.capabilities.drop=["ALL"]),
        runAsNonRoot != true (pod or container "test" must set
 securityContext.runAsNonRoot=true),
        seccompProfile (pod or container "test" must set
 securityContext.seccompProfile.type
        to "RuntimeDefault" or "Localhost")'
      reason: FailedCreate
      status: "True"
      type: ReplicaFailure
...
```

감사 및 경고 모드 모두에서 포드 제한으로 인해 포드 위반이 생성 및 시작되지는 않습니다. 그러나 이러한 모드에서는 kubectl과 같은 API 서버 감사 로그 이벤트 및 API 서버 클라이언트에 대한 경고에 대한 감사 주석이 포드와 포드를 생성하는 객체에 위반이 있는 podSpecs 포함된 경우 각각 트리거됩니다. kubectl 경고 메시지는 다음과 같습니다.

```
Warning: would violate PodSecurity "restricted:latest":
 allowPrivilegeEscalation != false (container "test" must set
 securityContext.allowPrivilegeEscalation=false), unrestricted capabilities (container
 "test" must set securityContext.capabilities.drop=["ALL"]), runAsNonRoot != true (pod
 or container "test" must set securityContext.runAsNonRoot=true), seccompProfile (pod
 or container "test" must set securityContext.seccompProfile.type to "RuntimeDefault"
 or "Localhost")
deployment.apps/test created
```

PSA 감사 및 경고 모드는 클러스터 작업에 부정적인 영향을 주지 않고 PSS를 도입할 때 유용합니다.

PSA 운영 모드는 상호 배타적이지 않으며 누적 방식으로 사용할 수 있습니다. 아래 표시된 것처럼 여러 모드를 단일 네임스페이스에서 구성할 수 있습니다.

```
apiVersion: v1
kind: Namespace
metadata:
  name: policy-test
  labels:
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/warn: restricted
```

위 예제에서는 배포를 적용할 때 사용자에게 친숙한 경고 및 감사 주석이 제공되는 반면, 위반 적용은 포드 수준에서도 제공됩니다. 실제로 여러 PSA 레이블은 아래와 같이 서로 다른 프로필 수준을 사용할 수 있습니다.

```
apiVersion: v1
kind: Namespace
metadata:
  name: policy-test
  labels:
    pod-security.kubernetes.io/enforce: baseline
    pod-security.kubernetes.io/warn: restricted
```

위 예제에서 PSA는 기준 프로파일 수준을 충족하는 모든 포드를 생성할 수 있도록 구성된 다음 제한된 프로파일 수준을 위반하는 포드(및 포드를 생성하는 객체)에 대해 경고합니다. 이는 기준에서 제한된 프로필로 변경할 때 발생할 수 있는 영향을 결정하는 데 유용한 접근 방식입니다.

기존 포드

기존 포드가 있는 네임스페이스가 보다 제한적인 PSS 프로파일을 사용하도록 수정된 경우 감사 및 경고 모드는 적절한 메시지를 생성하지만 적용 모드는 포드를 삭제하지 않습니다. 경고 메시지는 다음과 같습니다.

```
Warning: existing pods in namespace "policy-test" violate the new PodSecurity enforce
 level "restricted:latest"
Warning: test-688f68dc87-htm8x: allowPrivilegeEscalation != false, unrestricted
 capabilities, runAsNonRoot != true, seccompProfile
namespace/policy-test configured
```

예외

PSA는 예외를 사용하여 달리 적용되었을 포드에 대한 위반 적용을 제외합니다. 이러한 예외는 아래에 나열되어 있습니다.

- 사용자 이름: 인증된(또는 가장된) 사용자 이름이 면제된 사용자의 요청은 무시됩니다.
- RuntimeClassNames: 면제 런타임 클래스 이름을 지정하는 포드 및 워크로드 리소스는 무시됩니다.
- 네임스페이스: 제외된 네임스페이스의 포드 및 워크로드 리소스는 무시됩니다.

이러한 예외는 API 서버 구성의 일부로 PSA 승인 컨트롤러 구성에 정적으로 적용됩니다.

Webhook 구현 검증에서 pod-security-webhook 컨테이너에 볼륨으로 마운트되는 Kubernetes ConfigMap 리소스 내에서 예외를 구성할 수 있습니다.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: pod-security-webhook
  namespace: pod-security-webhook
data:
  podsecurityconfiguration.yaml: |
    apiVersion: pod-security.admission.config.k8s.io/v1
    kind: PodSecurityConfiguration
    defaults:
      enforce: "restricted"
      enforce-version: "latest"
      audit: "restricted"
      audit-version: "latest"
      warn: "restricted"
      warn-version: "latest"
    exemptions:
      # Array of authenticated usernames to exempt.
      usernames: []
      # Array of runtime class names to exempt.
      runtimeClasses: []
      # Array of namespaces to exempt.
      namespaces: ["kube-system","policy-test1"]
```

위의 ConfigMap YAML에서 볼 수 있듯이 클러스터 전체의 기본 PSS 수준은 모든 PSA 모드, 감사, 적용 및 경고에 대해 제한으로 설정되었습니다. 이는 제외된 네임스페이스를 제외한 모든 네임스페이스에 영향을 미칩니다`namespaces: ["kube-system","policy-test1"]`. 또한 아래 표시된

ValidatingWebhookConfiguration 리소스에서 pod-security-webhook 네임스페이스도 구성된 PSS에서 제외됩니다.

```
...
webhooks:
  # Audit annotations will be prefixed with this name
  - name: "pod-security-webhook.kubernetes.io"
    # Fail-closed admission webhooks can present operational challenges.
    # You may want to consider using a failure policy of Ignore, but should
    # consider the security tradeoffs.
    failurePolicy: Fail
    namespaceSelector:
      # Exempt the webhook itself to avoid a circular dependency.
      matchExpressions:
        - key: kubernetes.io/metadata.name
          operator: NotIn
          values: ["pod-security-webhook"]
...
```

Important

포드 보안 승인은 Kubernetes v1.25에서 안정적인 상태로 전환되었습니다. 기본적으로 활성화되기 전에 포드 보안 승인 기능을 사용하려면 동적 승인 컨트롤러(변환 웹후크)를 설치해야 합니다. Webhook 설치 및 구성 지침은 여기에서 확인할 수 있습니다.

policy-as-code 포드 보안 표준 중에서 선택

포드 보안 표준(PSS)은 Kubernetes에 기본 제공되고 Kubernetes 에코시스템의 솔루션이 필요하지 않은 솔루션을 제공하여 포드 보안 정책(PSP)을 대체하도록 개발되었습니다. 즉, policy-as-code(PAC) 솔루션은 훨씬 더 유연합니다.

다음 Pros and Cons 목록은 포드 보안 솔루션에 대해 정보에 입각한 결정을 내리는 데 도움이 되도록 설계되었습니다.

Policy-as-code(포드 보안 표준과 비교)

장점:

- 더 유연하고 세분화됨(필요한 경우 리소스의 속성으로 축소)
- 포드에만 초점을 맞출 뿐만 아니라 다양한 리소스 및 작업에 사용할 수 있습니다.

- 네임스페이스 수준에서만 적용되는 것이 아님
- 포드 보안 표준보다 성숙함
- 결정은 API 서버 요청 페이로드의 모든 항목과 기존 클러스터 리소스 및 외부 데이터를 기반으로 할 수 있습니다(솔루션에 따라 다름).
- 검증 전 API 서버 요청 변경 지원(솔루션에 따라 다름)
- 보완 정책 및 Kubernetes 리소스(솔루션 종속 - 포드 정책에서 Kyverno는 배포와 같은 상위 수준 컨트롤러에 대한 정책을 자동 생성할 수 있습니다. 또한 Kyverno는 규칙 생성을 사용하여 추가 Kubernetes 리소스 "`새 리소스가 생성되거나 소스가 업데이트될 때""를 생성할 수 있습니다.)
- Kubernetes API 서버를 호출하기 전에 왼쪽을 CICD 파이프라인으로 전환하는 데 사용할 수 있습니다(솔루션에 따라 다름).
- 모범 사례, 조직 표준 등과 같이 반드시 보안과 관련된 것은 아닌 동작을 구현하는 데 사용할 수 있습니다.
- Kubernetes 이외의 사용 사례에서 사용할 수 있음(솔루션에 따라 다름)
- 유연성 때문에 사용자 경험을 사용자의 요구 사항에 맞게 조정할 수 있습니다.

단점:

- Kubernetes에 내장되지 않음
- 더 복잡한 학습, 구성 및 지원
- 정책 작성에는 새로운 skills/languages/capabilities 필요할 수 있습니다.

포드 보안 승인(policy-as-code 비교)

장점:

- Kubernetes에 내장
- 더 간단한 구성
- 작성할 새 언어 또는 정책 없음
- 클러스터 기본 승인 수준이 권한으로 구성된 경우 네임스페이스 레이블을 사용하여 포드 보안 프로필에 네임스페이스를 선택할 수 있습니다.

단점:

- policy-as-code만큼 유연하거나 세분화되지 않음

- 3가지 제한 수준만
- 주로 포드에 집중

요약

현재 PSP 이외의 포드 보안 솔루션이 없고 필요한 포드 보안 태세가 포드 보안 표준(PSS)에 정의된 모델에 맞는 경우 policy-as-code 솔루션 대신 PSS를 채택하는 것이 더 쉬운 경로일 수 있습니다. 그러나 포드 보안 태세가 PSS 모델에 맞지 않거나 PSS에서 정의한 것 외에 추가 제어를 추가할 계획이라면 policy-as-code 솔루션이 더 적합한 것처럼 보일 것입니다.

권장 사항

더 나은 사용자 경험을 위해 여러 PSA(포드 보안 승인) 모드 사용

앞서 언급했듯이 PSA 적용 모드는 PSS 위반이 있는 포드가 적용되지 않도록 방지하지만 배포와 같은 상위 수준 컨트롤러를 중지하지는 않습니다. 실제로 포드를 적용하지 못했다는 표시 없이 배포가 성공적으로 적용됩니다. kubectl을 사용하여 배포 객체를 검사하고 PSA에서 실패한 포드 메시지를 검색할 수 있지만 사용자 경험이 더 나을 수 있습니다. 사용자 경험을 개선하려면 여러 PSA 모드(감사, 시행, 경고)를 사용해야 합니다.

```
apiVersion: v1
kind: Namespace
metadata:
  name: policy-test
  labels:
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/warn: restricted
```

위 예제에서 적용 모드가 정의되면 해당 podSpec에서 PSS 위반이 있는 배포 매니페스트를 Kubernetes API 서버에 적용하려고 하면 배포가 성공적으로 적용되지만 포드는 적용되지 않습니다. 또한 감사 및 경고 모드도 활성화되어 있으므로 API 서버 클라이언트는 경고 메시지를 수신하고 API 서버 감사 로그 이벤트에도 메시지로 주석이 추가됩니다.

권한 있는 상태로 실행할 수 있는 컨테이너 제한

앞서 언급했듯이 권한으로 실행되는 컨테이너는 호스트의 루트에 할당된 모든 Linux 기능을 상속합니다. 컨테이너가 제대로 작동하려면 이러한 유형의 권한이 거의 필요하지 않습니다. 컨테이너의 권한과 기능을 제한하는 데 사용할 수 있는 여러 가지 방법이 있습니다.

> **Important**
>
> Fargate는 포드의 컨테이너가 AWS가 관리하는 인프라에서 실행되는 "서버리스" 컨테이너(들)를 실행할 수 있는 시작 유형입니다. Fargate에서는 권한이 있는 컨테이너를 실행하거나 hostNetwork 또는 hostPort를 사용하도록 포드를 구성할 수 없습니다.

컨테이너에서 프로세스를 루트로 실행하지 마십시오.

모든 컨테이너는 기본적으로 루트로 실행됩니다. 이는 공격자가 애플리케이션의 취약성을 악용하고 실행 중인 컨테이너에 대한 셸 액세스를 가져올 수 있는 경우 문제가 될 수 있습니다. 다양한 방법으로 이 위험을 완화할 수 있습니다. 먼저 컨테이너 이미지에서 쉘을 제거합니다. 둘째, Dockerfile에 사용자 지시문을 추가하거나 루트가 아닌 사용자로 포드의 컨테이너를 실행합니다. Kubernetes podSpec에는 에 애플리케이션을 실행할 사용자 및/또는 그룹을 지정할 수 `spec.securityContext`있는 필드 세트가 포함되어 있습니다. 이러한 필드는 `runAsGroup` 각각 `runAsUser` 및 입니다.

Kubernetes podSpec 내에서 `spec.securityContext`및 관련 요소의 사용을 강제하기 위해 policy-as-code 또는 포드 보안 표준을 클러스터에 추가할 수 있습니다. 이러한 솔루션을 사용하면 인바운드 Kubernetes API 서버 요청 페이로드를 etcd에 유지되기 전에 검증할 수 있는 정책 또는 프로파일을 작성 및/또는 사용할 수 있습니다. 또한 policy-as-code 솔루션은 인바운드 요청을 변경하고 경우에 따라 새 요청을 생성할 수 있습니다.

Docker에서 Docker를 실행하거나 소켓을 컨테이너에 탑재하지 마십시오.

이렇게 하면 Docker 컨테이너에서 이미지를 편리하게 빌드/실행할 수 있지만 기본적으로 노드에 대한 완전한 제어를 컨테이너에서 실행되는 프로세스로 해제하는 것입니다. Kubernetes에서 컨테이너 이미지를 빌드해야 하는 경우 Kaniko, buildah 또는 CodeBuild와 같은 빌드 서비스를 대신 사용합니다.

> **Note**
>
> 컨테이너 이미지 빌드와 같은 CICD 처리에 사용되는 Kubernetes 클러스터는 보다 일반화된 워크로드를 실행하는 클러스터와 격리되어야 합니다.

hostPath 사용을 제한하거나 hostPath가 필요한 경우 사용할 수 있는 접두사를 제한하고 볼륨을 읽기 전용으로 구성합니다.

`hostPath`는 호스트에서 컨테이너로 직접 디렉터리를 탑재하는 볼륨입니다. 포드에 이러한 유형의 액세스가 필요한 경우는 드물지만 필요한 경우 위험을 인식해야 합니다. 기본적으로 루트로 실행되는

포드는 hostPath에서 노출되는 파일 시스템에 대한 쓰기 액세스 권한을 갖습니다. 이를 통해 공격자는 kubelet 설정을 수정하고, /etc/shadow와 같이 hostPath에서 직접 노출되지 않은 디렉터리 또는 파일에 대한 심볼 링크를 생성하고, ssh 키를 설치하고, 호스트에 탑재된 암호를 읽을 수 있습니다. hostPath로 인한 위험을 완화하려면를 `spec.containers.volumeMounts`로 구성합니다`readOnly`. 예를 들면 다음과 같습니다.

```
volumeMounts:
- name: hostPath-volume
    readOnly: true
    mountPath: /host-path
```

또한 policy-as-code 솔루션을 사용하여 `hostPath` 볼륨에서 사용할 수 있는 디렉터리를 제한하거나 `hostPath` 사용을 완전히 방지해야 합니다. 포드 보안 표준 기준 또는 제한 정책을 사용하여 사용을 방지할 수 있습니다`hostPath`.

권한 에스컬레이션의 위험에 대한 자세한 내용은 Seth Art의 Bad Pods: Kubernetes Pod Privilege Escalation 블로그를 참조하세요.

리소스 경합 및 DoS 공격을 방지하기 위해 각 컨테이너에 대한 요청 및 제한 설정

요청이나 제한이 없는 포드는 이론적으로 호스트에서 사용 가능한 모든 리소스를 사용할 수 있습니다. 노드에 추가 포드가 예약되면 노드에 CPU 또는 메모리 압력이 발생하여 Kubelet이 노드에서 포드를 종료하거나 제거할 수 있습니다. 이러한 일이 모두 발생하는 것을 방지할 수는 없지만 요청과 제한을 설정하면 리소스 경합을 최소화하고 과도한 양의 리소스를 소비하는 잘못 작성된 애플리케이션의 위험을 완화하는 데 도움이 됩니다.

를 `podSpec` 사용하면 CPU 및 메모리에 대한 요청 및 제한을 지정할 수 있습니다. CPU는 과도하게 구독할 수 있으므로 압축 가능한 리소스로 간주됩니다. 메모리는 압축할 수 없습니다. 즉, 여러 컨테이너 간에 공유할 수 없습니다.

CPU 또는 메모리에 대한 요청을 지정할 때 기본적으로 컨테이너가 가져오도록 보장되는 메모리 양을 지정합니다. Kubernetes는 포드에 있는 모든 컨테이너의 요청을 집계하여 포드를 예약할 노드를 결정합니다. 컨테이너가 요청된 메모리 양을 초과하는 경우 노드에 메모리 압력이 있는 경우 종료될 수 있습니다.

제한은 컨테이너가 사용할 수 있는 최대 CPU 및 메모리 리소스 양이며 컨테이너에 대해 생성된 cgroup의 `memory.limit_in_bytes` 값에 직접 해당합니다. 메모리 제한을 초과하는 컨테이너는 OOM이 종료됩니다. 컨테이너가 CPU 제한을 초과하면 병목 현상이 발생합니다.

> **Note**
>
> 컨테이너를 사용할 때는 로드 테스트를 기반으로 컨테이너 리소스 사용량(리소스 풋프린트라고도 함)이 데이터 기반이고 정확`resources.limits`해야 합니다. 정확하고 신뢰할 수 있는 리소스 공간이 없으면 컨테이너를 채울 `resources.limits` 수 있습니다. 예를 들어,는 잠재적인 메모리 리소스 제한 부정확성을 고려하기 위해 관찰 가능한 최대값보다 20~30% 더 높게 패딩될 `resources.limits.memory` 수 있습니다.

Kubernetes는 세 가지 QoS(서비스 품질) 클래스를 사용하여 노드에서 실행되는 워크로드의 우선 순위를 지정합니다. 다음이 포함됩니다.

- 보장
- 버스트 가능
- 최선의 노력

제한 및 요청이 설정되지 않은 경우 포드는 최선의 노력(가장 낮은 우선 순위)으로 구성됩니다. 메모리가 부족할 때 가장 먼저 죽는 것이 최선의 포드입니다. 포드 내의 모든 컨테이너에 제한이 설정되거나 요청 및 제한이 0이 아닌 동일한 값으로 설정된 경우 포드가 보장됨(가장 높은 우선 순위)으로 구성됩니다. 보장된 포드는 구성된 메모리 제한을 초과하지 않는 한 종료되지 않습니다. 제한 및 요청이 0이 아닌 다른 값으로 구성되거나 포드 내의 컨테이너 하나가 제한을 설정하고 다른 컨테이너가 다른 리소스에 대한 제한을 설정하지 않거나 가지고 있는 경우 포드는 버스트 가능(중간 우선 순위)으로 구성됩니다. 이러한 포드에는 일부 리소스 보장이 있지만 요청된 메모리를 초과하면 종료할 수 있습니다.

> **Important**
>
> 요청은 컨테이너의 cgroup `memory_limit_in_bytes` 값에 영향을 주지 않습니다. cgroup 제한은 호스트에서 사용할 수 있는 메모리 양으로 설정됩니다. 하지만 요청 값을 너무 낮게 설정하면 노드에 메모리 압력이 가해지면 포드가 kubelet에 의해 종료 대상으로 지정될 수 있습니다.

Class	우선순위	조건	조건 종료
보장	가장 높음	limit = 요청 != 0	메모리 제한만 초과

Class	우선순위	조건	조건 종료
버스트 가능	medium	limit != 요청 != 0	요청 메모리를 초과하는 경우 종료될 수 있음
최선의 노력	가장 낮음	제한 및 요청 설정되지 않음	첫 번째는 메모리가 부족할 때 죽는 것입니다.

리소스 QoS에 대한 자세한 내용은 Kubernetes 설명서를 참조하세요.

네임스페이스에 리소스 할당량을 설정하거나 제한 범위를 생성하여 요청 및 제한을 강제로 사용할 수 있습니다. 리소스 할당량을 사용하면 네임스페이스에 할당된 CPU 및 RAM과 같은 총 리소스 양을 지정할 수 있습니다. 네임스페이스에 적용되면 해당 네임스페이스에 배포된 모든 컨테이너에 대한 요청 및 제한을 지정해야 합니다. 반대로 제한 범위를 사용하면 리소스 할당을 보다 세밀하게 제어할 수 있습니다. 제한 범위를 사용하면 포드당 또는 네임스페이스 내의 컨테이너당 CPU 및 메모리 리소스에 대해 최소/최대를 지정할 수 있습니다. 제공되지 않은 경우 기본 요청/한도 값을 설정하는 데에도 사용할 수 있습니다.

Policy-as-code 솔루션을 사용하면 요청 및 제한을 적용할 수 있습니다. 또는 네임스페이스를 생성할 때 리소스 할당량 및 제한 범위를 생성할 수도 있습니다.

권한 있는 에스컬레이션 허용 안 함

권한 에스컬레이션을 사용하면 프로세스가 실행 중인 보안 컨텍스트를 변경할 수 있습니다. Sudo는 SUID 또는 SGID 비트가 있는 바이너리와 마찬가지로 이에 대한 좋은 예입니다. 권한 에스컬레이션은 기본적으로 사용자가 다른 사용자 또는 그룹의 권한으로 파일을 실행하는 방법입니다. 를 로 설정`false`하거나에서를 설정하는 policy-as-code 변경 정책을 구현하여 컨테이너가 권한 있는 에스컬레이션을 사용하지 못하도록 `allowPrivilegeEscalation`할 수 `securityContext.allowPrivilegeEscalation` 있습니다`podSpec`. 잘못된 설정이 감지되면 Policy-as-code 정책을 사용하여 API 서버 요청이 성공하지 못하도록 할 수도 있습니다. 포드 보안 표준을 사용하여 포드가 권한 에스컬레이션을 사용하지 못하도록 할 수도 있습니다.

ServiceAccount 토큰 탑재 비활성화

Kubernetes API에 액세스할 필요가 없는 포드의 경우 포드 사양 또는 특정 ServiceAccount를 사용하는 모든 포드에 ServiceAccount 토큰의 자동 탑재를 비활성화할 수 있습니다.

Example

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-no-automount
spec:
  automountServiceAccountToken: false
```

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sa-no-automount
automountServiceAccountToken: false
```

서비스 검색 비활성화

클러스터 내 서비스를 조회하거나 호출할 필요가 없는 포드의 경우 포드에 제공된 정보의 양을 줄일 수 있습니다. CoreDNS를 사용하지 않고 포드 네임스페이스의 서비스를 환경 변수로 노출하지 않도록 포드의 DNS 정책을 설정할 수 있습니다. 서비스 링크에 대한 자세한 내용은 환경 변수에 대한 Kubernetes 문서를 참조하세요. 포드의 DNS 정책의 기본값은 클러스터 내 DNS를 사용하는 "ClusterFirst"인 반면, 기본값이 아닌 "Default"는 기본 노드의 DNS 확인을 사용합니다. 자세한 내용은 포드 DNS 정책에 대한 Kubernetes 문서를 참조하세요.

Example

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-no-service-info
spec:
    dnsPolicy: Default # "Default" is not the true default value
    enableServiceLinks: false
```

읽기 전용 루트 파일 시스템으로 이미지 구성

읽기 전용 루트 파일 시스템으로 이미지를 구성하면 공격자가 애플리케이션이 사용하는 파일 시스템에서 바이너리를 덮어쓰지 못합니다. 애플리케이션이 파일 시스템에 작성해야 하는 경우 임시 디렉터리에 쓰거나 볼륨을 연결 및 탑재하는 것이 좋습니다. 포드의 SecurityContext를 다음과 같이 설정하여 이를 적용할 수 있습니다.

```
...
securityContext:
  readOnlyRootFilesystem: true
...
```

Policy-as-code 및 포드 보안 표준을 사용하여이 동작을 적용할 수 있습니다.

Example

레지스트리 및 시스템 프로세스가 컨테이너 내에서 실행되려면 쓰기 액세스가 필요하므로 Windows에서 실행되는 컨테이너에 `true` 대해 Kubernetes의 Windows 컨테이너를 로 설정할 수 `securityContext.readOnlyRootFilesystem` 없습니다.

도구 및 리소스

- Amazon EKS Security Immersion 워크숍 - 포드 보안
- open-policy-agent/gatekeeper-library: OPA Gatekeeper 정책 라이브러리는 PSPs.
- Kyverno 정책 라이브러리
- EKS에 대한 일반적인 OPA 및 Kyverno 정책 모음입니다.
- 정책 기반 대응책: 1부
- 정책 기반 대응책: 2부
- 포드 보안 정책 마이그레이션 PSPs를 OPA/Gatekeeper, KubeWarden 또는 Kyverno 정책으로 변환하는 도구
- SUSE 오픈 소스의 NeuVector, 제로 트러스트 컨테이너 보안 플랫폼은 프로세스 및 파일 시스템 정책과 승인 제어 규칙을 제공합니다.

테넌트 격리

다중 테넌시를 생각할 때 종종 공유 인프라에서 실행되는 다른 사용자 또는 애플리케이션과 사용자 또는 애플리케이션을 격리하려고 합니다.

Kubernetes는 단일 테넌트 오케스트레이터입니다. 즉, 컨트롤 플레인의 단일 인스턴스가 클러스터 내의 모든 테넌트 간에 공유됩니다. 그러나 다중 테넌시의 스셈을 생성하는 데 사용할 수 있는 다양한 Kubernetes 객체가 있습니다. 예를 들어 네임스페이스와 역할 기반 액세스 제어(RBAC)를 구현하여 테넌트를 서로 논리적으로 격리할 수 있습니다. 마찬가지로 할당량 및 제한 범위를 사용하여 각 테넌트가 사용할 수 있는 클러스터 리소스의 양을 제어할 수 있습니다. 그럼에도 불구하고 클러스터는 강력한 보안 경계를 제공하는 유일한 구문입니다. 이는 클러스터 내의 호스트에 액세스하기 위해를 관리하는 공

격자가 해당 호스트에 탑재된 모든 보안 암호, ConfigMaps 및 볼륨을 검색할 수 있기 때문입니다. 또한 Kubelet을 가장하여 노드의 속성을 조작하거나 클러스터 내에서 측면으로 이동할 수 있습니다.

다음 섹션에서는 Kubernetes와 같은 단일 테넌트 오케스트레이터 사용의 위험을 완화하면서 테넌트 격리를 구현하는 방법을 설명합니다.

소프트 멀티테넌시

소프트 멀티테넌시에서는 네임스페이스, 역할 및 역할 바인딩, 네트워크 정책과 같은 네이티브 Kubernetes 구문을 사용하여 테넌트 간에 논리적 분리를 생성합니다. 예를 들어 RBAC는 테넌트가 서로의 리소스에 액세스하거나 조작하지 못하도록 할 수 있습니다. 할당량 및 제한 범위는 각 테넌트가 사용할 수 있는 클러스터 리소스의 양을 제어하는 반면, 네트워크 정책은 서로 다른 네임스페이스에 배포된 애플리케이션이 서로 통신하는 것을 방지하는 데 도움이 될 수 있습니다.

그러나 이러한 제어 중 어느 것도 다른 테넌트의 포드가 노드를 공유하는 것을 방지하지 않습니다. 더 강력한 격리가 필요한 경우 노드 선택기, 친화 방지 규칙 및/또는 테인트 및 허용치를 사용하여 여러 테넌트의 포드를 별도의 노드로 강제 예약할 수 있습니다. 이를 종종 유일한 테넌트 노드라고 합니다. 테넌트가 많은 환경에서는 다소 복잡하고 비용이 많이 들 수 있습니다.

> **Important**
>
> 네임스페이스로 구현된 소프트 다중 테넌시는 네임스페이스가 전역적으로 범위가 지정된 유형이므로 테넌트에게 필터링된 네임스페이스 목록을 제공할 수 없습니다. 테넌트가 특정 네임스페이스를 볼 수 있는 경우 클러스터 내의 모든 네임스페이스를 볼 수 있습니다.

> **Warning**
>
> soft-multi-tenancy를 사용하면 테넌트는 기본적으로 클러스터 내에서 실행되는 모든 서비스에 대해 CoreDNS를 쿼리할 수 있는 기능을 유지합니다. 공격자는 클러스터의 모든 포드 `..svc.cluster.local`에서 dig SRV를 실행하여 이를 악용할 수 있습니다. 클러스터 내에서 실행되는 서비스의 DNS 레코드에 대한 액세스를 제한해야 하는 경우 CoreDNS용 방화벽 또는 정책 플러그인을 사용하는 것이 좋습니다. 자세한 내용은 https://github.com/coredns/policy#kubernetes-metadata-multi-tenancy-policy 참조하십시오.

Kiosk는 소프트 멀티테넌시 구현을 지원할 수 있는 오픈 소스 프로젝트입니다. 다음 기능을 제공하는 일련의 CRDs 및 컨트롤러로 구현됩니다.

- 공유 Kubernetes 클러스터에서 테넌트를 분리할 계정 및 계정 사용자
- 계정 사용자를 위한 셀프 서비스 네임스페이스 프로비저닝
- 클러스터를 공유할 때 서비스 품질과 공정성을 보장하기 위한 계정 제한
- 안전한 테넌트 격리 및 셀프 서비스 네임스페이스 초기화를 위한 네임스페이스 템플릿

Loft는 Kiosk 및 DevSpace의 유지 관리 담당자가 제공하는 상용 제품으로, 다음 기능을 추가합니다.

- 여러 클러스터의 스페이스에 대한 액세스 권한을 부여하기 위한 다중 클러스터 액세스
- 절전 모드는 비활성 기간 동안 스페이스의 배포를 축소합니다.
- GitHub와 같은 OIDC 인증 공급자를 사용한 Single Sign-On

소프트 멀티테넌시로 해결할 수 있는 세 가지 주요 사용 사례가 있습니다.

엔터프라이즈 설정

첫 번째는 "테넌트"가 직원, 계약업체이거나 조직의 승인을 받았다는 점에서 "테넌트"가 반신뢰되는 엔터프라이즈 설정입니다. 각 테넌트는 일반적으로 부서 또는 팀과 같은 관리 부서에 맞춰집니다.

이러한 유형의 설정에서 클러스터 관리자는 일반적으로 네임스페이스를 생성하고 정책을 관리할 책임이 있습니다. 또한 특정 개인이 네임스페이스를 감독하는 위임된 관리 모델을 구현하여 배포, 서비스, 포드, 작업 등과 같은 정책이 아닌 관련 객체에 대해 CRUD 작업을 수행할 수 있습니다.

컨테이너 런타임에서 제공하는 격리는이 설정 내에서 허용 가능하거나 포드 보안을 위한 추가 제어로 강화해야 할 수 있습니다. 더 엄격한 격리가 필요한 경우 다른 네임스페이스의 서비스 간 통신을 제한해야 할 수도 있습니다.

서비스형 Kubernetes

반대로 소프트 멀티테넌시는 Kubernetes를 서비스형(KaaS)으로 제공하려는 설정에서 사용할 수 있습니다. KaaS를 사용하면 애플리케이션이 PaaS 서비스 세트를 제공하는 컨트롤러 및 CRDs 모음과 함께 공유 클러스터에서 호스팅됩니다. 테넌트는 Kubernetes API 서버와 직접 상호 작용하며 정책이 적용되지 않는 객체에 대해 CRUD 작업을 수행할 수 있습니다. 테넌트가 자체 네임스페이스를 생성하고 관리하도록 허용될 수 있다는 셀프 서비스 요소도 있습니다. 이러한 유형의 환경에서 테넌트는 신뢰할 수 없는 코드를 실행하는 것으로 간주됩니다.

이러한 유형의 환경에서 테넌트를 격리하려면 포드 샌드박싱뿐만 아니라 엄격한 네트워크 정책을 구현해야 할 수 있습니다. 샌드박싱은 Firecracker와 같은 마이크로 VM 내부 또는 사용자 공간 커널에서

포드의 컨테이너를 실행하는 곳입니다. 현재 EKS Fargate를 사용하여 샌드박스 포드를 생성할 수 있습니다.

서비스형 소프트웨어(SaaS)

소프트 멀티테넌시의 최종 사용 사례는 Software-as-a-Service(SaaS) 설정입니다. 이 환경에서 각 테넌트는 클러스터 내에서 실행 중인 애플리케이션의 특정 인스턴스와 연결됩니다. 각 인스턴스에는 종종 자체 데이터가 있으며 일반적으로 Kubernetes RBAC와 독립적인 별도의 액세스 제어를 사용합니다.

다른 사용 사례와 달리 SaaS 설정의 테넌트는 Kubernetes API와 직접 인터페이스하지 않습니다. 대신 SaaS 애플리케이션은 Kubernetes API와 인터페이스하여 각 테넌트를 지원하는 데 필요한 객체를 생성할 책임이 있습니다.

Kubernetes 구문

이러한 각 인스턴스에서 다음 구문은 테넌트를 서로 격리하는 데 사용됩니다.

네임스페이스

네임스페이스는 소프트 멀티테넌시 구현의 기본입니다. 이를 통해 클러스터를 논리적 파티션으로 나눌 수 있습니다. 다중 테넌시를 구현하는 데 필요한 할당량, 네트워크 정책, 서비스 계정 및 기타 객체는 네임스페이스로 범위가 지정됩니다.

네트워크 정책

기본적으로 Kubernetes 클러스터의 모든 포드는 서로 통신할 수 있습니다. 이 동작은 네트워크 정책을 사용하여 변경할 수 있습니다.

네트워크 정책은 레이블 또는 IP 주소 범위를 사용하여 포드 간의 통신을 제한합니다. 테넌트 간에 엄격한 네트워크 격리가 필요한 다중 테넌트 환경에서는 포드 간의 통신을 거부하는 기본 규칙과 모든 포드가 이름 확인을 위해 DNS 서버를 쿼리하도록 허용하는 다른 규칙으로 시작하는 것이 좋습니다. 이를 통해 네임스페이스 내에서 통신을 허용하는 더 허용적인 규칙을 추가할 수 있습니다. 필요에 따라 이를 더욱 구체화할 수 있습니다.

> **Note**
>
> Amazon VPC CNI는 이제 Kubernetes 네트워크 정책을 지원하여 AWS에서 Kubernetes를 실행할 때 민감한 워크로드를 격리하고 무단 액세스로부터 보호할 수 있는 정책을 생성합니다.

즉, Amazon EKS 클러스터 내에서 네트워크 정책 API의 모든 기능을 사용할 수 있습니다. 이러한 수준의 세분화된 제어를 통해 최소 권한 원칙을 구현할 수 있으므로 승인된 포드만 서로 통신할 수 있습니다.

Important

네트워크 정책은 필요하지만 충분하지 않습니다. 네트워크 정책을 적용하려면 Calico 또는 Cilium과 같은 정책 엔진이 필요합니다.

역할 기반 액세스 제어(RBAC)

역할 및 역할 바인딩은 Kubernetes에서 역할 기반 액세스 제어(RBAC)를 적용하는 데 사용되는 Kubernetes 객체입니다. 역할에는 클러스터의 객체에 대해 수행할 수 있는 작업 목록이 포함됩니다. 역할 바인딩은 역할이 적용되는 개인 또는 그룹을 지정합니다. 엔터프라이즈 및 KaaS 설정에서 RBAC를 사용하여 선택한 그룹 또는 개인이 객체를 관리할 수 있습니다.

할당량

할당량은 클러스터에서 호스팅되는 워크로드에 대한 제한을 정의하는 데 사용됩니다. 할당량을 사용하면 포드가 사용할 수 있는 최대 CPU 및 메모리 양을 지정하거나 클러스터 또는 네임스페이스에 할당할 수 있는 리소스 수를 제한할 수 있습니다. 제한 범위를 사용하면 각 제한에 대한 최소값, 최대값 및 기본값을 선언할 수 있습니다.

공유 클러스터에서 리소스를 오버커밋하면 리소스를 최대화할 수 있으므로 종종 유용합니다. 그러나 클러스터에 대한 무제한 액세스로 인해 리소스 결핍이 발생하여 성능이 저하되고 애플리케이션 가용성이 손실될 수 있습니다. 포드의 요청이 너무 낮게 설정되고 실제 리소스 사용률이 노드 용량을 초과하는 경우 노드에 CPU 또는 메모리 압력이 발생하기 시작합니다. 이 경우 포드가 다시 시작되거나 노드에서 제거될 수 있습니다.

이를 방지하려면 클러스터에서 포드를 예약할 때 테넌트가 요청 및 제한을 지정하도록 강제하는 다중 테넌트 환경의 네임스페이스에 할당량을 적용할 계획을 세워야 합니다. 또한 포드가 사용할 수 있는 리소스의 양을 제한하여 잠재적인 서비스 거부를 완화합니다.

또한 할당량을 사용하여 테넌트의 지출에 맞게 클러스터의 리소스를 할당할 수 있습니다. 이는 KaaS 시나리오에서 특히 유용합니다.

포드 우선 순위 및 선점

포드 우선 순위 및 선점은 다른 포드에 비해 포드에 더 중요한 역할을 하려는 경우에 유용할 수 있습니다. 예를 들어 포드 우선 순위를 사용하면 고객 B보다 높은 우선 순위로 실행되도록 고객 A의 포드를 구성할 수 있습니다. 사용 가능한 용량이 충분하지 않으면 스케줄러는 고객 B의 우선 순위가 낮은 포드를 제거하여 고객 A의 우선 순위가 높은 포드를 수용할 수 있습니다. 이는 고객이 프리미엄 요금을 지불하려는 SaaS 환경에서 특히 유용할 수 있습니다.

> **Important**
>
> 포드 우선 순위는 우선 순위가 낮은 다른 포드에 원치 않는 영향을 미칠 수 있습니다. 예를 들어, 피해자 포드는 정상적으로 종료되지만 PodDisruptionBudget는 보장되지 않으므로 포드 쿼럼에 의존하는 우선 순위가 낮은 애플리케이션이 중단될 수 있습니다. 선점 제한을 참조하세요.

제어 완화

다중 테넌트 환경의 관리자로서 주요 관심사는 공격자가 기본 호스트에 액세스할 수 없도록 하는 것입니다. 이 위험을 완화하려면 다음 제어를 고려해야 합니다.

컨테이너용 샌드박스 실행 환경

샌드박싱은 각 컨테이너가 자체 격리된 가상 머신에서 실행되는 기법입니다. 포드 샌드박싱을 수행하는 기술에는 Firecracker와 Weave의 Firekube가 포함됩니다.

Firecracker를 EKS에 지원되는 런타임으로 만들기 위한 노력에 대한 자세한 내용은 https://threadreaderapp.com/thread/1238496944684597248.html 참조하십시오.

정책 에이전트(OPA) 및 게이트키퍼 열기

Gatekeeper는 OPA로 생성된 정책을 적용하는 Kubernetes 승인 컨트롤러입니다. OPA를 사용하면 별도의 인스턴스에 있는 테넌트에서 포드를 실행하거나 다른 테넌트보다 높은 우선 순위로 포드를 실행하는 정책을 생성할 수 있습니다. 일반적인 OPA 정책 모음은이 프로젝트의 GitHub 리포지토리에서 찾을 수 있습니다.

또한 CoreDNS에 대한 실험적 OPA 플러그인이 있어 OPA를 사용하여 CoreDNS에서 반환하는 레코드를 필터링/제어할 수 있습니다.

키버노

Kyverno는 정책을 Kubernetes 리소스로 사용하여 구성을 검증, 변경 및 생성할 수 있는 Kubernetes 네이티브 정책 엔진입니다. Kyverno는 검증을 위해 Kustomize 스타일 오버레이를 사용하고, 변형을 위해 JSON 패치 및 전략적 병합 패치를 지원하며, 유연한 트리거를 기반으로 네임스페이스에 리소스를 복제할 수 있습니다.

Kyverno를 사용하여 네임스페이스를 격리하고, 포드 보안 및 기타 모범 사례를 적용하고, 네트워크 정책과 같은 기본 구성을 생성할 수 있습니다. 이 프로젝트의 GitHub 리포지토리에는 몇 가지 예가 포함되어 있습니다. 다른 많은가 Kyverno 웹 사이트의 정책 라이브러리에 포함되어 있습니다.

테넌트 워크로드를 특정 노드로 격리

테넌트 워크로드가 특정 노드에서 실행되도록 제한하면 소프트 멀티테넌시 모델에서 격리를 강화하는 데 사용할 수 있습니다. 이 접근 방식을 사용하면 테넌트별 워크로드는 각 테넌트에 대해 프로비저닝된 노드에서만 실행됩니다. 이러한 격리를 달성하기 위해 네이티브 Kubernetes 속성(노드 선호도, 테인트 및 허용치)을 사용하여 포드 예약을 위한 특정 노드를 대상으로 지정하고 다른 테넌트의 포드가 테넌트별 노드에 예약되지 않도록 합니다.

1부 - 노드 선호도

Kubernetes 노드 선호도는 노드 레이블을 기반으로 스케줄링을 위해 노드를 대상으로 지정하는 데 사용됩니다. 노드 선호도 규칙을 사용하면 포드가 선택기 용어와 일치하는 특정 노드로 유인됩니다. 아래 포드 사양에서는 `requiredDuringSchedulingIgnoredDuringExecution` 노드 선호도가 각 포드에 적용됩니다. 결과적으로 포드는 키/값으로 레이블이 지정된 노드를 대상으로 합니다`node-restriction.kubernetes.io/tenant: tenants-x`.

```
...
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: node-restriction.kubernetes.io/tenant
            operator: In
            values:
            - tenants-x
...
```

이 노드 선호도를 사용하면 레이블이 예약 중에는 필요하지만 실행 중에는 필요하지 않습니다. 기본 노드의 레이블이 변경되면 해당 레이블 변경으로 인해 포드가 제거되지 않습니다. 그러나 향후 일정이 영향을 받을 수 있습니다.

> **Warning**
>
> 의 레이블 접두사는 Kubernetes에서 특별한 의미를 `node-restriction.kubernetes.io/` 갖습니다. EKS 클러스터에 대해 활성화된 NodeRestriction은 `kubelet`가이 접두사를 사용하여 레이블을 adding/removing/updating 방지합니다. 는 이러한 레이블을 수정할 수 `kubelet's credentials to update the node object or modify the system setup to pass these labels into `kubelet` 없으므로 공격자`kubelet`는를 사용할 수 없습니다. 이 접두사를 모든 포드에서 노드로 예약하는 데 사용하는 경우 공격자가 노드 레이블을 수정하여 노드에 다른 워크로드 세트를 유치하려는 시나리오를 방지합니다.

Example

노드 선호도 대신 노드 선택기를 사용할 수 있었습니다. 그러나 노드 선호도는 더 표현력이 뛰어나며 포드 예약 중에 더 많은 조건을 고려할 수 있습니다. 차이점과 고급 예약 선택 사항에 대한 자세한 내용은 고급 Kubernetes 포드에서 노드로의 예약에 대한이 CNCF 블로그 게시물을 참조하세요.

2부 - 테인트 및 허용치

포드를 노드로 끌어들이는 것은이 세 부분으로 구성된 접근 방식의 첫 번째 부분일 뿐입니다. 이 접근 방식이 작동하려면 포드가 승인되지 않은 노드로 포드가 예약되지 않도록 해야 합니다. 원치 않거나 승인되지 않은 포드를 제거하기 위해 Kubernetes는 노드 테인트를 사용합니다. 테인트는 포드가 예약되지 않도록 노드에 조건을 배치하는 데 사용됩니다. 아래 테인트는의 키-값 페어를 사용합니다`tenant: tenants-x`.

```
...
    taints:
      - key: tenant
        value: tenants-x
        effect: NoSchedule
...
```

위의 노드를 고려할 때 테인트를 허용하는 포드`taint`만 노드에서 예약할 수 있습니다. 승인된 포드를 노드에 예약하려면 아래 표시된 대로 각 포드 사양에 테인트`toleration`에 대한가 포함되어야 합니다.

```
...
  tolerations:
  - effect: NoSchedule
    key: tenant
    operator: Equal
    value: tenants-x
...
```

위와 같은 포드`toleration`는 최소한 특정 테인트 때문에 노드에서 예약이 중지되지 않습니다. 또한 Kubernetes는 테인트를 사용하여 노드 리소스 압력과 같은 특정 조건에서 포드 예약을 일시적으로 중지합니다. 노드 선호도, 테인트 및 허용치를 사용하면 원하는 포드를 특정 노드로 효과적으로 유치하고 원치 않는 포드를 제거할 수 있습니다.

> **Important**
>
> 특정 Kubernetes 포드는 모든 노드에서 실행해야 합니다. 이러한 포드의 예로는 컨테이너 네트워크 인터페이스(CNI) 및 kube-proxy 데몬 세트에서 시작된 포드가 있습니다. 이를 위해 이러한 포드의 사양에는 매우 허용적인 허용 오차가 포함되어 있어 다양한 테인트를 견딜 수 있습니다. 이러한 허용치를 변경하지 않도록 주의해야 합니다. 이러한 허용치를 변경하면 잘못된 클러스터 작업이 발생할 수 있습니다. 또한 OPA/Gatekeeper 및 Kyverno와 같은 정책 관리 도구를 사용하여 권한이 없는 포드가 이러한 허용 허용 오차를 사용하지 못하도록 하는 검증 정책을 작성할 수 있습니다.

3부 - 노드 선택을 위한 정책 기반 관리

CICD 파이프라인의 규칙 적용을 포함하여 포드 사양의 노드 선호도 및 허용 오차를 관리하는 데 사용할 수 있는 몇 가지 도구가 있습니다. 그러나 격리 적용은 Kubernetes 클러스터 수준에서도 수행해야 합니다. 이를 위해 정책 관리 도구를 사용하여 요청 페이로드를 기반으로 인바운드 Kubernetes API 서버 요청을 변경하여 위에서 언급한 각 노드 선호도 규칙 및 허용치를 적용할 수 있습니다.

예를 들어 테넌트-x 네임스페이스로 향하는 포드에는 테넌트-x 노드에 대한 예약을 허용하는 올바른 노드 선호도 및 허용 오차를 찍을 수 있습니다. Kubernetes Mutating Admission Webhook를 사용하여 구성된 정책 관리 도구를 사용하여 정책을 사용하여 인바운드 포드 사양을 변경할 수 있습니다. 변형은 원하는 예약을 허용하는 데 필요한 요소를 추가합니다. 노드 선호도를 추가하는 OPA/Gatekeeper 정책의 예는 다음과 같습니다.

```
apiVersion: mutations.gatekeeper.sh/v1alpha1
```

```
kind: Assign
metadata:
  name: mutator-add-nodeaffinity-pod
  annotations:
    aws-eks-best-practices/description: >-
      Adds Node affinity - https://kubernetes.io/docs/concepts/scheduling-eviction/
assign-pod-node/#node-affinity
spec:
  applyTo:
  - groups: [""]
    kinds: ["Pod"]
    versions: ["v1"]
  match:
    namespaces: ["tenants-x"]
  location:
 "spec.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms"
  parameters:
    assign:
      value:
        - matchExpressions:
          - key: "tenant"
            operator: In
            values:
            - "tenants-x"
```

위의 정책은 Kubernetes API 서버 요청에 적용되어 테넌트-x 네임스페이스에 포드를 적용합니다. 이 정책은 `requiredDuringSchedulingIgnoredDuringExecution` 노드 선호도 규칙을 추가하여 포드가 `tenant: tenants-x` 레이블이 있는 노드로 유인되도록 합니다.

아래에 표시된 두 번째 정책은 대상 네임스페이스 및 그룹, 종류 및 버전과 동일한 일치 기준을 사용하여 동일한 포드 사양에 허용치를 추가합니다.

```
apiVersion: mutations.gatekeeper.sh/v1alpha1
kind: Assign
metadata:
  name: mutator-add-toleration-pod
  annotations:
    aws-eks-best-practices/description: >-
      Adds toleration - https://kubernetes.io/docs/concepts/scheduling-eviction/taint-
and-toleration/
spec:
  applyTo:
  - groups: [""]
```

```
    kinds: ["Pod"]
    versions: ["v1"]
  match:
    namespaces: ["tenants-x"]
  location: "spec.tolerations"
  parameters:
    assign:
      value:
      - key: "tenant"
        operator: "Equal"
        value: "tenants-x"
        effect: "NoSchedule"
```

위의 정책은 포드에 고유합니다. 이는 정책 요소의 변형 `location` 요소 경로 때문입니다. 배포 및 작업 리소스와 같이 포드를 생성하는 리소스를 처리하기 위해 추가 정책을 작성할 수 있습니다. 나열된 정책 및 기타 예제는이 가이드의 동반 GitHub 프로젝트에서 확인할 수 있습니다.

이 두 변형의 결과는 포드가 원하는 노드로 끌리는 동시에 특정 노드 테인트에 의해 제거되지 않기 때문입니다. 이를 확인하기 위해 두 `kubectl` 호출의 출력 조각을 확인하여 로 레이블이 지정된 노드를 가져오`tenant=tenants-x`고 네`tenants-x`임스페이스에서 포드를 가져올 수 있습니다.

```
kubectl get nodes -l tenant=tenants-x
NAME
ip-10-0-11-255...
ip-10-0-28-81...
ip-10-0-43-107...

kubectl -n tenants-x get pods -owide
NAME                                 READY   STATUS    RESTARTS   AGE   IP
 NODE
tenant-test-deploy-58b895ff87-2q7xw   1/1     Running   0          13s   10.0.42.143
 ip-10-0-43-107...
tenant-test-deploy-58b895ff87-9b6hg   1/1     Running   0          13s   10.0.18.145
 ip-10-0-28-81...
tenant-test-deploy-58b895ff87-nxvw5   1/1     Running   0          13s   10.0.30.117
 ip-10-0-28-81...
tenant-test-deploy-58b895ff87-vw796   1/1     Running   0          13s   10.0.3.113
 ip-10-0-11-255...
tenant-test-pod                       1/1     Running   0          13s   10.0.35.83
 ip-10-0-43-107...
```

위 출력에서 볼 수 있듯이 모든 포드는 레이블이 지정된 노드에 예약됩니다`tenant=tenants-x`. 간단히 말해서 포드는 원하는 노드에서만 실행되고 다른 포드(필요한 선호도 및 허용치 없음)는 실행되지 않습니다. 테넌트 워크로드는 효과적으로 격리됩니다.

다음은 변형 포드 사양의 예입니다.

```
apiVersion: v1
kind: Pod
metadata:
  name: tenant-test-pod
  namespace: tenants-x
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: tenant
            operator: In
            values:
            - tenants-x
...
  tolerations:
  - effect: NoSchedule
    key: tenant
    operator: Equal
    value: tenants-x
...
```

Important

변경 및 검증 승인 웹후크를 사용하여 Kubernetes API 서버 요청 흐름에 통합된 정책 관리 도구는 지정된 기간 내에 API 서버의 요청에 응답하도록 설계되었습니다. 일반적으로 3초 이하입니다. 웹후크 호출이 구성된 시간 내에 응답을 반환하지 못하면 인바운드 API 서버 요청의 변형 및/또는 검증이 발생하거나 발생하지 않을 수 있습니다. 이 동작은 승인 웹후크 구성이 Fail Open 또는 Fail Close로 설정되어 있는지 여부에 따라 달라집니다.

위 예제에서는 OPA/Gatekeeper에 대해 작성된 정책을 사용했습니다. 그러나 노드 선택 사용 사례를 처리하는 다른 정책 관리 도구도 있습니다. 예를 들어이 Kyverno 정책을 사용하여 노드 선호도 변형을 처리할 수 있습니다.

> **Note**
>
> 올바르게 작동하는 경우 변경 정책은 인바운드 API 서버 요청 페이로드에 원하는 변경 사항에 영향을 미칩니다. 그러나 변경 사항이 지속되기 전에 원하는 변경 사항이 발생하는지 확인하기 위해 검증 정책도 포함되어야 합니다. 이는 tenant-to-node 격리에 이러한 정책을 사용할 때 특히 중요합니다. 또한 감사 정책을 포함하여 클러스터에 원치 않는 구성이 있는지 정기적으로 확인하는 것이 좋습니다.

참조

- k-rail 특정 정책의 적용을 통해 다중 테넌트 환경을 보호하는 데 도움이 되도록 설계되었습니다.
- Amazon EKS를 사용하는 MultiTenant SaaS 애플리케이션에 대한 보안 사례

하드 멀티테넌시

각 테넌트에 대해 별도의 클러스터를 프로비저닝하여 하드 멀티테넌시를 구현할 수 있습니다. 이는 테넌트 간에 매우 강력한 격리를 제공하지만 몇 가지 단점이 있습니다.

먼저 테넌트가 많은 경우이 접근 방식은 빠르게 비용이 많이 들 수 있습니다. 각 클러스터의 컨트롤 플레인 비용을 지불해야 할 뿐만 아니라 클러스터 간에 컴퓨팅 리소스를 공유할 수 없습니다. 이로 인해 결국 클러스터의 하위 집합이 제대로 사용되지 않는 반면 다른 하위 집합은 과도하게 사용되는 조각화가 발생합니다.

둘째, 이러한 모든 클러스터를 관리하기 위해 특수 도구를 구매하거나 구축해야 할 수 있습니다. 시간이 지나면 수백 또는 수천 개의 클러스터를 관리하는 것이 너무 불안정해질 수 있습니다.

마지막으로, 네임스페이스 생성에 비해 테넌트당 클러스터 생성 속도가 느려집니다. 그러나 엄격한 테넌시 접근 방식은 규제가 엄격한 산업 또는 강력한 격리가 필요한 SaaS 환경에서 필요할 수 있습니다.

향후 지침

Kubernetes 커뮤니티는 소프트 멀티테넌시의 현재 단점과 하드 멀티테넌시의 문제를 인식했습니다. 다중 테넌시 특수 관심 그룹(SIG)은 계층적 네임스페이스 컨트롤러(HNC) 및 가상 클러스터를 비롯한 여러 트랜스포메이션 프로젝트를 통해 이러한 단점을 해결하려고 시도하고 있습니다.

HNC 제안(KEP)은 테넌트 관리자가 하위 네임스페이스를 생성할 수 있는 기능과 함께 [정책] 객체 상속이 있는 네임스페이스 간에 상위-하위 관계를 생성하는 방법을 설명합니다.

가상 클러스터 제안에서는 클러스터 내의 각 테넌트('Kubernetes on Kubernetes'라고도 함)에 대해 API 서버, 컨트롤러 관리자 및 스케줄러를 포함하여 컨트롤 플레인 서비스의 별도의 인스턴스를 생성하는 메커니즘을 설명합니다.

다중 테넌시 벤치마크 제안은 격리 및 세분화를 위한 네임스페이스를 사용하여 클러스터를 공유하기 위한 지침과 지침 준수를 검증하기 위한 명령줄 도구 kubectl-mtb를 제공합니다.

다중 클러스터 관리 도구 및 리소스

- 반자이 클라우드
- Kommander
- 렌즈
- 니마타
- Rafay
- 랜처
- 위브 플럭스

감사 및 로깅

[감사] 로그를 수집하고 분석하는 것은 다양한 이유로 유용합니다. 로그는 근본 원인 분석 및 속성, 즉 특정 사용자에 대한 변경 사항을 설명하는 데 도움이 될 수 있습니다. 충분한 로그가 수집되면 이상 동작을 감지하는 데에도 사용할 수 있습니다. EKS에서는 감사 로그가 Amazon Cloudwatch Logs로 전송됩니다. EKS에 대한 감사 정책은 다음과 같습니다.

```
apiVersion: audit.k8s.io/v1beta1
kind: Policy
rules:
  # Log full request and response for changes to aws-auth ConfigMap in kube-system
 namespace
  - level: RequestResponse
    namespaces: ["kube-system"]
    verbs: ["update", "patch", "delete"]
    resources:
      - group: "" # core
        resources: ["configmaps"]
        resourceNames: ["aws-auth"]
    omitStages:
      - "RequestReceived"
```

```
  # Do not log watch operations performed by kube-proxy on endpoints and services
  - level: None
    users: ["system:kube-proxy"]
    verbs: ["watch"]
    resources:
      - group: "" # core
        resources: ["endpoints", "services", "services/status"]
  # Do not log get operations performed by kubelet on nodes and their statuses
  - level: None
    users: ["kubelet"] # legacy kubelet identity
    verbs: ["get"]
    resources:
      - group: "" # core
        resources: ["nodes", "nodes/status"]
  # Do not log get operations performed by the system:nodes group on nodes and their
 statuses
  - level: None
    userGroups: ["system:nodes"]
    verbs: ["get"]
    resources:
      - group: "" # core
        resources: ["nodes", "nodes/status"]
  # Do not log get and update operations performed by controller manager, scheduler,
 and endpoint-controller on endpoints in kube-system namespace
  - level: None
    users:
      - system:kube-controller-manager
      - system:kube-scheduler
      - system:serviceaccount:kube-system:endpoint-controller
    verbs: ["get", "update"]
    namespaces: ["kube-system"]
    resources:
      - group: "" # core
        resources: ["endpoints"]
  # Do not log get operations performed by apiserver on namespaces and their statuses/
finalizations
  - level: None
    users: ["system:apiserver"]
    verbs: ["get"]
    resources:
      - group: "" # core
        resources: ["namespaces", "namespaces/status", "namespaces/finalize"]
  # Do not log get and list operations performed by controller manager on
 metrics.k8s.io resources
```

```
  - level: None
    users:
      - system:kube-controller-manager
    verbs: ["get", "list"]
    resources:
      - group: "metrics.k8s.io"
  # Do not log access to health, version, and swagger non-resource URLs
  - level: None
    nonResourceURLs:
      - /healthz*
      - /version
      - /swagger*
  # Do not log events resources
  - level: None
    resources:
      - group: "" # core
        resources: ["events"]
  # Log request for updates/patches to nodes and pods statuses by kubelet and node
 problem detector
  - level: Request
    users: ["kubelet", "system:node-problem-detector", "system:serviceaccount:kube-
system:node-problem-detector"]
    verbs: ["update", "patch"]
    resources:
      - group: "" # core
        resources: ["nodes/status", "pods/status"]
    omitStages:
      - "RequestReceived"
  # Log request for updates/patches to nodes and pods statuses by system:nodes group
  - level: Request
    userGroups: ["system:nodes"]
    verbs: ["update", "patch"]
    resources:
      - group: "" # core
        resources: ["nodes/status", "pods/status"]
    omitStages:
      - "RequestReceived"
  # Log delete collection requests by namespace-controller in kube-system namespace
  - level: Request
    users: ["system:serviceaccount:kube-system:namespace-controller"]
    verbs: ["deletecollection"]
    omitStages:
      - "RequestReceived"
  # Log metadata for secrets, configmaps, and tokenreviews to protect sensitive data
```

```
  - level: Metadata
    resources:
      - group: "" # core
        resources: ["secrets", "configmaps"]
      - group: authentication.k8s.io
        resources: ["tokenreviews"]
    omitStages:
      - "RequestReceived"
  # Log requests for serviceaccounts/token resources
  - level: Request
    resources:
      - group: "" # core
        resources: ["serviceaccounts/token"]
  # Log get, list, and watch requests for various resource groups
  - level: Request
    verbs: ["get", "list", "watch"]
    resources:
      - group: "" # core
      - group: "admissionregistration.k8s.io"
      - group: "apiextensions.k8s.io"
      - group: "apiregistration.k8s.io"
      - group: "apps"
      - group: "authentication.k8s.io"
      - group: "authorization.k8s.io"
      - group: "autoscaling"
      - group: "batch"
      - group: "certificates.k8s.io"
      - group: "extensions"
      - group: "metrics.k8s.io"
      - group: "networking.k8s.io"
      - group: "policy"
      - group: "rbac.authorization.k8s.io"
      - group: "scheduling.k8s.io"
      - group: "settings.k8s.io"
      - group: "storage.k8s.io"
    omitStages:
      - "RequestReceived"
  # Default logging level for known APIs to log request and response
  - level: RequestResponse
    resources:
      - group: "" # core
      - group: "admissionregistration.k8s.io"
      - group: "apiextensions.k8s.io"
      - group: "apiregistration.k8s.io"
```

```
      - group: "apps"
      - group: "authentication.k8s.io"
      - group: "authorization.k8s.io"
      - group: "autoscaling"
      - group: "batch"
      - group: "certificates.k8s.io"
      - group: "extensions"
      - group: "metrics.k8s.io"
      - group: "networking.k8s.io"
      - group: "policy"
      - group: "rbac.authorization.k8s.io"
      - group: "scheduling.k8s.io"
      - group: "settings.k8s.io"
      - group: "storage.k8s.io"
    omitStages:
      - "RequestReceived"
  # Default logging level for all other requests to log metadata only
  - level: Metadata
    omitStages:
      - "RequestReceived"
```

권장 사항

감사 로그 활성화

감사 로그는 EKS에서 관리하는 EKS 관리형 Kubernetes 컨트롤 플레인 로그의 일부입니다. Kubernetes API 서버, 컨트롤러 관리자 및 스케줄러에 대한 로그가 포함된 컨트롤 플레인 로그를 활성화/비활성화하는 지침은 감사 로그와 함께 https://docs.aws.amazon.com/eks/latest/userguide/control-plane-logs.html#enabling-control-plane-log-export 확인할 수 있습니다.

> **Note**
>
> 컨트롤 플레인 로깅을 활성화하면 CloudWatch에 로그를 저장하는 데 비용이 발생합니다. 이로 인해 지속적인 보안 비용에 대한 더 광범위한 문제가 발생합니다. 궁극적으로 이러한 비용을 재정적 손실, 평판 손상 등과 같은 보안 침해 비용과 비교해야 합니다. 이 가이드의 권장 사항 중 일부만 구현하여 환경을 적절하게 보호할 수 있습니다.

> **Warning**
>
> CloudWatch Logs 항목의 최대 크기는 1MB이고 최대 Kubernetes API 요청 크기는 1.5MiB입니다. 1MB를 초과하는 로그 항목은 잘리거나 요청 메타데이터만 포함됩니다.

감사 메타데이터 활용

Kubernetes 감사 로그에는 요청이 승인되었는지 여부와 결정 이유를 나타내는 두 개의 주석`authorization.k8s.io/decision`이 포함되어 있습니다`authorization.k8s.io/reason`. 이러한 속성을 사용하여 특정 API 호출이 허용된 이유를 확인합니다.

의심스러운 이벤트에 대한 경보 생성

경보를 생성하여 금지된 응답 403개와 승인되지 않은 응답 401개가 증가한 위치를 자동으로 경고한 다음 , `host` `sourceIPs`및와 같은 속성을 사용하여 해당 요청의 출처`k8s_user.username`를 확인합니다.

Log Insights를 사용하여 로그 분석

CloudWatch Log Insights를 사용하여 역할, RoleBindings, ClusterRoles 및 ClusterRoleBindings와 같은 RBAC 객체에 대한 변경 사항을 모니터링합니다. 다음은 몇 가지 샘플 쿼리입니다.

`aws-auth` ConfigMap에 대한 업데이트를 나열합니다.

```
fields @timestamp, @message
| filter @logStream like "kube-apiserver-audit"
| filter verb in ["update", "patch"]
| filter objectRef.resource = "configmaps" and objectRef.name = "aws-auth" and
 objectRef.namespace = "kube-system"
| sort @timestamp desc
```

검증 웹후크에 대한 새 생성 또는 변경 사항을 나열합니다.

```
fields @timestamp, @message
| filter @logStream like "kube-apiserver-audit"
| filter verb in ["create", "update", "patch"] and responseStatus.code = 201
| filter objectRef.resource = "validatingwebhookconfigurations"
| sort @timestamp desc
```

역할에 대한 생성, 업데이트, 삭제 작업을 나열합니다.

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="roles" and verb in ["create", "update", "patch", "delete"]
```

RoleBindings.

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="rolebindings" and verb in ["create", "update", "patch",
 "delete"]
```

ClusterRoles에 대한 생성, 업데이트, 삭제 작업을 나열합니다.

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="clusterroles" and verb in ["create", "update", "patch",
 "delete"]
```

ClusterRoleBindings.

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="clusterrolebindings" and verb in ["create", "update",
 "patch", "delete"]
```

Secrets에 대한 무단 읽기 작업을 도표화합니다.

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="secrets" and verb in ["get", "watch", "list"] and
 responseStatus.code="401"
| stats count() by bin(1m)
```

실패한 익명 요청 목록:

```
fields @timestamp, @message, sourceIPs.0
| sort @timestamp desc
| limit 100
| filter user.username="system:anonymous" and responseStatus.code in ["401", "403"]
```

CloudTrail 로그 감사

서비스 계정에 대한 IAM 역할(IRSA)을 사용하는 포드에서 호출하는 AWS APIs는 서비스 계정의 이름과 함께 CloudTrail에 자동으로 로깅됩니다. API 호출 권한이 명시적으로 부여되지 않은 서비스 계정의 이름이 로그에 표시되는 경우 IAM 역할의 신뢰 정책이 잘못 구성되었음을 나타낼 수 있습니다. 일반적으로 Cloudtrail은 특정 IAM 보안 주체에 대한 AWS API 호출을 구독하는 좋은 방법입니다.

CloudTrail Insights를 사용하여 의심스러운 활동 파악

CloudTrail 인사이트는 CloudTrail 추적의 쓰기 관리 이벤트를 자동으로 분석하고 비정상적인 활동을 알려줍니다. 이를 통해 IRSA를 사용하여 IAM 역할을 수임하는 포드를 포함하여 AWS 계정의 쓰기 APIs에서 호출 볼륨이 증가하는 시기를 식별할 수 있습니다. 자세한 내용은 CloudTrail Insights 발표: 비정상적인 API 활동 식별 및 대응을 참조하세요.

추가 리소스

로그 볼륨이 증가하면 Log Insights 또는 다른 로그 분석 도구를 사용하여 로그를 구문 분석하고 필터링하는 것이 효과적이지 않을 수 있습니다. 대안으로 Sysdig Falco 및 ekscloudwatch 실행을 고려할 수 있습니다. Falco는 감사 로그를 분석하고 장기간 이상 또는 남용에 플래그를 지정합니다. ekscloudwatch 프로젝트는 분석을 위해 CloudWatch에서 Falco로 감사 로그 이벤트를 전달합니다. Falco는 사용자 고유의 추가 기능과 함께 기본 감사 규칙 세트를 제공합니다.

그러나 또 다른 옵션은 감사 로그를 S3에 저장하고 SageMaker Random Cut Forest 알고리즘을 사용하여 추가 조사가 필요한 비정상적인 동작을 수행하는 것입니다.

도구 및 리소스

다음과 같은 상용 및 오픈 소스 프로젝트를 사용하여 클러스터가 설정된 모범 사례에 부합하는지 평가할 수 있습니다.

- Amazon EKS 보안 침수 워크숍 - 탐지 제어
- kubeaudit
- kube-scan Kubernetes Common Configuration Scoring System 프레임워크에 따라 클러스터에서 실행되는 워크로드에 위험 점수를 할당합니다.

- kubesec.io
- 극성
- 스타보드
- Snyk
- Kubescape Kubescape는 클러스터, YAML 파일 및 차트 Helm을 스캔하는 오픈 소스 kubernetes 보안 도구입니다. 여러 프레임워크(NSA-CISA 및 MITRE ATT&CK® 포함)에 따라 잘못된 구성을 감지합니다.

네트워크 보안

네트워크 보안에는 여러 패싯이 있습니다. 첫 번째는 서비스 간 네트워크 트래픽 흐름을 제한하는 규칙의 적용을 포함합니다. 두 번째에는 전송 중 트래픽 암호화가 포함됩니다. EKS에서 이러한 보안 조치를 구현하는 메커니즘은 다양하지만 종종 다음 항목을 포함합니다.

트래픽 제어

- 네트워크 정책
- 보안 그룹

네트워크 암호화

- 서비스 메시
- 컨테이너 네트워크 인터페이스(CNIs)
- 수신 컨트롤러 및 로드 밸런서
- Nitro 인스턴스
- cert-manager를 사용하는 ACM Private CA

네트워크 정책

Kubernetes 클러스터 내에서 모든 포드 간 통신은 기본적으로 허용됩니다. 이러한 유연성은 실험을 촉진하는 데 도움이 될 수 있지만 안전한 것으로 간주되지 않습니다. Kubernetes 네트워크 정책은 포드 간(종종 동부/서부 트래픽이라고 함) 및 포드와 외부 서비스 간의 네트워크 트래픽을 제한하는 메커니즘을 제공합니다. Kubernetes 네트워크 정책은 OSI 모델의 계층 3 및 4에서 작동합니다. 네트워크 정

책은 포드, 네임스페이스 선택기 및 레이블을 사용하여 소스 및 대상 포드를 식별하지만 IP 주소, 포트 번호, 프로토콜 또는 이들의 조합을 포함할 수도 있습니다. 네트워크 정책은 종종 수신 및 송신 규칙이라고 하는 포드에 대한 인바운드 또는 아웃바운드 연결 모두에 적용할 수 있습니다.

Amazon VPC CNI 플러그인의 기본 네트워크 정책 지원을 통해 네트워크 정책을 구현하여 kubernetes 클러스터에서 네트워크 트래픽을 보호할 수 있습니다. 이는 업스트림 Kubernetes 네트워크 정책 API와 통합되어 호환성과 Kubernetes 표준 준수를 보장합니다. 업스트림 API에서 지원하는 다양한 식별자를 사용하여 정책을 정의할 수 있습니다. 기본적으로 모든 수신 및 송신 트래픽은 포드에 허용됩니다. policyType Ingress가 있는 네트워크 정책을 지정하면 포드에 허용되는 연결만 포드 노드의 연결과 수신 규칙에서 허용하는 연결입니다. 송신 규칙에도 동일하게 적용됩니다. 여러 규칙이 정의된 경우 결정을 내릴 때 모든 규칙의 조합이 고려됩니다. 따라서 평가 순서는 정책 결과에 영향을 주지 않습니다.

> **Important**
>
> EKS 클러스터를 처음 프로비저닝할 때 VPC CNI 네트워크 정책 기능은 기본적으로 활성화되어 있지 않습니다. 지원되는 VPC CNI 추가 기능 버전을 배포했는지 확인하고 vpc-cni 추가 기능`true`에서 `ENABLE_NETWORK_POLICY` 플래그를 로 설정하여 이를 활성화합니다. 자세한 지침은 Amazon EKS 사용 설명서를 참조하세요.

권장 사항

네트워크 정책 시작하기 - 최소 권한 원칙 준수

기본 거부 정책 생성

RBAC 정책과 마찬가지로 네트워크 정책에 대한 최소 권한 액세스 원칙을 따르는 것이 좋습니다. 네임스페이스에서를 사용하여 모든 인바운드 및 아웃바운드 트래픽을 제한하는 모든 거부 정책을 생성하는 것으로 시작합니다.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
  namespace: default
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

```
  - Egress
```

default-deny

		Target		Source				
#	Policy name	Namespace	Pods	Namespace	Pods	Subnet	Ports	
1	default-deny	default	Any	Any	Any	Any	Any	

> **Note**
>
> 위 이미지는 Tufin의 네트워크 정책 뷰어에 의해 생성되었습니다.

DNS 쿼리를 허용하는 규칙 생성

기본 거부 모든 규칙이 적용되면 포드가 이름 확인을 위해 CoreDNS를 쿼리하도록 허용하는 규칙과 같은 추가 규칙에 대한 계층화를 시작할 수 있습니다.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-dns-access
  namespace: default
spec:
  podSelector:
    matchLabels: {}
  policyTypes:
  - Egress
  egress:
  - to:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: kube-system
      podSelector:
        matchLabels:
          k8s-app: kube-dns
    ports:
    - protocol: UDP
      port: 53
```

allow-dns-access

		Target		Destination				
#	Policy name	Namespace	Pods	Namespace	Pods	Subnet	Ports	
1	allow-dns-access	default	Any	name: kube-system	Any		UDP: 53	

네임스페이스/포드 간의 트래픽 흐름을 선택적으로 허용하는 규칙을 점진적으로 추가

애플리케이션 요구 사항을 이해하고 필요에 따라 세분화된 수신 및 송신 규칙을 생성합니다. 아래 예제에서는 포트 80의 수신 트래픽을 app-one에서 로 제한하는 방법을 보여줍니다client-one. 이렇게 하면 공격 표면을 최소화하고 무단 액세스 위험을 줄일 수 있습니다.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-ingress-app-one
  namespace: default
spec:
  podSelector:
    matchLabels:
      k8s-app: app-one
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          k8s-app: client-one
    ports:
    - protocol: TCP
      port: 80
```

allow-ingress-app-one

		Target		Source				
#	Policy name	Namespace	Pods	Namespace	Pods	Subnet	Ports	
1	allow-ingress-app-one	default	k8s-app: app-one	default	k8s-app: client-one		TCP: 80	

네트워크 정책 적용 모니터링

- 네트워크 정책 편집기 사용

  - 네트워크 정책 편집기는 네트워크 흐름 로그에서 시각화, 보안 점수, 자동 생성에 도움이 됩니다.
  - 대화형 방식으로 네트워크 정책 구축
- 감사 로그
  - EKS 클러스터의 감사 로그를 정기적으로 검토
  - 감사 로그는 네트워크 정책 변경을 포함하여 클러스터에서 수행된 작업에 대한 풍부한 정보를 제공합니다.
  - 이 정보를 사용하여 시간 경과에 따른 네트워크 정책의 변경 사항을 추적하고 무단 또는 예상치 못한 변경 사항을 감지합니다.
- 자동 테스트
  - 프로덕션 환경을 미러링하는 테스트 환경을 생성하여 자동 테스트를 구현하고 네트워크 정책을 위반하려고 시도하는 워크로드를 주기적으로 배포합니다.
- 지표 모니터링
  - 가 에이전트 상태 및 sdk 오류를 모니터링할 수 있도록 VPC CNI 노드 에이전트에서 prometheus 지표를 스크레이프하도록 관찰성 에이전트를 구성합니다.
- 네트워크 정책을 정기적으로 감사
  - 네트워크 정책을 정기적으로 감사하여 현재 애플리케이션 요구 사항을 충족하는지 확인합니다. 애플리케이션이 발전함에 따라 감사를 통해 중복 수신, 송신 규칙을 제거하고 애플리케이션에 과도한 권한이 없는지 확인할 수 있습니다.
- OPA(Open Policy Agent)를 사용하여 네트워크 정책이 존재하는지 확인
  - 애플리케이션 포드를 온보딩하기 전에 네트워크 정책이 항상 존재하는지 확인하려면 아래 표시된 것과 같은 OPA 정책을 사용합니다. 이 정책은 해당 네트워크 정책이 없는 `k8s-app: sample-app` 경우 레이블이 있는 k8s 포드 온보딩을 거부합니다.

```
package kubernetes.admission
import data.kubernetes.networkpolicies

deny[msg] {
    input.request.kind.kind == "Pod"
    pod_label_value := {v["k8s-app"] | v := input.request.object.metadata.labels}
    contains_label(pod_label_value, "sample-app")
    np_label_value := {v["k8s-app"] | v :=
 networkpolicies[_].spec.podSelector.matchLabels}
    not contains_label(np_label_value, "sample-app")
    msg:= sprintf("The Pod %v could not be created because it is missing an associated
 Network Policy.", [input.request.object.metadata.name])
```

```
}
contains_label(arr, val) {
    arr[_] == val
}
```

문제 해결

vpc-network-policy-controller, node-agent 로그 모니터링

EKS 컨트롤 플레인 컨트롤러 관리자 로그를 활성화하여 네트워크 정책 기능을 진단합니다. 컨트롤 플레인 로그를 CloudWatch 로그 그룹으로 스트리밍하고 CloudWatch Log 인사이트를 사용하여 고급 쿼리를 수행할 수 있습니다. 로그에서 네트워크 정책으로 확인된 포드 엔드포인트 객체, 정책의 조정 상태를 확인하고 정책이 예상대로 작동하는지 디버깅할 수 있습니다.

또한 Amazon VPC CNI를 사용하면 정책 적용 로그를 수집하고 EKS 작업자 노드에서 Amazon Cloudwatch로 내보낼 수 있습니다. 활성화되면 CloudWatch Container Insights를 활용하여 네트워크 정책과 관련된 사용량에 대한 인사이트를 제공할 수 있습니다.

또한 Amazon VPC CNI는 노드의 eBPF 프로그램과 상호 작용할 수 있는 인터페이스를 제공하는 SDK를 제공합니다. SDK는가 노드에 배포될 때 설치aws-node됩니다. 노드의 /opt/cni/bin 디렉터리 아래에 설치된 SDK 바이너리를 찾을 수 있습니다. 시작 시 SDK는 eBPF 프로그램 및 맵 검사와 같은 기본 기능에 대한 지원을 제공합니다.

```
sudo /opt/cni/bin/aws-eks-na-cli ebpf progs
```

네트워크 트래픽 메타데이터 로깅

AWS VPC 흐름 로그는 수락/중단된 패킷과 함께 소스 및 대상 IP 주소 및 포트와 같이 VPC를 통해 흐르는 트래픽에 대한 메타데이터를 캡처합니다. 이 정보를 분석하여 포드를 포함한 VPC 내 리소스 간에 의심스럽거나 비정상적인 활동을 찾을 수 있습니다. 그러나 포드의 IP 주소는 교체될 때 자주 변경되므로 흐름 로그만으로는 충분하지 않을 수 있습니다. Calico Enterprise는 포드 레이블 및 기타 메타데이터를 사용하여 흐름 로그를 확장하므로 포드 간의 트래픽 흐름을 더 쉽게 확인할 수 있습니다.

보안 그룹

EKS는 AWS VPC 보안 그룹(SGs)을 사용하여 Kubernetes 컨트롤 플레인과 클러스터의 작업자 노드 간의 트래픽을 제어합니다. 보안 그룹은 작업자 노드, 기타 VPC 리소스 및 외부 IP 주소 간의 트래픽을 제어하는 데에도 사용됩니다. EKS 클러스터(Kubernetes 버전 1.14-eks.3 이상)를 프로비저닝하면 클러스터 보안 그룹이 자동으로 생성됩니다. 이 보안 그룹은 EKS 컨트롤 플레인과 관리형 노드 그룹의

노드 간에 원활한 통신을 허용합니다. 간소화를 위해 비관리형 노드 그룹을 포함한 모든 노드 그룹에 클러스터 SG를 추가하는 것이 좋습니다.

Kubernetes 버전 1.14 및 EKS 버전 eks.3 이전에는 EKS 컨트롤 플레인 및 노드 그룹에 대해 별도의 보안 그룹이 구성되었습니다. 컨트롤 플레인 및 노드 그룹 보안 그룹에 대한 최소 및 권장 규칙은 https://docs.aws.amazon.com/eks/latest/userguide/sec-group-reqs.html 확인할 수 있습니다. 컨트롤 플레인 보안 그룹의 최소 규칙은 작업자 노드 SG에서 인바운드로 포트 443을 허용합니다. 이 규칙은 kubelet이 Kubernetes API 서버와 통신할 수 있도록 허용합니다. 작업자 노드 SG로의 아웃바운드 트래픽을 위한 포트 10250도 포함되어 있습니다. 10250은 kubelet이 수신하는 포트입니다. 마찬가지로 최소 노드 그룹 규칙은 컨트롤 플레인 SG에서 포트 10250 인바운드를 허용하고 컨트롤 플레인 SG로 443 아웃바운드를 허용합니다. 마지막으로 노드 그룹 내의 노드 간 원활한 통신을 허용하는 규칙이 있습니다.

클러스터 내에서 실행되는 서비스와 RDS 데이터베이스와 같은 클러스터 외부에서 실행되는 서비스 간의 통신을 제어해야 하는 경우 포드의 보안 그룹을 고려하세요. 포드의 보안 그룹을 사용하면 포드 컬렉션에 기존 보안 그룹을 할당할 수 있습니다.

> **Warning**
>
> 포드를 생성하기 전에 존재하지 않는 보안 그룹을 참조하면 포드가 예약되지 않습니다.

`SecurityGroupPolicy` 객체를 생성하고 `PodSelector` 또는를 지정하여 보안 그룹에 할당되는 포드를 제어할 수 있습니다`ServiceAccountSelector`. 선택기를 로 설정`{}`하면에서 참조되는 SGs가 네임스페이스의 `SecurityGroupPolicy` 모든 포드 또는 네임스페이스의 모든 서비스 계정에 할당됩니다. 포드에 대한 보안 그룹을 구현하기 전에 모든 고려 사항을 숙지해야 합니다.

> **Important**
>
> 포드에 SGs를 사용하는 경우 클러스터 보안 그룹에 대한 포트 53 아웃바운드를 허용하는 SGs를 생성해야 합니다. 마찬가지로 포드 보안 그룹의 포트 53 인바운드 트래픽을 수락하도록 클러스터 보안 그룹을 업데이트해야 합니다.

> **Important**
>
> 포드에 보안 그룹을 사용할 때는 보안 그룹에 대한 제한이 계속 적용되므로 신중하게 사용하십시오.

> **Important**
>
> 포드에 대해 구성된 모든 프로브에 대해 클러스터 보안 그룹(kubelet)의 인바운드 트래픽에 대한 규칙을 생성해야 합니다.

> **Important**
>
> 포드의 보안 그룹은 EC2 인스턴스의 ENI 밀도를 높이기 위해 생성된 ENI 트렁킹이라는 기능을 사용합니다. 포드가 SG에 할당되면 VPC 컨트롤러는 노드 그룹의 브랜치 ENI를 포드와 연결합니다. 포드가 예약된 시점에 노드 그룹에서 사용할 수 있는 브랜치 ENIs가 충분하지 않으면 포드는 보류 상태로 유지됩니다. 인스턴스가 지원할 수 있는 브랜ENIs 수는 인스턴스 유형/계열에 따라 다릅니다. 자세한 내용은 https://docs.aws.amazon.com/eks/latest/userguide/security-groups-for-pods.html#supported-instance-types 참조하십시오.

포드의 보안 그룹은 정책 데몬의 오버헤드 없이 클러스터 내부 및 외부의 네트워크 트래픽을 제어하는 AWS 네이티브 방법을 제공하지만 다른 옵션을 사용할 수 있습니다. 예를 들어 Cilium 정책 엔진을 사용하면 네트워크 정책에서 DNS 이름을 참조할 수 있습니다. Calico Enterprise에는 네트워크 정책을 AWS 보안 그룹에 매핑하는 옵션이 포함되어 있습니다. Istio와 같은 서비스 메시를 구현한 경우 송신 게이트웨이를 사용하여 네트워크 송신을 정규화된 특정 도메인 또는 IP 주소로 제한할 수 있습니다. 이 옵션에 대한 자세한 내용은 Istio에서 송신 트래픽 제어에 대한 세 부분 시리즈를 참조하십시오.

포드에 네트워크 정책 대 보안 그룹을 사용해야 하는 경우는 언제입니까?

Kubernetes 네트워크 정책을 사용해야 하는 경우

- pod-to-pod 트래픽 제어
  - 클러스터 내의 포드 간 네트워크 트래픽을 제어하는 데 적합(동서 트래픽)
- IP 주소 또는 포트 수준에서 트래픽 제어(OSI 계층 3 또는 4)

포드에 AWS 보안 그룹을 사용해야 하는 경우(SGP)

- 기존 AWS 구성 활용
  - AWS 서비스에 대한 액세스를 관리하는 복잡한 EC2 보안 그룹 세트가 이미 있고 EC2 인스턴스에서 EKS로 애플리케이션을 마이그레이션하는 경우 SGPs를 사용하면 보안 그룹 리소스를 재사용하고 포드에 적용할 수 있습니다.

- AWS 서비스에 대한 액세스 제어
  - EKS 클러스터 내에서 실행되는 애플리케이션은 다른 AWS 서비스(RDS 데이터베이스)와 통신하려는 경우 SGPs 효율적인 메커니즘으로 사용하여 포드에서 AWS 서비스로의 트래픽을 제어합니다.
- 포드 및 노드 트래픽 격리
  - 포드 트래픽을 나머지 노드 트래픽과 완전히 분리하려면 `POD_SECURITY_GROUP_ENFORCING_MODE=strict` 모드에서 SGP를 사용합니다.

포드 및 네트워크 정책에 대한 보안 그룹 사용 모범 사례

- 계층형 보안
  - 계층화된 보안 접근 방식을 위해 SGP와 kubernetes 네트워크 정책의 조합 사용
  - SGPs 사용하여 클러스터에 속하지 않는 AWS 서비스에 대한 네트워크 수준 액세스를 제한하는 반면, kubernetes 네트워크 정책은 클러스터 내의 포드 간 네트워크 트래픽을 제한할 수 있습니다.
- 최소 권한의 원칙
  - 포드 또는 네임스페이스 간에 필요한 트래픽만 허용
- 애플리케이션 세그먼트화
  - 가능하면 네트워크 정책에 따라 애플리케이션을 분할하여 애플리케이션이 손상된 경우 블래스트 반경을 줄입니다.
- 정책을 간단하고 명확하게 유지
  - Kubernetes 네트워크 정책은 매우 세분화되고 복잡할 수 있으므로 구성 오류 위험을 줄이고 관리 오버헤드를 완화하기 위해 최대한 간단하게 유지하는 것이 가장 좋습니다.
- 공격 표면 축소
  - 애플리케이션의 노출을 제한하여 공격 표면 최소화

> **Important**
>
> 포드용 보안 그룹은 `strict` 및의 두 가지 적용 모드를 제공합니다`standard`. EKS 클러스터의 포드 기능에 네트워크 정책 및 보안 그룹을 모두 사용할 때는 `standard` 모드를 사용해야 합니다.

네트워크 보안과 관련하여 계층화된 접근 방식이 가장 효과적인 솔루션인 경우가 많습니다. kubernetes 네트워크 정책과 SGP를 함께 사용하면 EKS에서 실행되는 애플리케이션에 대한 강력한 defense-in-depth 전략을 제공할 수 있습니다.

Service Mesh 정책 적용 또는 Kubernetes 네트워크 정책

`service mesh`는 애플리케이션에 추가할 수 있는 전용 인프라 계층입니다. 이를 통해 자체 코드에 추가하지 않고도 관찰성, 트래픽 관리 및 보안과 같은 기능을 투명하게 추가할 수 있습니다.

서비스 메시는 OSI 모델의 계층 7(애플리케이션)에서 정책을 적용하는 반면 kubernetes 네트워크 정책은 계층 3(네트워크) 및 계층 4(전송)에서 작동합니다. 이 스페이스에는 AWS AppMesh, Istio, Linkerd 등과 같은 다양한 상품이 있습니다.

정책 적용을 위해 서비스 메시를 사용해야 하는 경우

- 서비스 메시에 대한 기존 투자 보유
- 트래픽 관리, 관찰성 및 보안과 같은 고급 기능 필요
  - 트래픽 제어, 로드 밸런싱, 회로 차단, 속도 제한, 제한 시간 등
  - 서비스 성능에 대한 자세한 인사이트(지연 시간, 오류율, 초당 요청 수, 요청 볼륨 등)
  - mTLS와 같은 보안 기능을 위해 서비스 메시를 구현하고 활용하려는 경우

더 간단한 사용 사례를 위해 Kubernetes 네트워크 정책 선택

- 서로 통신할 수 있는 포드 제한
- 네트워크 정책은 서비스 메시보다 더 적은 리소스가 필요하므로 더 간단한 사용 사례 또는 서비스 메시를 실행하고 관리하는 오버헤드가 정당화되지 않을 수 있는 소규모 클러스터에 적합합니다.

> **Note**
>
> 네트워크 정책과 서비스 메시를 함께 사용할 수도 있습니다. 네트워크 정책을 사용하여 포드 간에 기준 수준의 보안 및 격리를 제공한 다음 서비스 메시를 사용하여 트래픽 관리, 관찰성 및 보안과 같은 추가 기능을 추가합니다.

ThirdParty 네트워크 정책 엔진

글로벌 네트워크 정책, DNS 호스트 이름 기반 규칙 지원, 계층 7 규칙, ServiceAccount 기반 규칙 및 명시적 거부/로그 작업 등과 같은 고급 정책 요구 사항이 있는 경우 타사 네트워크 정책 엔진을 고려하세요. Calico는 EKS와 잘 작동하는 Tigera의 오픈 소스 정책 엔진입니다. Calico는 전체 Kubernetes 네트워크 정책 기능 세트를 구현하는 것 외에도 Istio와 통합될 때 HTTP와 같은 계층 7 규칙에 대한 지원을 포함하여 더 풍부한 기능 세트를 갖춘 확장된 네트워크 정책을 지원합니다. Calico 정책의 범위는 네임스페이스, 포드, 서비스 계정 또는 전역으로 지정할 수 있습니다. 정책 범위가 서비스 계정으로 지정되면 수신/송신 규칙 세트를 해당 서비스 계정과 연결합니다. 적절한 RBAC 규칙을 마련하면 팀이 이러한 규칙을 재정의하지 못하도록 하여 IT 보안 전문가가 네임스페이스 관리를 안전하게 위임할 수 있습니다. Cilium의 유지 관리인 등가는 HTTP와 같은 계층 7 규칙에 대한 부분 지원을 포함하도록 네트워크 정책도 확장했습니다. 또한 Cilium은 Kubernetes 서비스/포드와 VPC 내부 또는 외부에서 실행되는 리소스 간의 트래픽을 제한하는 데 유용할 수 있는 DNS 호스트 이름을 지원합니다. 반면 Calico Enterprise에는 Kubernetes 네트워크 정책을 AWS 보안 그룹과 DNS 호스트 이름에 매핑할 수 있는 기능이 포함되어 있습니다.

일반적인 Kubernetes 네트워크 정책 목록은 https://github.com/ahmetb/kubernetes-network-policy-recipes 확인할 수 있습니다. Calico에 대한 유사한 규칙 세트는 https://docs.projectcalico.org/security/calico-network-policy 확인할 수 있습니다.

Amazon VPC CNI 네트워크 정책 엔진으로 마이그레이션

일관성을 유지하고 예기치 않은 포드 통신 동작을 방지하려면 클러스터에 네트워크 정책 엔진을 하나만 배포하는 것이 좋습니다. 3P에서 VPC CNI 네트워크 정책 엔진으로 마이그레이션하려면 VPC CNI 네트워크 정책 지원을 활성화하기 전에 기존 3P NetworkPolicy CRDs를 Kubernetes NetworkPolicy 리소스로 변환하는 것이 좋습니다. 또한 마이그레이션된 정책을 프로덕션 환경에 적용하기 전에 별도의 테스트 클러스터에서 테스트합니다. 이를 통해 포드 통신 동작의 잠재적 문제 또는 불일치를 식별하고 해결할 수 있습니다.

마이그레이션 도구

마이그레이션 프로세스를 지원하기 위해 기존 Calico/Cilium 네트워크 정책 CRD를 Kubernetes 기본 네트워크 정책으로 변환하는 K8s Network Policy Migrator라는 도구를 개발했습니다. CRDs 변환 후 VPC CNI 네트워크 정책 컨트롤러를 실행하는 새 클러스터에서 변환된 네트워크 정책을 직접 테스트할 수 있습니다. 이 도구는 마이그레이션 프로세스를 간소화하고 원활한 전환을 보장하도록 설계되었습니다.

> **Important**
>
> 마이그레이션 도구는 네이티브 kubernetes 네트워크 정책 API와 호환되는 3P 정책만 변환합니다. 3P 플러그인에서 제공하는 고급 네트워크 정책 기능을 사용하는 경우 마이그레이션 도구는 이를 건너뛰고 보고합니다.

마이그레이션 도구는 현재 AWS VPC CNI 네트워크 정책 엔지니어링 팀에서 지원되지 않으며 최선을 다해 고객에게 제공됩니다. 이 도구를 활용하여 마이그레이션 프로세스를 용이하게 하는 것이 좋습니다. 도구에 문제나 버그가 발생하는 경우 GitHub 문제를 생성해 주시기 바랍니다. 귀하의 피드백은 매우 소중하며 서비스를 지속적으로 개선하는 데 도움이 될 것입니다.

추가 리소스

- Kubernetes 및 Tigera: 네트워크 정책, 보안 및 감사
- Calico 엔터프라이즈
- Cilium
- NetworkPolicy Editor Cilium의 대화형 정책 편집기
- Inspektor Gadget, 네트워크 정책 권고 가제트 네트워크 트래픽 분석을 기반으로 네트워크 정책 제안

전송 중 암호화

PCI, HIPAA 또는 기타 규정을 준수해야 하는 애플리케이션은 전송 중 데이터를 암호화해야 할 수 있습니다. 오늘날 TLS는 유선의 트래픽을 암호화하기 위한 사실상의 선택입니다. TLS는 이전 SSL과 마찬가지로 암호화 프로토콜을 사용하여 네트워크를 통해 보안 통신을 제공합니다. TLS는 대칭 암호화를 사용합니다. 여기서 데이터를 암호화하는 키는 세션 시작 시 협상된 공유 보안 암호를 기반으로 생성됩니다. 다음은 Kubernetes 환경에서 데이터를 암호화하는 몇 가지 방법입니다.

Nitro 인스턴스

C5n, G4, I3en, M5dn, M5n, P3dn, R5dn 및 R5n과 같은 Nitro 인스턴스 유형 간에 교환되는 트래픽은 기본적으로 자동으로 암호화됩니다. R5n 전송 게이트웨이 또는 로드 밸런서와 같은 중간 홉이 있는 경우 트래픽은 암호화되지 않습니다. 전송 중 암호화에 대한 자세한 내용과 기본적으로 네트워크 암호화를 지원하는 인스턴스 유형의 전체 목록은 전송 중 암호화를 참조하세요.

컨테이너 네트워크 인터페이스(CNIs)

WeaveNet은 소매 트래픽의 경우 NaCl 암호화를, 빠른 데이터 경로 트래픽의 경우 IPsec ESP를 사용하여 모든 트래픽을 자동으로 암호화하도록 구성할 수 있습니다.

서비스 메시

App Mesh, Linkerd v2, Istio와 같은 서비스 메시를 사용하여 전송 중 암호화를 구현할 수도 있습니다. AppMesh는 X.509 인증서 또는 Envoy의 보안 암호 검색 서비스(SDS)를 통해 mTLS를 지원합니다. Linkerd와 Istio 모두 mTLS를 지원합니다.

aws-app-mesh-examples GitHub 리포지토리는 Envoy 컨테이너와 함께 X.509 인증서 및 SPIRE를 SDS 공급자로 사용하여 mTLS를 구성하는 방법을 안내합니다.

- X.509 인증서를 사용하여 mTLS 구성
- SPIRE(SDS)를 사용하여 TLS 구성

또한 App Mesh는 AWS Certificate Manager(ACM)에서 발급한 프라이빗 인증서 또는 가상 노드의 로컬 파일 시스템에 저장된 인증서를 사용하여 TLS 암호화를 지원합니다.

aws-app-mesh-examples GitHub 리포지토리는 ACM에서 발급한 인증서와 Envoy 컨테이너와 함께 패키징된 인증서를 사용하여 TLS를 구성하는 방법을 안내합니다.

- 파일 제공 TLS 인증서를 사용하여 TLS 구성
- AWS Certificate Manager를 사용하여 TLS 구성

수신 컨트롤러 및 로드 밸런서

수신 컨트롤러는 클러스터 외부에서 발생하는 HTTP/S 트래픽을 클러스터 내에서 실행되는 서비스로 지능적으로 라우팅하는 방법입니다. 이러한 수신은 Classic Load Balancer 또는 Network Load Balancer(NLB)와 같은 계층 4 Load Balancer가 앞에 오는 경우가 많습니다. 암호화된 트래픽은 로드 밸런서, 수신 리소스 또는 포드 등 네트워크 내의 여러 위치에서 종료될 수 있습니다. SSL 연결을 종료하는 방법과 위치는 궁극적으로 조직의 네트워크 보안 정책에 따라 결정됩니다. 예를 들어 end-to-end 암호화가 필요한 정책이 있는 경우 포드에서 트래픽을 복호화해야 합니다. 이렇게 하면 초기 핸드셰이크를 설정하는 주기를 소비해야 하므로 포드에 추가 부담이 발생합니다. 전체 SSL/TLS 처리는 CPU 집약적입니다. 따라서 유연성이 있는 경우 수신 또는 로드 밸런서에서 SSL 오프로드를 수행해 보세요.

AWS Elastic 로드 밸런서에서 암호화 사용

AWS Application Load Balancer(ALB)와 Network Load Balancer(NLB) 모두 전송 암호화(SSL 및 TLS)를 지원합니다. ALB에 대한 `alb.ingress.kubernetes.io/certificate-arn` 주석을 사용하면 ALB에 추가할 인증서를 지정할 수 있습니다. 주석을 생략하면 컨트롤러는 호스트 필드를 사용하여 사용 가능한 AWS Certificate Manager(ACM) 인증서와 일치시켜 필요한 인증서를 리스너에 추가하려고 시도합니다. EKS v1.15부터 아래 예제와 같이 NLB와 함께 `service.beta.kubernetes.io/aws-load-balancer-ssl-cert` 주석을 사용할 수 있습니다.

```
apiVersion: v1
kind: Service
metadata:
  name: demo-app
  namespace: default
  labels:
    app: demo-app
  annotations:
     service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
     service.beta.kubernetes.io/aws-load-balancer-ssl-cert: "<certificate ARN>"
     service.beta.kubernetes.io/aws-load-balancer-ssl-ports: "443"
     service.beta.kubernetes.io/aws-load-balancer-backend-protocol: "http"
spec:
  type: LoadBalancer
  ports:
  - port: 443
    targetPort: 80
    protocol: TCP
  selector:
    app: demo-app
//---
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx
  namespace: default
  labels:
    app: demo-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: demo-app
  template:
```

```
    metadata:
      labels:
        app: demo-app
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 443
              protocol: TCP
            - containerPort: 80
              protocol: TCP
```

다음은 SSL/TLS 종료에 대한 추가 예제입니다.

- Contour And Let's Encrypt The GitOps 방식으로 EKS 수신 보호
- ACM을 사용하여 Amazon EKS 워크로드에서 HTTPS 트래픽을 종료하려면 어떻게 해야 합니까?

> **Important**
>
> AWS LB 컨트롤러와 같은 일부 수신은 수신 사양의 일부로 대신 주석을 사용하여 SSL/TLS를 구현합니다.

cert-manager를 사용하는 ACM Private CA

인증서를 배포, 갱신 및 취소하는 인기 있는 Kubernetes 추가 기능인 ACM Private Certificate Authority(CA) 및 cert-manager를 사용하여 TLS 및 mTLS를 활성화하여 수신 시, 포드에서, 포드 간에 EKS 애플리케이션 워크로드를 보호할 수 있습니다. ACM Private CA는 자체 CA를 관리하는 데 드는 선결제 및 유지 관리 비용 없이 가용성이 높고 안전한 관리형 CA입니다. 기본 Kubernetes 인증 기관을 사용하는 경우 보안을 개선하고 ACM Private CA의 규정 준수 요구 사항을 충족할 수 있습니다. ACM Private CA는 메모리에 인코딩된 키를 저장하는 기본 CA(안전하지 않음)에 비해 FIPS 140-2 레벨 3 하드웨어 보안 모듈에서 프라이빗 키를 보호합니다(매우 안전함). 또한 중앙 집중식 CA를 사용하면 Kubernetes 환경 내부 및 외부 모두에서 프라이빗 인증서에 대한 제어를 강화하고 감사 가능성을 개선할 수 있습니다.

워크로드 간 상호 TLS를 위한 단기 CA 모드

EKS의 mTLS에 ACM Private CA를 사용하는 경우 수명이 짧은 CA 모드에서 수명이 짧은 인증서를 사용하는 것이 좋습니다. 범용 CA 모드에서 수명이 짧은 인증서를 발급할 수 있지만, 수명이 짧은 CA

모드를 사용하면 새 인증서를 자주 발급해야 하는 사용 사례에 더 비용 효율적으로(일반 모드보다 약 75% 저렴) 사용할 수 있습니다. 또한 프라이빗 인증서의 유효 기간을 EKS 클러스터의 포드 수명에 맞게 조정해야 합니다. 여기에서 ACM Private CA와 그 이점에 대해 자세히 알아보세요.

ACM 설정 지침

먼저 ACM Private CA 기술 문서에 제공된 절차에 따라 Private CA를 생성합니다. Private CA가 있으면 일반 설치 지침에 따라 cert-manager를 설치합니다. cert-manager를 설치한 후 GitHub의 설정 지침에 따라 Private CA Kubernetes cert-manager 플러그인을 설치합니다. 플러그인을 사용하면 cert-manager가 ACM Private CA에서 프라이빗 인증서를 요청할 수 있습니다.

이제 Private CA와 cert-manager 및 플러그인이 설치된 EKS 클러스터가 있으므로 권한을 설정하고 발급자를 생성할 차례입니다. ACM Private CA에 대한 액세스를 허용하도록 EKS 노드 역할의 IAM 권한을 업데이트합니다. 를 프라이빗 CA<CA_ARN>의 값으로 바꿉니다.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "awspcaissuer",
            "Action": [
                "acm-pca:DescribeCertificateAuthority",
                "acm-pca:GetCertificate",
                "acm-pca:IssueCertificate"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:acm-pca:us-west-2:123456789012:certificate-
authority/12345678-1234-1234-1234-123456789012"
        }
    ]
}
```

IAM 계정 또는 IRSA에 대한 서비스 역할도 사용할 수 있습니다. 전체 예제는 아래의 추가 리소스 섹션을 참조하세요.

다음 텍스트가 포함된 cluster-issuer.yaml이라는 사용자 지정 리소스 정의 파일을 생성하고 <CA_ARN> 및 <Region> 정보를 프라이빗 CA로 대체하여 Amazon EKS에서 발급자를 생성합니다.

```
apiVersion: awspca.cert-manager.io/v1beta1
kind: AWSPCAClusterIssuer
metadata:
```

```
          name: demo-test-root-ca
spec:
          arn: <CA_ARN>
          region: <Region>
```

생성한 발급자를 배포합니다.

```
kubectl apply -f cluster-issuer.yaml
```

EKS 클러스터는 Private CA에서 인증서를 요청하도록 구성되어 있습니다. 이제 cert-manager의 `Certificate` 리소스를 사용하여 `issuerRef` 필드 값을 위에서 생성한 프라이빗 CA 발급자로 변경하여 인증서를 발급할 수 있습니다. 인증서 리소스를 지정하고 요청하는 방법에 대한 자세한 내용은 cert-manager의 인증서 리소스 가이드를 참조하세요. 여기에서 예제를 참조하세요.

Istio 및 cert-manager가 포함된 ACM Private CA

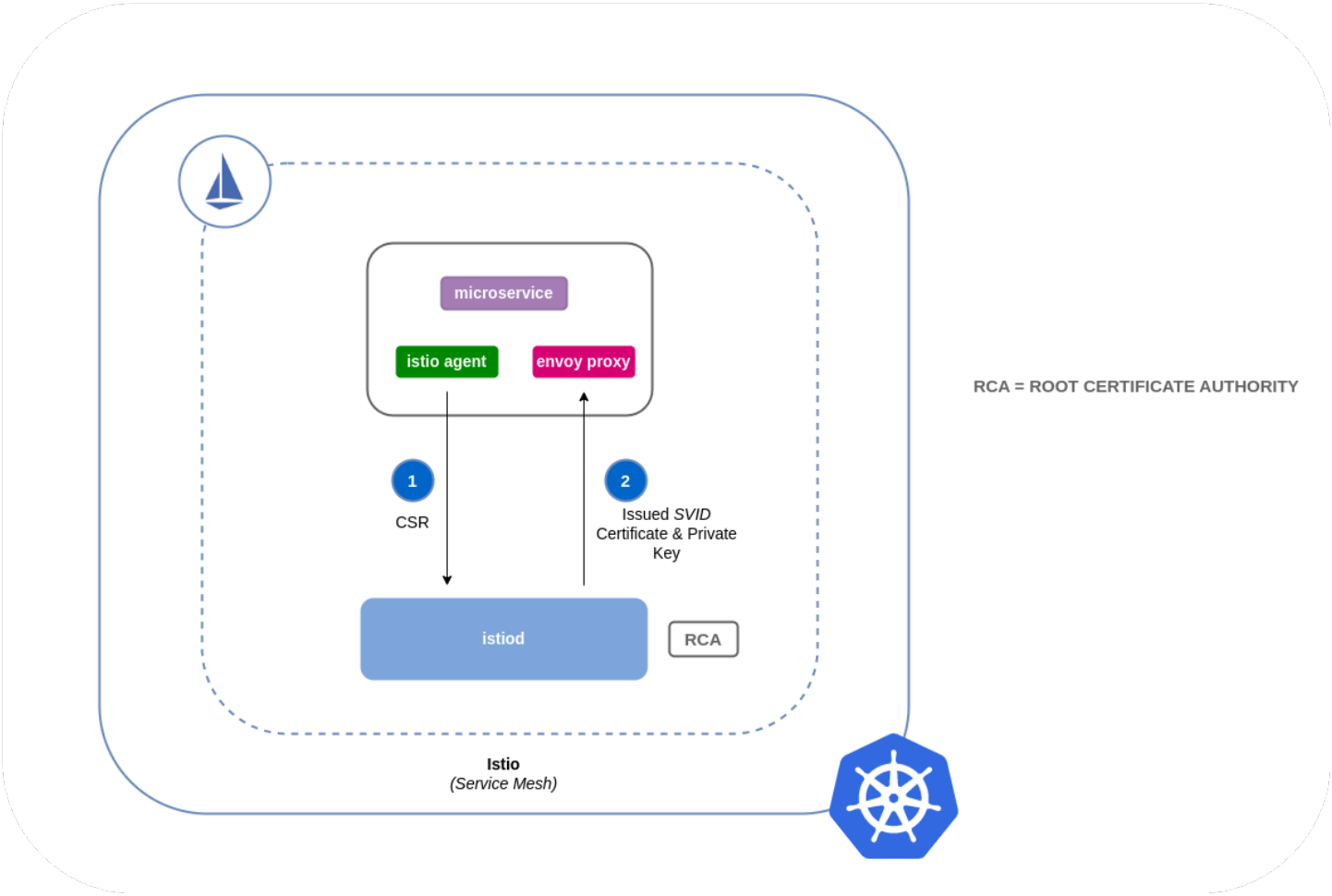
EKS 클러스터에서 Istio를 실행하는 경우 Istio 컨트롤 플레인(특히 `istiod`)이 루트 인증 기관(CA)으로 작동하지 않도록 비활성화하고 ACM Private CA를 워크로드 간 mTLS의 루트 CA로 구성할 수 있습니다. 이 접근 방식을 사용하는 경우 ACM Private CA에서 수명이 짧은 CA 모드를 사용하는 것이 좋습니다. 자세한 내용은 이전 섹션과이 블로그 게시물을 참조하세요.

Istio에서 인증서 서명 작동 방식(기본값)

Kubernetes의 워크로드는 서비스 계정을 사용하여 식별됩니다. 서비스 계정을 지정하지 않으면 Kubernetes가 워크로드에 자동으로 할당합니다. 또한 서비스 계정은 연결된 토큰을 자동으로 탑재합니다. 이 토큰은 서비스 계정에서 워크로드가 Kubernetes API에 대해 인증하는 데 사용됩니다. 서비스 계정은 Kubernetes의 자격 증명으로 충분할 수 있지만 Istio에는 자체 자격 증명 관리 시스템과 CA가 있습니다. 워크로드가 인코딩 사이드카 프록시로 시작되면 신뢰할 수 있는 것으로 간주되고 메시의 다른 서비스와 통신할 수 있도록 Istio에서 할당된 자격 증명이 필요합니다.

Istio에서이 자격 증명을 가져오기 위해는 인증서 서명 요청(또는 CSR)이라는 요청을 Istio 컨트롤 플레인으로 `istio-agent` 보냅니다. 이 CSR에는 처리 전에 워크로드의 자격 증명을 확인할 수 있도록 서비스 계정 토큰이 포함되어 있습니다. 이 확인 프로세스는 등록 기관(또는 RA)과 CA의 `istiod`역할을 하는에서 처리합니다. RA는 확인된 CSR만 CA로 전달하도록 하는 게이트키퍼 역할을 합니다. CSR이 확인되면 CA로 전달되어 서비스 계정에 SPIFFE 자격 증명이 포함된 인증서를 발급합니다. 이 인증서를 SPIFFE 확인 가능 자격 증명 문서(또는 SVID)라고 합니다. SVID는 식별을 위해 요청 서비스에 할당되고 통신 서비스 간에 전송 중인 트래픽을 암호화합니다.

Istio 인증서 서명 요청의 기본 흐름:



ACM Private CA를 사용하는 Istio에서 인증서 서명 작동 방식

Istio Certificate Signing Request 에이전트(istio-csr)라는 cert-manager 추가 기능을 사용하여 Istio를 ACM Private CA와 통합할 수 있습니다. 이 에이전트를 사용하면 Istio 워크로드 및 컨트롤 플레인 구성 요소를 인증서 관리자 발급자,이 경우 ACM Private CA로 보호할 수 있습니다. istio-csr 에이전트는 수신 CSR을 검증하는 기본 구성에서 istiod가 제공하는 것과 동일한 서비스를 노출합니다. CSRs 단, 확인 후에는 요청을 인증서 관리자가 지원하는 리소스(예: 외부 CA 발급자와의 통합)로 변환합니다.

워크로드에서 CSR이 있을 때마다 istio-csr로 전달되어 ACM Private CA에 인증서를 요청합니다. istio-csr와 ACM Private CA 간의이 통신은 AWS Private CA 발급자 플러그인에 의해 활성화됩니다. Cert Manager는이 플러그인을 사용하여 ACM Private CA에서 TLS 인증서를 요청합니다. 발급자 플러그인은 ACM Private CA 서비스와 통신하여 워크로드에 대해 서명된 인증서를 요청합니다. 인증서가 서명되면 istio-csr로 반환되어 서명된 요청을 읽고 CSR을 시작한 워크로드로 반환됩니다.

istio-csr을 사용한 Istio 인증서 서명 요청 흐름

image::istio-csr-with-acm-private-ca.png[istio-csr을 사용한 Istio 인증서 서명 요청 흐름]

프라이빗 CA 설정 지침이 포함된 Istio

1. 먼저이 섹션의 동일한 설정 지침에 따라 다음을 완료합니다.
2. 사설 CA 생성
3. cert-manager 설치
4. 발급자 플러그인 설치
5. 권한을 설정하고 발급자를 생성합니다. 발급자는 CA를 나타내며 워크로드 인증서에 서명하고 `istiod` 메시하는 데 사용됩니다. ACM Private CA와 통신합니다.
6. `istio-system` 네임스페이스를 생성합니다. 여기에서 `istiod certificate` 및 기타 Istio 리소스가 배포됩니다.
7. AWS Private CA 발급자 플러그인으로 구성된 Istio CSR을 설치합니다. 워크로드에 대한 인증서 서명 요청을 보존하여 승인 및 서명()되었는지 확인할 수 있습니다`preserveCertificateRequests=true`.

```
helm install -n cert-manager cert-manager-istio-csr jetstack/cert-manager-istio-csr \
--set "app.certmanager.issuer.group=awspca.cert-manager.io" \
--set "app.certmanager.issuer.kind=AWSPCAClusterIssuer" \
--set "app.certmanager.issuer.name=<the-name-of-the-issuer-you-created>" \
--set "app.certmanager.preserveCertificateRequests=true" \
--set "app.server.maxCertificateDuration=48h" \
--set "app.tls.certificateDuration=24h" \
--set "app.tls.istiodCertificateDuration=24h" \
--set "app.tls.rootCAFile=/var/run/secrets/istio-csr/ca.pem" \
--set "volumeMounts[0].name=root-ca" \
--set "volumeMounts[0].mountPath=/var/run/secrets/istio-csr" \
--set "volumes[0].name=root-ca" \
--set "volumes[0].secret.secretName=istio-root-ca"
```

8. 를 메시의 인증서 공급자로 대체할 사용자 지정 구성`istiod`으로 `cert-manager istio-csr` Istio를 설치합니다. 이 프로세스는 Istio 연산자를 사용하여 수행할 수 있습니다.

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: istio
  namespace: istio-system
spec:
  profile: "demo"
  hub: gcr.io/istio-release
  values:
```

```
global:
  # Change certificate provider to cert-manager istio agent for istio agent
  caAddress: cert-manager-istio-csr.cert-manager.svc:443
components:
  pilot:
    k8s:
      env:
        # Disable istiod CA Sever functionality
      - name: ENABLE_CA_SERVER
        value: "false"
      overlays:
      - apiVersion: apps/v1
        kind: Deployment
        name: istiod
        patches:

          # Mount istiod serving and webhook certificate from Secret mount
        - path: spec.template.spec.containers.[name:discovery].args[7]
          value: "--tlsCertFile=/etc/cert-manager/tls/tls.crt"
        - path: spec.template.spec.containers.[name:discovery].args[8]
          value: "--tlsKeyFile=/etc/cert-manager/tls/tls.key"
        - path: spec.template.spec.containers.[name:discovery].args[9]
          value: "--caCertFile=/etc/cert-manager/ca/root-cert.pem"

        - path: spec.template.spec.containers.[name:discovery].volumeMounts[6]
          value:
            name: cert-manager
            mountPath: "/etc/cert-manager/tls"
            readOnly: true
        - path: spec.template.spec.containers.[name:discovery].volumeMounts[7]
          value:
            name: ca-root-cert
            mountPath: "/etc/cert-manager/ca"
            readOnly: true

        - path: spec.template.spec.volumes[6]
          value:
            name: cert-manager
            secret:
              secretName: istiod-tls
        - path: spec.template.spec.volumes[7]
          value:
            name: ca-root-cert
            configMap:
```

```
                defaultMode: 420
                name: istio-ca-root-cert
```

9. 생성한 위의 사용자 지정 리소스를 배포합니다.

```
istioctl operator init
kubectl apply -f istio-custom-config.yaml
```

10.이제 EKS 클러스터의 메시에 워크로드를 배포하고 mTLS를 적용할 수 있습니다.

Istio 인증서 서명 요청

image::istio-csr-requests.png[Istio 인증서 서명 요청]

도구 및 리소스

- Amazon EKS Security Immersion 워크숍 - 네트워크 보안
- cert-manager 및 ACM Private CA 플러그인을 구현하여 EKS에서 TLS를 활성화하는 방법.
- 새로운 AWS Load Balancer Controller 및 ACM Private CA를 사용하여 Amazon EKS에서 end-to-end TLS 암호화를 설정합니다.
- GitHub의 Private CA Kubernetes cert-manager 플러그인.
- Private CA Kubernetes cert-manager 플러그인 사용 설명서.
- AWS Private Certificate Authority 단기 인증서 모드를 사용하는 방법
- egress-operator 프로토콜 검사 없이 클러스터에서 송신 트래픽을 제어하는 운영자 및 DNS 플러그인
- SUSE 오픈 소스, 제로 트러스트 컨테이너 보안 플랫폼의 NeuVector는 정책 네트워크 규칙, 데이터 손실 방지(DLP), 웹 애플리케이션 방화벽(WAF) 및 네트워크 위협 서명을 제공합니다.

데이터 암호화 및 보안 암호 관리

저장 시 암호화

Kubernetes에는 EBS, EFS EFS및 FSx for Lustre라는 세 가지 AWS 네이티브 스토리지 옵션이 있습니다. 세 가지 모두 서비스 관리형 키 또는 고객 마스터 키(CMK)를 사용하여 저장 시 암호화를 제공합니다. EBS의 경우 트리 내 스토리지 드라이버 또는 EBS CSI 드라이버를 사용할 수 있습니다. 둘 다 볼륨을 암호화하고 CMK를 제공하기 위한 파라미터를 포함합니다. EFS의 경우 EFS CSI 드라이

버를 사용할 수 있지만 EBS와 달리 EFS CSI 드라이버는 동적 프로비저닝을 지원하지 않습니다. EFS를 EKS와 함께 사용하려면 PV를 생성하기 전에 파일 시스템에 대한 저장 데이터 암호화를 프로비저닝하고 구성해야 합니다. EFS 파일 암호화에 대한 자세한 내용은 유휴 데이터 암호화를 참조하세요. 유휴 데이터 암호화 외에도 EFS 및 FSx for Lustre에는 전송 중 데이터를 암호화하는 옵션이 포함되어 있습니다. FSx for Lustre는 기본적으로이 작업을 수행합니다. EFS의 경우이 예제와 같이 PV의 `mountOptions`에 `tls` 파라미터를 추가하여 전송 암호화를 추가할 수 있습니다.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: efs-pv
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: efs-sc
  mountOptions:
    - tls
  csi:
    driver: efs.csi.aws.com
    volumeHandle: <file_system_id>
```

FSx CSI 드라이버는 Lustre 파일 시스템의 동적 프로비저닝을 지원합니다. 이 예제와 같이 자체 CMK를 제공하는 옵션이 있지만 기본적으로 서비스 관리형 키로 데이터를 암호화합니다.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fsx-sc
provisioner: fsx.csi.aws.com
parameters:
  subnetId: subnet-056da83524edbe641
  securityGroupIds: sg-086f61ea73388fb6b
  deploymentType: PERSISTENT_1
  kmsKeyId: <kms_arn>
```

> **Important**
>
> 2020년 5월 28일부터 EKS Fargate 포드의 임시 볼륨에 기록된 모든 데이터는 기본적으로 업계 표준 AES-256 암호화 알고리즘을 사용하여 암호화됩니다. 암호화 및 복호화는 서비스에서 원활하게 처리되므로 애플리케이션을 수정할 필요가 없습니다.

저장 데이터 암호화

저장 데이터를 암호화하는 것이 모범 사례로 간주됩니다. 암호화가 필요한지 확실하지 않은 경우 데이터를 암호화합니다.

CMKs 주기적으로 교체

CMKs 자동으로 교체하도록 KMS를 구성합니다. 이렇게 하면 이전 키를 무기한 저장하면서 1년에 한 번 키를 교체하여 데이터를 해독할 수 있습니다. 자세한 내용은 고객 마스터 키 교체를 참조하세요.

EFS 액세스 포인트를 사용하여 공유 데이터 세트에 대한 액세스 간소화

POSIX 파일 권한이 다른 공유 데이터 세트가 있거나 다른 탑재 지점을 생성하여 공유 파일 시스템의 일부에 대한 액세스를 제한하려는 경우 EFS 액세스 포인트를 사용하는 것이 좋습니다. 액세스 포인트 작업에 대한 자세한 내용은 https://docs.aws.amazon.com/efs/latest/ug/efs-access-points.html 참조하십시오. 오늘 액세스 포인트(AP)를 사용하려면 PV `volumeHandle` 파라미터에서 AP를 참조해야 합니다.

> **Important**
>
> 2021년 3월 23일부터 EFS CSI 드라이버는 EFS 액세스 포인트의 동적 프로비저닝을 지원합니다. 액세스 포인트는 여러 포드 간에 파일 시스템을 더 쉽게 공유할 수 있는 EFS 파일 시스템의 애플리케이션별 진입점입니다. 각 EFS 파일 시스템은 최대 120개의 PVs. 자세한 내용은 Amazon EFS CSI 동적 프로비저닝 소개를 참조하세요.

보안 암호 관리

Kubernetes 보안 암호는 사용자 인증서, 암호 또는 API 키와 같은 민감한 정보를 저장하는 데 사용됩니다. 이러한 문자열은 etcd에서 base64 인코딩 문자열로 유지됩니다. EKS에서 etcd 노드의 EBS 볼륨은 EBS 암호화로 암호화됩니다. 포드는의 보안 암호를 참조하여 Kubernetes 보안 암호 객체를 검색할 수 있습니다`podSpec`. 이러한 보안 암호는 환경 변수에 매핑하거나 볼륨으로 마운트할 수 있습니다. 보안

암호 생성에 대한 자세한 내용은 https://kubernetes.io/docs/concepts/configuration/secret/ 참조하십시오.

> **Warning**
>
> 특정 네임스페이스의 보안 암호는 보안 암호 네임스페이스의 모든 포드에서 참조할 수 있습니다.

> **Warning**
>
> 노드 권한 부여자를 사용하면 Kubelet이 노드에 마운트된 모든 보안 암호를 읽을 수 있습니다.

Kubernetes 보안 암호의 봉투 암호화에 AWS KMS 사용

이렇게 하면 고유한 데이터 암호화 키(DEK)를 사용하여 보안 암호를 암호화할 수 있습니다. 그런 다음 DEK는 반복 일정에 따라 자동으로 교체될 수 있는 AWS KMS의 키 암호화 키(KEK)를 사용하여 암호화됩니다. Kubernetes용 KMS 플러그인을 사용하면 모든 Kubernetes 보안 암호가 일반 텍스트 대신 사이퍼텍스트로 etcd에 저장되며 Kubernetes API 서버에서만 복호화할 수 있습니다. 자세한 내용은 [심층 방어를 위한 EKS 암호화 공급자 지원 사용을 참조하세요.]

Kubernetes 보안 암호 사용 감사

EKS에서 감사 로깅을 켜고 CloudWatch 지표 필터 및 경보를 생성하여 보안 암호가 사용될 때 알려줍니다(선택 사항). 다음은 Kubernetes 감사 로그에 대한 지표 필터의 예입니다`{($.verb="get") && ($.objectRef.resource="secret")}`. CloudWatch Log Insights에서 다음 쿼리를 사용할 수도 있습니다.

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| stats count(*) by objectRef.name as secret
| filter verb="get" and objectRef.resource="secrets"
```

위 쿼리에는 특정 기간 내에 보안 암호에 액세스한 횟수가 표시됩니다.

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
```

```
| filter verb="get" and objectRef.resource="secrets"
| display objectRef.namespace, objectRef.name, user.username, responseStatus.code
```

이 쿼리는 보안 암호 및 응답 코드에 액세스하려고 시도한 사용자의 네임스페이스 및 사용자 이름과 함께 보안 암호를 표시합니다.

보안 암호를 주기적으로 교체

Kubernetes는 보안 암호를 자동으로 교체하지 않습니다. 보안 암호를 교체해야 하는 경우 볼트 또는 AWS Secrets Manager와 같은 외부 보안 암호 저장소를 사용하는 것이 좋습니다.

다른 애플리케이션에서 보안 암호를 격리하는 방법으로 별도의 네임스페이스 사용

네임스페이스의 애플리케이션 간에 공유할 수 없는 보안 암호가 있는 경우 해당 애플리케이션에 대해 별도의 네임스페이스를 생성합니다.

환경 변수 대신 볼륨 마운트 사용

환경 변수의 값은 의도하지 않게 로그에 나타날 수 있습니다. 볼륨으로 마운트된 보안 암호는 포드가 삭제될 때 노드에서 자동으로 제거되는 tmpfs 볼륨(RAM 지원 파일 시스템)으로 인스턴스화됩니다.

외부 보안 암호 공급자 사용

AWS Secrets Manager 및 Hashicorp's Vault를 포함하여 Kubernetes 보안 암호 사용에 대한 몇 가지 실행 가능한 대안이 있습니다. 이러한 서비스는 세분화된 액세스 제어, 강력한 암호화, Kubernetes Secrets에서 사용할 수 없는 보안 암호의 자동 교체와 같은 기능을 제공합니다. Bitnami의 봉인된 보안 암호는 비대칭 암호화를 사용하여 "봉인된 보안 암호"를 생성하는 또 다른 접근 방식입니다. 퍼블릭 키는 보안 암호를 암호화하는 데 사용되며 보안 암호를 해독하는 데 사용되는 프라이빗 키는 클러스터 내에 유지되므로 Git과 같은 소스 제어 시스템에 봉인된 보안 암호를 안전하게 저장할 수 있습니다. 자세한 내용은 봉인된 보안 암호를 사용하여 Kubernetes에서 보안 암호 배포 관리를 참조하세요.

외부 보안 암호 스토어의 사용이 증가함에 따라는 이를 Kubernetes와 통합해야 합니다. 보안 암호 스토어 CSI 드라이버는 CSI 드라이버 모델을 사용하여 외부 보안 암호 스토어에서 보안 암호를 가져오는 커뮤니티 프로젝트입니다. 현재 드라이버는 AWS Secrets Manager, Azure, Vault 및 GCP를 지원합니다. AWS 공급자는 AWS Secrets Manager와 AWS Parameter Store를 모두 지원합니다. 또한 보안 암호가 만료될 때 보안 암호를 교체하도록 구성할 수 있으며 AWS Secrets Manager 보안 암호를 Kubernetes 보안 암호와 동기화할 수 있습니다. 보안 암호 동기화는 보안 암호를 볼륨에서 읽는 대신 환경 변수로 참조해야 할 때 유용할 수 있습니다.

> **Note**
>
> 보안 암호 스토어 CSI 드라이버가 보안 암호를 가져와야 하는 경우 보안 암호를 참조하는 포드에 할당된 IRSA 역할을 수임합니다. 이 작업의 코드는 여기에서 찾을 수 있습니다.

AWS Secrets & Configuration Provider(ASCP)에 대한 자세한 내용은 다음 리소스를 참조하세요.

- Kubernetes 보안 암호 스토어 CSI 드라이버와 함께 AWS 보안 암호 구성 공급자를 사용하는 방법
- Secrets Manager 보안 암호를 Kubernetes 보안 암호 스토어 CSI 드라이버와 통합

외부 보안 암호는 Kubernetes에서 외부 보안 암호 스토어를 사용하는 또 다른 방법입니다. CSI 드라이버와 마찬가지로 외부 보안 암호는 AWS Secrets Manager를 비롯한 다양한 백엔드에서 작동합니다. 차이점은 외부 보안 암호 스토어에서 보안 암호를 검색하는 대신 외부 보안 암호가 이러한 백엔드의 보안 암호를 보안 암호로 Kubernetes에 복사한다는 것입니다. 이를 통해 원하는 보안 암호 스토어를 사용하여 보안 암호를 관리하고 Kubernetes 네이티브 방식으로 보안 암호와 상호 작용할 수 있습니다.

도구 및 리소스

- Amazon EKS Security Immersion 워크숍 - 데이터 암호화 및 보안 암호 관리

런타임 보안

런타임 보안은 컨테이너가 실행되는 동안 컨테이너를 능동적으로 보호합니다. 이 아이디어는 컨테이너 내에서 악의적인 활동이 발생하는 것을 감지 및/또는 방지하는 것입니다. 이는 Linux 기능, 보안 컴퓨팅(seccomp), AppArmor 또는 SELinux와 같이 Kubernetes와 통합된 Linux 커널 또는 커널 확장의 여러 메커니즘을 통해 달성할 수 있습니다. 또한 Amazon GuardDuty 및 타사 도구와 같은 옵션이 있어 Linux 커널 메커니즘의 수동 구성을 줄이면서 기준을 설정하고 비정상적인 활동을 감지하는 데 도움이 될 수 있습니다.

> **Important**
>
> Kubernetes는 현재 seccomp, AppArmor 또는 SELinux 프로파일을 노드에 로드하기 위한 기본 메커니즘을 제공하지 않습니다. 수동으로 로드하거나 부트스트래핑할 때 노드에 설치해야 합니다. 스케줄러가 프로파일이 있는 노드를 인식하지 못하므로 포드에서 이를 참조하기 전에이

작업을 수행해야 합니다. Security Profiles Operator와 같은 도구가 노드에 대한 프로필 프로비저닝을 자동화하는 데 어떻게 도움이 되는지 아래에서 확인하세요.

보안 컨텍스트 및 내장 Kubernetes 제어

많은 Linux 런타임 보안 메커니즘은 Kubernetes와 긴밀하게 통합되며 Kubernetes 보안 컨텍스트를 통해 구성할 수 있습니다. 이러한 옵션 중 하나는 `privileged` 플래그입니다. 플래그는 `false` 기본적으로 이고 활성화된 경우 기본적으로 호스트의 루트와 동일합니다. 프로덕션 워크로드에서 권한 모드를 활성화하는 것은 거의 항상 부적절하지만 컨테이너에 더 세분화된 권한을 적절하게 제공할 수 있는 제어가 더 많습니다.

Linux 기능

Linux 기능을 사용하면 루트 사용자의 모든 기능을 제공하지 않고도 포드 또는 컨테이너에 특정 기능을 부여할 수 있습니다. 예를 들어 네트워크 인터페이스 또는 방화벽을 구성할 수 `CAP_NET_ADMIN`있는 , 또는 시스템 클럭을 조작할 수 `CAP_SYS_TIME`있는가 있습니다.

Seccomp

보안 컴퓨팅(seccomp)을 사용하면 컨테이너화된 애플리케이션이 기본 호스트 운영 체제의 커널을 특정 syscall하지 못하도록 할 수 있습니다. Linux 운영 체제에는 수백 개의 시스템 호출이 있지만 컨테이너를 실행하는 데는이 중 사자의 공유가 필요하지 않습니다. 컨테이너에서 수행할 수 있는 시스템 호출을 제한하면 애플리케이션의 공격 표면을 효과적으로 줄일 수 있습니다.

Seccomp는 시스템 호출을 가로채고 허용 목록에 있는 호출만 통과하도록 허용하는 방식으로 작동합니다. Docker에는 대부분의 범용 워크로드에 적합한 기본 seccomp 프로파일이 있으며, 컨테이너식과 같은 다른 컨테이너 런타임은 유사한 기본값을 제공합니다. 포드 사양의 `securityContext` 섹션에 다음을 추가하여 컨테이너 런타임의 기본 seccomp 프로파일을 사용하도록 컨테이너 또는 포드를 구성할 수 있습니다.

```
securityContext:
  seccompProfile:
    type: RuntimeDefault
```

1.22부터(알파로 표시, 1.27부터 안정적) 단일 kubelet 플래그를 사용하여 노드의 모든 포드에 위의를 사용할 `RuntimeDefault` 수 있습니다`--seccomp-default`. 그러면에 지정된 프로필`securityContext`은 다른 프로필에만 필요합니다.

추가 권한이 필요한 사물에 대해 자체 프로필을 생성할 수도 있습니다. 이는 수동으로 수행하는 것이 매우 지루할 수 있지만, eBPF 또는 로그와 같은 도구를 사용하여 기준 권한 요구 사항을 seccomp 프로파일로 기록하도록 지원하는 Inspektor Gadget(네트워크 정책 생성을 위한 네트워크 보안 섹션에서도 권장됨) 및 Security Profiles Operator와 같은 도구가 있습니다. 또한 Security Profiles Operator를 사용하면 포드 및 컨테이너에서 사용할 수 있도록 기록된 프로필을 노드에 자동으로 배포할 수 있습니다.

AppArmor 및 SELinux

AppArmor 및 SELinux를 필수 액세스 제어 또는 MAC 시스템이라고 합니다. seccomp와 개념이 비슷하지만 APIs와 기능이 다르기 때문에 특정 파일 시스템 경로 또는 네트워크 포트에 대한 액세스 제어가 가능합니다. 이러한 도구에 대한 지원은 Linux 배포판에 따라 달라지며, Debian/Ubuntu는 AppArmor를 지원하고 RHEL/CentOS/Bottlerocket/Amazon Linux 2023은 SELinux를 지원합니다. SELinux에 대한 자세한 내용은 인프라 보안 섹션을 참조하세요.

AppArmor와 SELinux는 모두 Kubernetes와 통합되지만 Kubernetes 1.28부터는 주석을 통해 AppArmor 프로파일을 지정해야 하며 SELinux 레이블은 보안 컨텍스트의 SELinuxOptions 필드를 통해 직접 설정할 수 있습니다.

seccomp 프로파일과 마찬가지로 위에서 언급한 Security Profiles Operator는 클러스터의 노드에 프로파일을 배포하는 데 도움을 줄 수 있습니다. (향후 프로젝트는 seccomp와 마찬가지로 AppArmor 및 SELinux에 대한 프로필도 생성하는 것을 목표로 합니다.)

권장 사항

Amazon GuardDuty를 사용하여 EKS 환경에 대한 위협 모니터링 및 탐지

현재 EKS 런타임을 지속적으로 모니터링하고 EKS 감사 로그를 분석하고 맬웨어 및 기타 의심스러운 활동을 검사할 수 있는 솔루션이 없는 경우 Amazon은 AWS 환경을 보호하는 간단하고 빠르고 안전하며 확장 가능하고 비용 효율적인 원클릭 방법을 원하는 고객에게 Amazon GuardDuty를 사용할 것을 강력히 권장합니다. Amazon GuardDuty는 AWS CloudTrail 관리 이벤트, AWS CloudTrail 이벤트 로그, VPC 흐름 로그(Amazon EC2 인스턴스), Kubernetes 감사 로그 및 DNS 로그와 같은 기본 데이터 소스를 분석하고 처리하는 보안 모니터링 서비스입니다. 여기에는 EKS 런타임 모니터링도 포함됩니다. 악성 IP 주소 및 도메인 목록, 기계 학습 등 지속적으로 업데이트된 위협 인텔리전스 피드를 사용하여 AWS 환경 내에서 예상치 못한, 잠재적으로 승인되지 않은, 악의적인 활동을 식별합니다. 여기에는 권한 에스컬레이션, 노출된 자격 증명 사용, 악성 IP 주소, 도메인과의 통신, Amazon EC2 인스턴스 및 EKS 컨테이너 워크로드에 멀웨어 존재, 의심스러운 API 활동 검색과 같은 문제가 포함될 수 있습니다. GuardDuty는 GuardDuty 콘솔 또는 Amazon EventBridge를 통해 볼 수 있는 보안 조사 결과를 생성하여 AWS 환경의 상태를 알려줍니다. 또한 GuardDuty는 조사 결과를 Amazon Simple Storage

Service(S3) 버킷으로 내보내고 AWS Security Hub 및 Detective와 같은 다른 서비스와 통합할 수 있도록 지원합니다.

이 AWS Online Tech Talk "Amazon GuardDuty를 사용한 Amazon EKS에 대한 향상된 위협 탐지 - AWS Online Tech Talks"를 시청하여 이러한 추가 EKS 보안 기능을 몇 분 만에 step-by-step 활성화하는 방법을 확인하세요.

선택 사항: 런타임 모니터링을 위한 타사 솔루션 사용

Linux 보안에 익숙하지 않은 경우 seccomp 및 Apparmor 프로파일을 생성하고 관리하는 것이 어려울 수 있습니다. 능숙해질 시간이 없는 경우 타사 상용 솔루션을 사용하는 것이 좋습니다. 이들 중 상당수는 Apparmor 및 seccomp와 같은 정적 프로필을 벗어나 기계 학습을 사용하여 의심스러운 활동을 차단하거나 경고하기 시작했습니다. 이러한 몇 가지 솔루션은 도구 ??? 섹션에서 찾을 수 있습니다. 추가 옵션은 컨테이너용 AWS Marketplace에서 확인할 수 있습니다.

seccomp 정책을 작성하기 전에 Linux 기능 추가/삭제 고려

기능에는 시스템 호출로 연결할 수 있는 커널 함수의 다양한 검사가 포함됩니다. 검사에 실패하면 syscall은 일반적으로 오류를 반환합니다. 검사는 특정 시스템 호출이 시작될 때 또는 여러 가지 다른 시스템 호출을 통해 도달할 수 있는 영역(예: 특정 권한 있는 파일에 쓰기)의 커널에서 더 깊이 수행할 수 있습니다. 반면 Seccomp는 실행되기 전에 모든 시스템 호출에 적용되는 시스템 호출 필터입니다. 프로세스는 특정 시스템 호출 또는 특정 시스템 호출에 대한 특정 인수를 실행할 수 있는 권한을 취소할 수 있는 필터를 설정할 수 있습니다.

seccomp를 사용하기 전에 Linux 기능을 추가/제거하면 필요한 제어 기능이 제공되는지 고려합니다. 자세한 내용은 컨테이너에 대한 기능 설정을 참조하세요.

포드 보안 정책(PSPs)을 사용하여 목표를 달성할 수 있는지 확인합니다.

포드 보안 정책은 과도한 복잡성 없이 보안 태세를 개선할 수 있는 다양한 방법을 제공합니다. seccomp 및 Apparmor 프로파일을 빌드하기 전에 PSPs에서 사용할 수 있는 옵션을 살펴봅니다.

> **Warning**
>
> Kubernetes 1.25부터 PSPs 제거되고 포드 보안 승인 컨트롤러로 대체되었습니다. 존재하는 서드 파티 대안에는 OPA/Gatekeeper 및 Kyverno가 포함됩니다. PSPs에서 일반적으로 발견되는 정책을 구현하기 위한 Gatekeeper 제약 조건 및 제약 조건 템플릿 모음은 GitHub의 Gatekeeper 라이브러리 리포지토리에서 가져올 수 있습니다. 또한 전체 포드 보안 표준 컬렉션을 포함하여 Kyverno 정책 라이브러리에서 PSPs를 대체할 수 있습니다.

도구 및 리소스

- 시작하기 전에 알아야 할 7가지 사항
- AppArmor 로더
- 프로파일을 사용하여 노드 설정
- Security Profiles Operator는 사용자가 Kubernetes 클러스터에서 SELinux, seccomp 및 AppArmor를 더 쉽게 사용할 수 있도록 하는 Kubernetes 개선 사항입니다. 실행 중인 워크로드에서 프로파일을 생성하고 포드에 사용할 Kubernetes 노드에 프로파일을 로드하는 기능을 제공합니다.
- Inspektor Gadget을 사용하면 seccomp 프로파일 생성 지원을 포함하여 Kubernetes에서 런타임 동작의 여러 측면을 검사, 추적 및 프로파일링할 수 있습니다.
- Aqua
- 품질
- Stackrox
- Sysdig 보안
- 프리즈마
- SUSE 오픈 소스, 제로 트러스트 컨테이너 보안 플랫폼의 NeuVector는 프로세스 프로파일 규칙 및 파일 액세스 규칙을 제공합니다.

인프라(호스트) 보호

컨테이너 이미지를 보호하는 것이 중요하므로 컨테이너 이미지를 실행하는 인프라를 보호하는 것도 중요합니다. 이 섹션에서는 호스트에 대해 직접 시작된 공격으로 인한 위험을 완화하는 다양한 방법을 살펴봅니다. 이러한 지침은 런타임 보안 섹션에 설명된 지침과 함께 사용해야 합니다.

권장 사항

컨테이너 실행에 최적화된 OS 사용

Linux 컨테이너를 실행하도록 설계된 AWS의 특수 용도 OS인 Flatcar Linux, Project Atomic, RancherOS 및 Bottlerocket을 사용하는 것이 좋습니다. 여기에는 감소된 공격 표면, 부팅 시 확인된 디스크 이미지, SELinux를 사용하여 시행된 권한 경계가 포함됩니다.

또는 Kubernetes 작업자 노드에 EKS 최적화 AMI를 사용합니다. EKS 최적화 AMI는 정기적으로 릴리스되며 컨테이너화된 워크로드를 실행하는 데 필요한 최소한의 OS 패키지 및 바이너리 세트를 포함합니다.

Hashicorp Packer를 사용하여 Red Hat Enterprise Linux에서 실행되는 사용자 지정 Amazon EKS AMI를 빌드하는 데 사용할 수 있는 샘플 구성 스크립트는 Amazon EKS AMI RHEL 빌드 사양을 참조하세요. 이 스크립트를 추가로 활용하여 STIG 준수 EKS 사용자 지정 AMIs.

작업자 노드 OS 업데이트 유지

Bottlerocket과 같은 컨테이너 최적화 호스트 OS를 사용하든 더 크지만 여전히 미니멀리스트인 EKS 최적화 AMIs와 같은 Amazon Machine Image를 사용하든 관계없이 이러한 호스트 OS 이미지를 최신 보안 패치로 최신 상태로 유지하는 것이 좋습니다.

EKS 최적화 AMIs의 경우 변경 사항 및/또는 릴리스 정보 채널을 정기적으로 확인하고 업데이트된 작업자 노드 이미지를 클러스터에 자동으로 롤아웃합니다.

인프라를 변경할 수 없는 것으로 취급하고 작업자 노드 교체 자동화

새 패치 또는 업데이트를 사용할 수 있게 되면 현재 위치 업그레이드를 수행하는 대신 작업자를 교체합니다. 이는 몇 가지 방법으로 접근될 수 있습니다. 그룹의 모든 노드가 최신 AMI로 교체될 때까지 노드를 순차적으로 연결 및 드레이닝할 때 최신 AMI를 사용하여 기존 오토 스케일링 그룹에 인스턴스를 추가할 수 있습니다. 또는 모든 노드가 교체될 때까지 이전 노드 그룹에서 노드를 순차적으로 연결 및 드레이닝하는 동안 새 노드 그룹에 인스턴스를 추가할 수 있습니다. EKS 관리형 노드 그룹은 첫 번째 접근 방식을 사용하며 새 AMI를 사용할 수 있게 되면 콘솔에 메시지를 표시하여 작업자를 업그레이드합니다. `eksctl` 또한 에는 최신 AMI를 사용하여 노드 그룹을 생성하고 인스턴스가 종료되기 전에 노드 그룹에서 포드를 정상적으로 코딩하고 드레이닝하는 메커니즘이 있습니다. 작업자 노드를 교체하기 위해 다른 방법을 사용하기로 결정한 경우 새 업데이트/패치가 릴리스되고 컨트롤 플레인이 업그레이드될 때 작업자를 정기적으로 교체해야 하므로 인적 감독을 최소화하도록 프로세스를 자동화하는 것이 좋습니다.

EKS Fargate를 사용하면 업데이트가 제공되면 AWS가 기본 인프라를 자동으로 업데이트합니다. 이 작업을 원활하게 수행할 수 있는 경우가 많지만 업데이트로 인해 포드가 다시 예약되는 경우가 있을 수 있습니다. 따라서 애플리케이션을 Fargate 포드로 실행할 때 여러 복제본을 사용하여 배포를 생성하는 것이 좋습니다.

kube-bench를 정기적으로 실행하여 Kubernetes에 대한 CIS 벤치마크 준수 확인

kube-bench는 Kubernetes에 대한 CIS 벤치마크를 기준으로 클러스터를 평가하는 Aqua의 오픈 소스 프로젝트입니다. 벤치마크는 비관리형 Kubernetes 클러스터를 보호하기 위한 모범 사례를 설명합니다. CIS Kubernetes 벤치마크는 컨트롤 플레인과 데이터 플레인을 포함합니다. Amazon EKS는 완전 관리형 컨트롤 플레인을 제공하므로 CIS Kubernetes 벤치마크의 모든 권장 사항이 적용되는 것은 아닙니다. 이 범위가 Amazon EKS가 구현되는 방식을 반영하도록 AWS는 CIS Amazon EKS 벤치마크를

생성했습니다. EKS 벤치마크는 EKS 클러스터에 대한 특정 구성 고려 사항이 있는 커뮤니티의 추가 입력과 함께 CIS Kubernetes 벤치마크에서 상속됩니다.

EKS 클러스터에 대해 kube-bench를 실행하는 경우 Aqua Security의 다음 지침을 따르세요. 자세한 내용은 CIS Amazon EKS 벤치마크 소개를 참조하세요.

작업자 노드에 대한 액세스 최소화

호스트로 원격해야 하는 경우 SSH 액세스를 활성화하는 대신 SSM 세션 관리자를 사용합니다. 손실, 복사 또는 공유할 수 있는 SSH 키와 달리 Session Manager를 사용하면 IAM을 사용하여 EC2 인스턴스에 대한 액세스를 제어할 수 있습니다. 또한 인스턴스에서 실행된 명령의 감사 추적 및 로그를 제공합니다.

2020년 8월 19일부터 관리형 노드 그룹은 사용자 지정 AMIs 및 EC2 시작 템플릿을 지원합니다. 이렇게 하면 SSM 에이전트를 AMI에 임베드하거나 작업자 노드가 부트스트래핑될 때 설치할 수 있습니다. 최적화된 AMI 또는 ASG의 시작 템플릿을 수정하지 않으려면 이 예제와 같이 DaemonSet를 사용하여 SSM 에이전트를 설치할 수 있습니다.

SSM 기반 SSH 액세스에 대한 최소 IAM 정책

`AmazonSSMManagedInstanceCore` AWS 관리형 정책에는 SSH 액세스를 피하려는 경우 SSM 세션 관리자/SSM RunCommand에 필요하지 않은 여러 권한이 포함되어 있습니다. 특히 중요한 것은가 역할이 Parameter Store의 모든 파라미터(AWS 관리형 KMS 키가 구성된 SecureStrings 포함)에 액세스할 수 `ssm:GetParameter(s)` 있도록 허용하는 * 권한입니다.

다음 IAM 정책에는 SSM Systems Manager를 통해 노드 액세스를 활성화할 수 있는 최소한의 권한 집합이 포함되어 있습니다.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "EnableAccessViaSSMSessionManager",
      "Effect": "Allow",
      "Action": [
        "ssmmessages:OpenDataChannel",
        "ssmmessages:OpenControlChannel",
        "ssmmessages:CreateDataChannel",
        "ssmmessages:CreateControlChannel",
        "ssm:UpdateInstanceInformation"
      ],
```

```
      "Resource": "*"
    },
    {
      "Sid": "EnableSSMRunCommand",
      "Effect": "Allow",
      "Action": [
        "ssm:UpdateInstanceInformation",
        "ec2messages:SendReply",
        "ec2messages:GetMessages",
        "ec2messages:GetEndpoint",
        "ec2messages:FailMessage",
        "ec2messages:DeleteMessage",
        "ec2messages:AcknowledgeMessage"
      ],
      "Resource": "*"
    }
  ]
}
```

이 정책이 적용되고 Session Manager 플러그인이 설치된 상태에서를 실행할 수 있습니다.

```
aws ssm start-session --target [INSTANCE_ID_OF_EKS_NODE]
```

노드에 액세스합니다.

> **Note**
>
> Session Manager 로깅을 활성화하는 권한 추가를 고려할 수도 있습니다.

프라이빗 서브넷에 작업자 배포

프라이빗 서브넷에 작업자를 배포하면 공격이 자주 발생하는 인터넷에 대한 노출을 최소화할 수 있습니다. 2020년 4월 22일부터 관리형 노드 그룹의 노드에 대한 퍼블릭 IP 주소 할당은 배포된 서브넷에 의해 제어됩니다. 이전에는 관리형 노드 그룹의 노드에 퍼블릭 IP가 자동으로 할당되었습니다. 작업자 노드를 퍼블릭 서브넷에 배포하도록 선택한 경우 제한적인 AWS 보안 그룹 규칙을 구현하여 노출을 제한합니다.

Amazon Inspector를 실행하여 호스트의 노출, 취약성 및 모범 사례 이탈 여부를 평가합니다.

Amazon Inspector를 사용하여 노드에 대한 의도하지 않은 네트워크 액세스와 기본 Amazon EC2 인스턴스의 취약성을 확인할 수 있습니다.

Amazon Inspector는 Amazon EC2 Systems Manager(SSM) 에이전트가 설치되고 활성화된 경우에만 Amazon EC2 인스턴스에 대한 일반적인 취약성 및 노출(CVE) 데이터를 제공할 수 있습니다. 이 에이전트는 EKS 최적화 Amazon Linux AMIs 포함한 여러 Amazon Machine Image(AMI)에 사전 설치되어 있습니다. AMIs SSM 에이전트 상태에 관계없이 모든 Amazon EC2 인스턴스에서 네트워크 연결성 문제를 스캔합니다. Amazon EC2에 대한 스캔 구성에 대한 자세한 내용은 Amazon EC2 인스턴스 스캔을 참조하세요.

> **Important**
>
> Fargate 포드를 실행하는 데 사용되는 인프라에서는 Inspector를 실행할 수 없습니다.

대안

SELinux 실행

> **Note**
>
> Red Hat Enterprise Linux(RHEL), CentOS, Bottlerocket 및 Amazon Linux 2023에서 사용 가능

SELinux는 컨테이너를 서로 격리하고 호스트에서 격리할 수 있는 추가 보안 계층을 제공합니다. SELinux를 사용하면 관리자가 모든 사용자, 애플리케이션, 프로세스 및 파일에 대해 필수 액세스 제어(MAC)를 적용할 수 있습니다. 레이블 집합을 기반으로 특정 리소스에 대해 수행할 수 있는 작업을 제한하는 백스톱이라고 가정합니다. EKS에서 SELinux를 사용하여 컨테이너가 서로의 리소스에 액세스하지 못하도록 할 수 있습니다.

컨테이너 SELinux 정책은 container-selinux 패키지에 정의되어 있습니다. Docker CE에는 Docker(또는 기타 컨테이너 런타임)에서 생성한 프로세스 및 파일이 제한된 시스템 액세스로 실행되도록이 패키지(종속성과 함께)가 필요합니다. 컨테이너는의 별칭인 `container_t` 레이블을 활용합니다`svirt_lxc_net_t`. 이러한 정책은 컨테이너가 호스트의 특정 기능에 액세스하는 것을 효과적으로 방지합니다.

Docker용 SELinux를 구성하면 Docker가 자동으로 워크로드에 유형container_t으로 레이블을 지정하고 각 컨테이너에 고유한 MCS 수준을 부여합니다. 이렇게 하면 컨테이너가 서로 격리됩니다. 더 느슨한 제한이 필요한 경우 파일 시스템의 특정 영역에 컨테이너 권한을 부여하는 자체 프로필을 SElinux에서 생성할 수 있습니다. 이는 컨테이너/포드마다 다른 프로필을 생성할 수 있다는 점에서의 PSPs와 유사합니다. 예를 들어, 제한된 제어 세트가 있는 일반 워크로드에 대한 프로파일과 권한이 있는 액세스가 필요한 사물에 대한 프로파일을 가질 수 있습니다.

SELinux for Containers에는 기본 제한을 수정하도록 구성할 수 있는 옵션 세트가 있습니다. 필요에 따라 다음 SELinux 부울을 활성화하거나 비활성화할 수 있습니다.

부울	기본값	설명
container_connect_any	off	컨테이너가 호스트의 권한 있는 포트에 액세스하도록 허용합니다. 예를 들어, 호스트의 443 또는 80에 포트를 매핑해야 하는 컨테이너가 있는 경우
container_manage_cgroup	off	컨테이너가 cgroup 구성을 관리하도록 허용합니다. 예를 들어 systemd를 실행하는 컨테이너는 이를 활성화해야 합니다.
container_use_cephfs	off	컨테이너가 ceph 파일 시스템을 사용하도록 허용합니다.

기본적으로 컨테이너는에서 대부분의 콘텐츠를 읽/usr거나 실행할 수 있습니다/etc. /var/lib/docker 및 아래의 파일에는 레이블/var/lib/containers이 있습니다container_var_lib_t. 기본값의 전체 목록을 보려면 레이블은 container.fc 파일을 참조하세요.

```
docker container run -it \
  -v /var/lib/docker/image/overlay2/repositories.json:/host/repositories.json \
  centos:7 cat /host/repositories.json
# cat: /host/repositories.json: Permission denied
```

```
docker container run -it \
  -v /etc/passwd:/host/etc/passwd \
  centos:7 cat /host/etc/passwd
# cat: /host/etc/passwd: Permission denied
```

로 레이블이 지정된 파일은 컨테이너에서 쓸 수 있는 유일한 파일container_file_t입니다. 볼륨 마운트를 쓸 수 있도록 하려면 끝에 :z 또는 :Z를 지정해야 합니다.

- :z는 컨테이너가 읽거나 쓸 수 있도록 파일에 레이블을 다시 지정합니다.
- :Z는 컨테이너만 읽고 쓸 수 있도록 파일에 레이블을 다시 지정합니다.

```
ls -Z /var/lib/misc
# -rw-r--r--. root root system_u:object_r:var_lib_t:s0   postfix.aliasesdb-stamp

docker container run -it \
  -v /var/lib/misc:/host/var/lib/misc:z \
  centos:7 echo "Relabeled!"

ls -Z /var/lib/misc
#-rw-r--r--. root root system_u:object_r:container_file_t:s0 postfix.aliasesdb-stamp
```

```
docker container run -it \
  -v /var/log:/host/var/log:Z \
  fluentbit:latest
```

Kubernetes에서는 레이블 재지정이 약간 다릅니다. Docker가 자동으로 파일에 레이블을 다시 지정하는 대신 포드를 실행할 사용자 지정 MCS 레이블을 지정할 수 있습니다. 재라벨링을 지원하는 볼륨은 자동으로 재라벨링되어 액세스할 수 있습니다. MCS 레이블이 일치하는 포드는 볼륨에 액세스할 수 있습니다. 엄격한 격리가 필요한 경우 각 포드에 대해 다른 MCS 레이블을 설정합니다.

```
securityContext:
  seLinuxOptions:
    # Provide a unique MCS label per container
    # You can specify user, role, and type also
    # enforcement based on type and level (svert)
    level: s0:c144:c154
```

이 예제에서는 컨테이너가 액세스할 수 있는 파일에 할당된 MCS 레이블에 s0:c144:c154 해당합니다.

EKS에서는 FluentD와 같이 권한이 있는 컨테이너를 실행할 수 있도록 허용하는 정책을 생성하고 호스트 디렉터리에 레이블을 다시 지정할 필요 없이 호스트의 /var/log에서 읽을 수 있도록 SELinux 정책을 생성할 수 있습니다. 레이블이 동일한 포드는 동일한 호스트 볼륨에 액세스할 수 있습니다.

CentOS 7 및 RHEL 7에 SELinux가 구성된 Amazon EKS용 샘플 AMIs를 구현했습니다. 이러한 AMIs 규제가 엄격한 고객의 요구 사항을 충족하는 샘플 구현을 보여주기 위해 개발되었습니다.

> **Warning**
>
> SELinux는 유형이 제한되지 않은 컨테이너를 무시합니다.

도구 및 리소스

- 온프레미스 애플리케이션을 위한 SELinux Kubernetes RBAC 및 배송 보안 정책
- Kubernetes의 반복적인 강화
- Audit2Allow
- SEAlert
- Udica가 있는 컨테이너에 대한 SELinux 정책 생성은 Linux 기능, 포트 및 탑재 지점에 대한 컨테이너 사양 파일을 살펴보고 컨테이너가 제대로 실행되도록 허용하는 SELinux 규칙 세트를 생성하는 도구를 설명합니다.
- 다양한 규제 요구 사항을 충족하도록 OS를 강화하기 위한 AMI 강화 플레이북
- Keiko Upgrade Manager는 작업자 노드의 교체를 오케스트레이션하는 Intuit의 오픈 소스 프로젝트입니다.
- Sysdig 보안
- eksctl

규정 준수

규정 준수는 AWS와 서비스 소비자 간의 공동 책임입니다. 일반적으로 AWS는 "클라우드 보안"에 대한 책임이 있는 반면, AWS의 사용자는 "클라우드 보안"에 대한 책임이 있습니다. AWS와 해당 사용자가 담당하는 책임을 설명하는 줄은 서비스에 따라 달라집니다. 예를 들어 Fargate를 사용하면 AWS는 데이터 센터, 하드웨어, 가상 인프라(Amazon EC2) 및 컨테이너 런타임(Docker)의 물리적 보안을 관리할 책임이 있습니다. Fargate 사용자는 컨테이너 이미지와 애플리케이션을 보호할 책임이 있습니다. 규정 준수 표준을 준수해야 하는 워크로드를 실행할 때 중요한 고려 사항이 무엇인지 누가 담당하는지 파악합니다.

다음 표에는 다양한 컨테이너 서비스가 준수하는 규정 준수 프로그램이 나와 있습니다.

규정 준수 프로그램	Amazon ECS 오케스트레이터	Amazon EKS 오케스트레이터	ECS Fargate	Amazon ECR
PCI DSS 레벨 1	1	1	1	1
HIPAA 적격	1	1	1	1
SOC I	1	1	1	1
SOC II	1	1	1	1
SOC III	1	1	1	1
ISO 27001:2013	1	1	1	1
ISO 9001:2015	1	1	1	1
ISO 27017:2015	1	1	1	1
ISO 27018:2019	1	1	1	1
IRAP	1	1	1	1
FedRAMP Moderate(동부/서부)	1	1	0	1
FedRAMP High(GovCloud)	1	1	0	1
DOD CC SRG	1	DISA 검토(IL5)	0	1
HIPAA BAA	1	1	1	1
MTCS	1	1	0	1
C5	1	1	0	1
K-ISMS	1	1	0	1

규정 준수 프로그램	Amazon ECS 오케스트레이터	Amazon EKS 오케스트레이터	ECS Fargate	Amazon ECR
ENS High	1	1	0	1
OSPAR	1	1	0	1
HITRUST CSF	1	1	1	1

규정 준수 상태는 시간이 지남에 따라 변경됩니다. 최신 상태는 항상 https://aws.amazon.com/compliance/services-in-scope/ 참조하십시오.

클라우드 인증 모델 및 모범 사례에 대한 자세한 내용은 AWS 백서, 보안 클라우드 채택을 위한 인증 모델을 참조하세요.

왼쪽으로 이동

왼쪽으로 이동하는 개념에는 소프트웨어 개발 수명 주기 초기에 정책 위반 및 오류를 포착하는 것이 포함됩니다. 보안 관점에서 이는 매우 유용할 수 있습니다. 예를 들어 개발자는 애플리케이션이 클러스터에 배포되기 전에 구성 문제를 해결할 수 있습니다. 위와 같은 실수를 포착하면 정책을 위반하는 구성이 배포되는 것을 방지하는 데 도움이 됩니다.

코드형 정책

정책은 허용된 동작 또는 금지된 동작과 같은 동작을 관리하기 위한 규칙 세트로 생각할 수 있습니다. 예를 들어 모든 Dockerfile에 컨테이너가 루트가 아닌 사용자로 실행되도록 하는 사용자 지시문이 포함되어야 한다는 정책이 있을 수 있습니다. 문서로서 이와 같은 정책은 검색하고 적용하기 어려울 수 있습니다. 요구 사항이 변경되면 오래된 것일 수도 있습니다. 코드형 정책(PaC) 솔루션을 사용하면 알려진 위협과 지속적인 위협을 탐지, 방지, 감소 및 대응하는 보안, 규정 준수 및 개인 정보 보호 제어를 자동화할 수 있습니다. 또한 다른 코드 아티팩트와 마찬가지로 정책을 코드화하고 관리하는 메커니즘을 제공합니다. 이 접근 방식의 이점은 DevOps 및 GitOps 전략을 재사용하여 Kubernetes 클러스터 플릿에서 정책을 관리하고 일관되게 적용할 수 있다는 것입니다. PaC 옵션과 PSP의 미래에 대한 자세한 내용은 포드 보안을 참조하세요. PSPs

파이프라인에서 policy-as-code 도구를 사용하여 배포 전에 위반 탐지

- OPA는 CNCF의 일부인 오픈 소스 정책 엔진입니다. 정책 결정을 내리는 데 사용되며 언어 라이브러리 또는 서비스와 같은 다양한 방법으로 실행할 수 있습니다. OPA 정책은 Rego라는 도메인별 언어

(DSL)로 작성됩니다. Kubernetes Dynamic Admission Controller의 일부로 Gatekeeper 프로젝트로 실행되는 경우가 많지만 OPA를 CI/CD 파이프라인에 통합할 수도 있습니다. 이를 통해 개발자는 릴리스 주기 초기에 구성에 대한 피드백을 받을 수 있으므로 나중에 프로덕션에 도달하기 전에 문제를 해결하는 데 도움이 될 수 있습니다. 일반적인 OPA 정책 모음은이 프로젝트의 GitHub 리포지토리에서 찾을 수 있습니다.

- Conftest는 OPA를 기반으로 구축되었으며 Kubernetes 구성을 테스트하기 위한 개발자 중심 경험을 제공합니다.
- Kyverno는 Kubernetes용으로 설계된 정책 엔진입니다. Kyverno를 사용하면 정책이 Kubernetes 리소스로 관리되며 정책을 작성하는 데 새로운 언어가 필요하지 않습니다. 이를 통해 kubectl, git, kustomize와 같은 익숙한 도구를 사용하여 정책을 관리할 수 있습니다. Kyverno 정책은 Kubernetes 리소스를 검증, 변경 및 생성하고 OCI 이미지 공급망 보안을 보장할 수 있습니다. Kyverno CLI를 사용하여 정책을 테스트하고 CI/CD 파이프라인의 일부로 리소스를 검증할 수 있습니다. 모든 Kyverno 커뮤니티 정책은 Kyverno 웹 사이트에서 찾을 수 있으며, Kyverno CLI를 사용하여 파이프라인에서 테스트를 작성하는 예제는 정책 리포지토리를 참조하세요.

도구 및 리소스

- Amazon EKS Security Immersion 워크숍 - 규정 준수
- kube-bench
- docker-bench-security
- AWS Inspector
- Kubernetes 보안 검토 Kubernetes 1.13.4(2019)의 타사 보안 평가
- SUSE 오픈 소스의 NeuVector, 제로 트러스트 컨테이너 보안 플랫폼, 규정 준수 보고 및 사용자 지정 규정 준수 검사 제공

인시던트 대응 및 포렌식

인시던트에 신속하게 대응하는 기능은 위반으로 인한 손상을 최소화하는 데 도움이 될 수 있습니다. 의심스러운 행동을 경고할 수 있는 신뢰할 수 있는 알림 시스템을 갖추는 것이 좋은 인시던트 대응 계획의 첫 번째 단계입니다. 인시던트가 발생하면 영향을 받는 컨테이너를 폐기 및 교체할지 아니면 컨테이너를 격리 및 검사할지 신속하게 결정해야 합니다. 포렌식 조사 및 근본 원인 분석의 일부로 컨테이너를 격리하기로 선택한 경우 다음 활동 세트를 따라야 합니다.

샘플 인시던트 대응 계획

문제가 되는 포드 및 작업자 노드 식별

첫 번째 조치는 손상을 격리하는 것입니다. 먼저 위반이 발생한 위치를 식별하고 해당 포드와 해당 노드를 나머지 인프라에서 격리합니다.

워크로드 이름을 사용하여 문제가 되는 포드 및 작업자 노드 식별

문제가 되는 포드의 이름과 네임스페이스를 알고 있는 경우 다음과 같이 포드를 실행하는 작업자 노드를 식별할 수 있습니다.

```
kubectl get pods <name> --namespace <namespace> -o=jsonpath='{.spec.nodeName}{"\n"}'
```

배포와 같은 워크로드 리소스가 손상된 경우 워크로드 리소스의 일부인 모든 포드가 손상되었을 수 있습니다. 다음 명령을 사용하여 워크로드 리소스의 모든 포드와 해당 포드가 실행 중인 노드를 나열합니다.

```
selector=$(kubectl get deployments <name> \
 --namespace <namespace> -o json | jq -j \
'.spec.selector.matchLabels | to_entries | .[] | "\(.key)=\(.value)"')

kubectl get pods --namespace <namespace> --selector=$selector \
-o json | jq -r '.items[] | "\(.metadata.name) \(.spec.nodeName)"'
```

위 명령은 배포용입니다. 복제본, 상태 저장 세트 등과 같은 다른 워크로드 리소스에 대해 동일한 명령을 실행할 수 있습니다.

서비스 계정 이름을 사용하여 문제가 되는 포드 및 작업자 노드 식별

경우에 따라 서비스 계정이 손상되었음을 식별할 수 있습니다. 식별된 서비스 계정을 사용하는 포드가 손상되었을 수 있습니다. 다음 명령을 사용하여 서비스 계정 및 실행 중인 노드를 사용하여 모든 포드를 식별할 수 있습니다.

```
kubectl get pods -o json --namespace <namespace> | \
    jq -r '.items[] |
    select(.spec.serviceAccount == "<service account name>") |
    "\(.metadata.name) \(.spec.nodeName)"'
```

이미지 및 작업자 노드가 취약하거나 손상된 포드 식별

경우에 따라 클러스터의 포드에서 사용 중인 컨테이너 이미지가 악성이거나 손상된 것을 발견할 수 있습니다. 컨테이너 이미지에 맬웨어가 포함된 것으로 확인되거나 알려진 잘못된 이미지이거나 악용된 CVE가 있는 경우 악의적이거나 손상된 이미지입니다. 손상된 컨테이너 이미지를 사용하는 모든 포드를 고려해야 합니다. 다음 명령을 사용하여 실행 중인 이미지와 노드를 사용하여 포드를 식별할 수 있습니다.

```
IMAGE=<Name of the malicious/compromised image>

kubectl get pods -o json --all-namespaces | \
    jq -r --arg image "$IMAGE" '.items[] |
    select(.spec.containers[] | .image == $image) |
    "\(.metadata.name) \(.metadata.namespace) \(.spec.nodeName)"'
```

포드에 대한 모든 수신 및 송신 트래픽을 거부하는 네트워크 정책을 생성하여 포드를 격리합니다.

모든 트래픽 거부 규칙은 포드에 대한 모든 연결을 분리하여 이미 진행 중인 공격을 중지하는 데 도움이 될 수 있습니다. 다음 네트워크 정책은 레이블이 인 포드에 적용됩니다app=web.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector:
    matchLabels:
      app: web
  policyTypes:
  - Ingress
  - Egress
```

> **Important**
>
> 공격자가 기본 호스트에 액세스한 경우 네트워크 정책이 효과가 없는 것으로 입증될 수 있습니다. 가 발생한 것으로 의심되는 경우 AWS 보안 그룹을 사용하여 손상된 호스트를 다른 호스트에서 격리할 수 있습니다. 호스트의 보안 그룹을 변경할 때 호스트에서 실행되는 모든 컨테이너에 영향을 미친다는 점에 유의하세요.

필요한 경우 포드 또는 작업자 노드에 할당된 임시 보안 자격 증명 취소

작업자 노드에 포드가 다른 AWS 리소스에 액세스할 수 있는 IAM 역할이 할당된 경우 해당 역할을 인스턴스에서 제거하여 공격으로 인한 추가 손상을 방지합니다. 마찬가지로 포드에 IAM 역할이 할당된 경우 다른 워크로드에 영향을 미치지 않으면서 역할에서 IAM 정책을 안전하게 제거할 수 있는지 평가합니다.

작업자 노드 연결

영향을 받는 작업자 노드를 연결하면 영향을 받는 노드에 포드를 예약하지 않도록 스케줄러에 알립니다. 이렇게 하면 다른 워크로드를 중단하지 않고 포렌식 연구를 위한 노드를 제거할 수 있습니다.

> **Note**
>
> 이 지침은 각 Fargate 포드가 자체 샌드박스 환경에서 실행되는 Fargate에는 적용되지 않습니다. 차단 대신 모든 수신 및 송신 트래픽을 거부하는 네트워크 정책을 적용하여 영향을 받는 Fargate 포드를 격리합니다.

영향을 받는 작업자 노드에서 종료 방지 활성화

공격자는 영향을 받는 노드를 종료하여 잘못된 행동을 지우려고 할 수 있습니다. 종료 방지 기능을 활성화하면 이를 방지할 수 있습니다. 인스턴스 스케일 인 보호는 스케일 인 이벤트로부터 노드를 보호합니다.

> **Warning**
>
> 스팟 인스턴스에서는 종료 방지를 활성화할 수 없습니다.

문제가 되는 포드/노드에 활성 조사의 일부임을 나타내는 레이블을 지정합니다.

이는 조사가 완료될 때까지 클러스터 관리자가 영향을 받는 포드/노드를 변조하지 않도록 경고하는 역할을 합니다.

작업자 노드에서 휘발성 아티팩트 캡처

- 운영 체제 메모리를 캡처합니다. 그러면 Docker 데몬(또는 기타 컨테이너 런타임)과 컨테이너당 하위 프로세스가 캡처됩니다. 이는 LiME 및 변동성과 같은 도구를 사용하거나 그 위에 구축된

Amazon EC2용 Automated Forensics Orchestrator와 같은 상위 수준 도구를 통해 수행할 수 있습니다.

- 실행 중인 프로세스와 열린 포트의 netstat 트리 덤프를 수행합니다. 그러면 컨테이너당 Docker 데몬과 하위 프로세스가 캡처됩니다.
- 증거를 변경하기 전에 명령을 실행하여 컨테이너 수준 상태를 저장합니다. 컨테이너 런타임의 기능을 사용하여 현재 실행 중인 컨테이너에 대한 정보를 캡처할 수 있습니다. 예를 들어 Containerd에서는 다음을 수행할 수 있습니다.
  - `crictl ps` 실행 중인 프로세스의 경우.
  - `crictl logs CONTAINER` 데몬 수준 보관 로그의 경우.

  (`docker`예: `nerdctl inspect`) 대신 nerdctl CLI를 사용하여 컨테이너화된 에서도 동일한 작업을 수행할 수 있습니다. 컨테이너 런타임에 따라 몇 가지 추가 명령을 사용할 수 있습니다. 예를 들어 Docker는 컨테이너 파일 시스템의 변경 사항을 보거나 휘발성 메모리(RAM)`docker checkpoint`를 포함한 모든 컨테이너 상태를 저장`docker diff`해야 합니다. 컨테이너형 또는 CRI-O 런타임과 유사한 기능에 대한 설명은 이 Kubernetes 블로그 게시물을 참조하세요.
- 포렌식 캡처를 위해 컨테이너를 일시 중지합니다.
- 인스턴스의 EBS 볼륨을 스냅샷합니다.

손상된 포드 또는 워크로드 리소스 재배포

포렌식 분석을 위해 데이터를 수집한 후에는 손상된 포드 또는 워크로드 리소스를 재배포할 수 있습니다.

먼저 손상된 취약성에 대한 수정 사항을 롤아웃하고 새 대체 포드를 시작합니다. 그런 다음 취약한 포드를 삭제합니다.

취약한 포드가 상위 수준 Kubernetes 워크로드 리소스(예: 배포 또는 DaemonSet)에 의해 관리되는 경우 삭제하면 새 포드가 예약됩니다. 따라서 취약한 포드가 다시 시작됩니다. 이 경우 취약성을 수정한 후 새 대체 워크로드 리소스를 배포해야 합니다. 그런 다음 취약한 워크로드를 삭제해야 합니다.

권장 사항

AWS 보안 인시던트 대응 백서 검토

이 섹션에서는 의심되는 보안 침해를 처리하기 위한 몇 가지 권장 사항과 함께 간략한 개요를 제공하지만이 주제는 백서인 AWS 보안 인시던트 대응에서 자세히 다룹니다.

보안 게임 데이 연습

보안 실무자를 빨간색과 파란색의 두 팀으로 나눕니다. 레드 팀은 취약성에 대해 다양한 시스템을 탐색하는 데 집중하고 블루 팀은 취약성에 대해 방어할 책임이 있습니다. 보안 실무자가 부족하여 별도의 팀을 만들 수 없는 경우 Kubernetes 악용에 대해 알고 있는 외부 엔터티를 고용하는 것이 좋습니다.

Kubesploit은 게임 데이를 수행하는 데 사용할 수 있는 CyberArk의 침투 테스트 프레임워크입니다. 클러스터의 취약성을 스캔하는 다른 도구와 달리 kubesploit은 실제 공격을 시뮬레이션합니다. 이를 통해 블루 팀은 공격에 대한 대응을 연습하고 효과를 측정할 수 있습니다.

클러스터에 대해 침투 테스트 실행

자체 클러스터를 주기적으로 공격하면 취약성과 잘못된 구성을 발견하는 데 도움이 될 수 있습니다. 시작하기 전에 클러스터에 대해 테스트를 수행하기 전에 침투 테스트 지침을 따르십시오.

도구 및 리소스

- kube-hunter, Kubernetes용 침투 테스트 도구.
- Gremlin은 애플리케이션 및 인프라에 대한 공격을 시뮬레이션하는 데 사용할 수 있는 카오스 엔지니어링 툴킷입니다.
- Kubernetes 설치 공격 및 방어
- kubesploit
- SUSE 오픈 소스, 제로 트러스트 컨테이너 보안 플랫폼의 NeuVector는 취약성 및 위험 보고와 보안 이벤트 알림을 제공합니다.
- 고급 영구 위협
- Kubernetes 실제 공격 및 방어
- RBAC 권한을 악용하여 Kubernetes 클러스터 손상

이미지 보안

컨테이너 이미지를 공격에 대한 첫 번째 방어선으로 보아야 합니다. 이미지가 안전하지 않고 잘못 구성되어 있으면 공격자가 컨테이너의 경계를 벗어나 호스트에 액세스할 수 있습니다. 호스트에서 공격자는 민감한 정보에 액세스하거나 클러스터 내부 또는 AWS 계정으로 내부적으로 이동할 수 있습니다. 다음 모범 사례는 이러한 일이 발생할 위험을 완화하는 데 도움이 됩니다.

권장 사항

최소 이미지 생성

먼저 컨테이너 이미지에서 불필요한 바이너리를 모두 제거합니다. Dockerhub의 익숙하지 않은 이미지를 사용하는 경우 각 컨테이너 계층의 콘텐츠를 표시할 수 있는 Dive와 같은 애플리케이션을 사용하여 이미지를 검사합니다. SETUID 및 SETGID 비트가 있는 바이너리는 권한을 에스컬레이션하는 데 사용할 수 있으므로 모두 제거하고 악의적인 목적으로 사용할 수 있는 nc 및 curl과 같은 모든 셸 및 유틸리티를 제거하는 것이 좋습니다. 다음 명령을 사용하여 SETUID 및 SETGID 비트가 있는 파일을 찾을 수 있습니다.

```
find / -perm /6000 -type f -exec ls -ld {} \;
```

이러한 파일에서 특수 권한을 제거하려면 컨테이너 이미지에 다음 명령을 추가합니다.

```
RUN find / -xdev -perm /6000 -type f -exec chmod a-s {} \; || true
```

구체적으로 이는 이미지의 펑크를 해제하는 것으로 알려져 있습니다.

다단계 빌드 사용

다단계 빌드를 사용하면 최소한의 이미지를 만들 수 있습니다. 다단계 빌드는 연속 통합 주기의 일부를 자동화하는 데 사용되는 경우가 많습니다. 예를 들어 다단계 빌드를 사용하여 소스 코드를 린트하거나 정적 코드 분석을 수행할 수 있습니다. 이를 통해 개발자는 파이프라인이 실행될 때까지 기다리지 않고 거의 즉각적인 피드백을 받을 수 있습니다. 다단계 빌드는 컨테이너 레지스트리로 푸시되는 최종 이미지의 크기를 최소화할 수 있으므로 보안 관점에서 매력적입니다. 빌드 도구 및 기타 불필요한 바이너리가 없는 컨테이너 이미지는 이미지의 공격 표면을 줄여 보안 태세를 향상합니다. 다단계 빌드에 대한 자세한 내용은 Docker의 다단계 빌드 설명서를 참조하세요.

컨테이너 이미지에 대한 소프트웨어 재료표(SBOMs) 생성

"소프트웨어 재료표"(SBOM)는 컨테이너 이미지를 구성하는 소프트웨어 아티팩트의 중첩 인벤토리입니다. SBOM은 소프트웨어 보안 및 소프트웨어 공급망 위험 관리의 핵심 구성 요소입니다. SBOMS를 생성하여 중앙 리포지토리에 저장하고 SBOMs에서 취약성을 검사하면 다음과 같은 문제를 해결하는 데 도움이 됩니다.

- 가시성: 컨테이너 이미지를 구성하는 구성 요소를 이해합니다. 중앙 리포지토리에 저장하면 배포 후에도 언제든지 SBOMs을 감사하고 스캔하여 제로데이 취약성과 같은 새로운 취약성을 감지하고 대응할 수 있습니다.

- 증명 확인: 아티팩트의 출처와 방법에 대한 기존 가정이 사실이며 빌드 또는 전송 프로세스 중에 아티팩트 또는 관련 메타데이터가 변조되지 않았는지 확인합니다.
- 신뢰성: 주어진 아티팩트와 해당 콘텐츠를 신뢰할 수 있도록 하여 의도한 작업을 수행할 수 있도록 보장합니다. 즉,가 목적에 적합합니다. 여기에는 코드가 실행하기에 안전한지 판단하고 코드 실행과 관련된 위험에 대해 정보에 입각한 결정을 내리는 것이 포함됩니다. 신뢰할 수 있는 방법은 증명된 SBOM 및 증명된 CVE 스캔 보고서와 함께 증명된 파이프라인 실행 보고서를 생성하여이 이미지가 안전한 구성 요소를 포함하는 안전한 수단(파이프라인)을 통해 사실적으로 생성되었는지 확인하는 것입니다.
- 종속성 신뢰 확인: 아티팩트의 종속성 트리에서 사용하는 아티팩트의 신뢰성과 출처를 재귀적으로 확인합니다. SBOMs의 드리프트는 신뢰할 수 없는 무단 종속성, 침입 시도를 포함한 악의적인 활동을 탐지하는 데 도움이 될 수 있습니다.

다음 도구를 사용하여 SBOM을 생성할 수 있습니다.

- Amazon Inspector를 사용하여 SBOMs.
- Anchore의 Syft는 SBOM 생성에도 사용할 수 있습니다. 더 빠른 취약성 스캔을 위해 컨테이너 이미지에 대해 생성된 SBOM을 스캔할 입력으로 사용할 수 있습니다. 그런 다음 검토 및 감사 목적으로 Amazon ECR과 같은 중앙 OCI 리포지토리로 이미지를 푸시하기 전에 SBOM 및 스캔 보고서가 증명되고 이미지에 연결됩니다.

CNCF 소프트웨어 공급망 모범 사례 가이드를 검토하여 소프트웨어 공급망을 보호하는 방법에 대해 자세히 알아봅니다.

이미지에 취약성이 있는지 정기적으로 스캔

가상 머신과 마찬가지로 컨테이너 이미지에는 취약성이 있는 바이너리 및 애플리케이션 라이브러리가 포함되어 있거나 시간이 지남에 따라 취약성이 발생할 수 있습니다. 악용을 방지하는 가장 좋은 방법은 이미지 스캐너로 이미지를 정기적으로 스캔하는 것입니다. Amazon ECR에 저장된 이미지는 푸시 또는 온디맨드(24시간 동안 한 번)로 스캔할 수 있습니다. ECR은 현재 기본 및 향상된 두 가지 유형의 스캔을 지원합니다. 기본 스캔은 오픈 소스 이미지 스캔 솔루션인 Clair를 무료로 활용합니다. 고급 스캔은 Amazon Inspector를 사용하여 추가 비용을 지불하고 자동 연속 스캔을 제공합니다. 이미지가 스캔되면 결과는 EventBridge의 ECR에 대한 이벤트 스트림에 로깅됩니다. ECR 콘솔 내에서 스캔 결과를 볼 수도 있습니다. 높음 또는 중요 취약성이 있는 이미지는 삭제하거나 다시 빌드해야 합니다. 배포된 이미지에서 취약성이 발견되면 가능한 한 빨리 교체해야 합니다.

환경을 안전하게 유지하려면 취약성이 있는 이미지가 어디에 배포되었는지 알아야 합니다. 이미지 추적 솔루션을 직접 구축할 수 있지만 다음과 같이이 기능과 기타 고급 기능을 즉시 제공하는 몇 가지 상용 제품이 이미 있습니다.

- Grype
- Palo Alto - Prisma Cloud(Twistcli)
- Aqua
- 쿠베이
- 트리비
- Snyk

Kubernetes 검증 웹후크를 사용하여 이미지에 심각한 취약성이 없는지 확인할 수도 있습니다. 검증 웹후크는 Kubernetes API 이전에 호출됩니다. 일반적으로 웹후크에 정의된 검증 기준을 준수하지 않는 요청을 거부하는 데 사용됩니다. 이는 ECR describeImageScanFindings API를 호출하여 포드가 중요한 취약성이 있는 이미지를 가져오는지 여부를 확인하는 서버리스 웹후크의 예입니다. 취약성이 발견되면 포드가 거부되고 CVEs 목록이 포함된 메시지가 이벤트로 반환됩니다.

증명을 사용하여 아티팩트 무결성 검증

증명은 파이프라인 실행, SBOM 또는 취약성 스캔 보고서와 같은 '예측' 또는 컨테이너 이미지와 같은 다른 사물에 대해 참인 무언가를 주장하는 암호로 서명된 '문장'입니다.

증명은 사용자가 아티팩트가 소프트웨어 공급망의 신뢰할 수 있는 소스에서 비롯되었는지 검증하는 데 도움이 됩니다. 예를 들어, 해당 이미지에 포함된 모든 소프트웨어 구성 요소 또는 종속성을 모르고 컨테이너 이미지를 사용할 수 있습니다. 그러나 컨테이너 이미지의 생산자가 존재하는 소프트웨어에 대해 말하는 것을 신뢰하는 경우 생산자의 증명을 사용하여 해당 아티팩트에 의존할 수 있습니다. 즉, 분석을 수행하는 대신 워크플로에서 아티팩트를 안전하게 사용할 수 있습니다.

- AWS Signer 또는 Sigstore 공동 서명을 사용하여 증명을 생성할 수 있습니다.
- Kyverno와 같은 Kubernetes 승인 컨트롤러를 사용하여 증명을 확인할 수 있습니다.
- 컨테이너 이미지에 증명 생성 및 연결 등의 주제와 함께 오픈 소스 도구를 사용하는 AWS의 소프트웨어 공급망 관리 모범 사례에 대해 자세히 알아보려면이 워크숍을 참조하세요.

ECR 리포지토리에 대한 IAM 정책 생성

오늘날 조직에서 공유 AWS 계정 내에서 여러 개발 팀을 독립적으로 운영하는 것은 드문 일이 아닙니다. 이러한 팀이 자산을 공유할 필요가 없는 경우 각 팀이 상호 작용할 수 있는 리포지토리에 대한 액세

스를 제한하는 IAM 정책 세트를 생성할 수 있습니다. 이를 구현하는 좋은 방법은 ECR 네임스페이스를 사용하는 것입니다. 네임스페이스는 유사한 리포지토리를 그룹화하는 방법입니다. 예를 들어, 팀 A의 모든 레지스트리에는 team-a/라는 접두사를 붙일 수 있는 반면, 팀 B의 레지스트리에는 team-b/ 접두사를 사용할 수 있습니다. 액세스를 제한하는 정책은 다음과 같을 수 있습니다.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowPushPull",
      "Effect": "Allow",
      "Action": [
        "ecr:GetDownloadUrlForLayer",
        "ecr:BatchGetImage",
        "ecr:BatchCheckLayerAvailability",
        "ecr:PutImage",
        "ecr:InitiateLayerUpload",
        "ecr:UploadLayerPart",
        "ecr:CompleteLayerUpload"
      ],
      "Resource": [
        "arn:aws:ecr:us-east-1:123456789012:repository/team-a/*"
      ]
    }
  ]
}
```

ECR 프라이빗 엔드포인트 사용 고려

ECR API에는 퍼블릭 엔드포인트가 있습니다. 따라서 IAM에서 요청을 인증하고 승인하는 한 인터넷에서 ECR 레지스트리에 액세스할 수 있습니다. 클러스터 VPC에 인터넷 게이트웨이(IGW)가 없는 샌드박스 환경에서 운영해야 하는 사용자의 경우 ECR에 대한 프라이빗 엔드포인트를 구성할 수 있습니다. 프라이빗 엔드포인트를 생성하면 인터넷을 통해 트래픽을 라우팅하는 대신 프라이빗 IP 주소를 통해 ECR API에 비공개로 액세스할 수 있습니다. 이 주제에 대한 자세한 내용은 Amazon ECR 인터페이스 VPC 엔드포인트를 참조하세요.

ECR에 대한 엔드포인트 정책 구현

의 기본 엔드포인트 정책은 리전 내의 모든 ECR 리포지토리에 대한 액세스를 허용합니다. 이렇게 하면 공격자/내부자가 데이터를 컨테이너 이미지로 패키징하고 다른 AWS 계정의 레지스트리로 푸시하여 데이터를 유출할 수 있습니다. 이 위험을 완화하려면 ECR 리포지토리에 대한 API 액세스를 제한하는

엔드포인트 정책을 생성해야 합니다. 예를 들어 다음 정책은 계정의 모든 AWS 원칙이에 대해 모든 작업을 수행하고 ECR 리포지토리만 수행하도록 허용합니다.

```
{
  "Statement": [
    {
      "Sid": "LimitECRAccess",
      "Principal": "*",
      "Action": "*",
      "Effect": "Allow",
      "Resource": "arn:aws:ecr:<region>:<account_id>:repository/*"
    }
  ]
}
```

AWS Organization에 속하지 않는 IAM 원칙에 따라 이미지 푸시/풀링을 방지하는 새 `PrincipalOrgID` 속성을 사용하는 조건을 설정하여 이를 더욱 개선할 수 있습니다. 자세한 내용은 aws:PrincipalOrgID를 참조하세요. `com.amazonaws.<region>.ecr.dkr` 및 `com.amazonaws.<region>.ecr.api` 엔드포인트 모두에 동일한 정책을 적용하는 것이 좋습니다. EKS는 ECR에서 kube-proxy, coredns 및 aws-node에 대한 이미지를 가져오므로 엔드포인트 정책의 리소스 `602401143452.dkr.ecr.us-west-2.amazonaws.com/*` 목록에 레지스트리의 계정 ID를 추가하거나 정책을 변경하여에서 가져오기를 허용하고 계정 ID에 대한 푸시를 제한해야 합니다. 아래 표에는 EKS 이미지가 벤딩되는 AWS 계정과 클러스터 리전 간의 매핑이 나와 있습니다.

계정 번호	리전
602401143452	아래 나열된 리전을 제외한 모든 상용 리전
—	—
800184023465	ap-east-1 - 아시아 태평양(홍콩)
558608220178	me-south-1 - 중동(바레인)
918309763551	cn-north-1 - 중국(베이징)
961992271922	cn-northwest-1 - 중국(닝샤)

엔드포인트 정책 사용에 대한 자세한 내용은 VPC 엔드포인트 정책을 사용하여 Amazon ECR 액세스 제어를 참조하세요.

ECR에 대한 수명 주기 정책 구현

NIST 애플리케이션 컨테이너 보안 가이드는 "레지스트리의 오래된 이미지"의 위험에 대해 경고하며, 시간이 지남에 따라 소프트웨어 패키지가 취약하고 out-of-date 이미지를 제거하여 우발적인 배포 및 노출을 방지해야 합니다. 각 ECR 리포지토리는 이미지가 만료되는 시점에 대한 규칙을 설정하는 수명 주기 정책을 가질 수 있습니다. AWS 공식 설명서에서는 테스트 규칙을 설정하고 평가하여 적용하는 방법을 설명합니다. 공식 문서에는 리포지토리에서 이미지를 필터링하는 다양한 방법을 보여주는 몇 가지 수명 주기 정책 예제가 있습니다.

- 이미지 연령 또는 개수를 기준으로 필터링
- 태그가 지정되거나 지정되지 않은 이미지를 기준으로 필터링
- 여러 규칙 또는 단일 규칙에서 이미지 태그를 기준으로 필터링

???+ 경고 장기 실행 애플리케이션의 이미지가 ECR에서 제거되면 애플리케이션을 가로로 재배포하거나 확장할 때 이미지 가져오기 오류가 발생할 수 있습니다. 이미지 수명 주기 정책을 사용할 때는 배포 및 배포에서 참조하는 이미지를 최신 상태로 유지하고 릴리스/배포 빈도를 설명하는 [이미지] 만료 규칙을 항상 생성하는 데 적합한 CI/CD 관행이 있는지 확인합니다.

큐레이팅한 이미지 세트 생성

개발자가 자체 이미지를 생성하도록 허용하는 대신 조직의 다양한 애플리케이션 스택에 대해 심사된 이미지 세트를 생성하는 것이 좋습니다. 이렇게 하면 개발자는 Dockerfile 작성 방법을 배울 필요 없이 코드 작성에 집중할 수 있습니다. 변경 사항이 마스터로 병합되면 CI/CD 파이프라인은 자산을 자동으로 컴파일하고 아티팩트 리포지토리에 저장한 다음 ECR과 같은 도커 레지스트리로 푸시하기 전에 아티팩트를 적절한 이미지에 복사할 수 있습니다. 최소한 개발자가 자체 Dockerfiles를 생성할 기본 이미지 세트를 생성해야 합니다. Dockerhub에서 이미지를 가져오지 않는 것이 가장 좋습니다. 이미지에 무엇이 있는지 항상 알지는 못하며 상위 1000개 이미지 https://www.kennasecurity.com/blog/one-fifth-of-the-most-used-docker-containers-have-at-least-one-critical-vulnerability/ 중 2/5에 취약성이 있기 때문입니다. 이러한 이미지 및 취약성 목록은 여기에서 확인할 수 있습니다.

루트가 아닌 사용자로 실행되도록 Dockerfiles에 사용자 지시문 추가

포드 보안 섹션에서 언급했듯이 컨테이너를 루트로 실행하지 않아야 합니다. 이를 podSpec의 일부로 구성할 수 있지만 Dockerfiles에 `USER` 대한 명령을 사용하는 것이 좋습니다. `USER` 명령은 `USER` 명령 뒤에 나타나는 `RUN`, `ENTRYPOINT`또는 `CMD` 명령을 실행할 때 사용할 UID를 설정합니다.

Dockerfiles 린트

린팅을 사용하여 Dockerfile이 USER 지시문 포함 또는 모든 이미지에 태그를 지정해야 하는 요구 사항과 같은 사전 정의된 지침 세트를 준수하는지 확인할 수 있습니다. dockerfile_lint는 일반적인 모범 사례를 확인하고 Dockerfile 린팅을 위한 자체 규칙을 구축하는 데 사용할 수 있는 규칙 엔진을 포함하는 RedHat의 오픈 소스 프로젝트입니다. 규칙을 위반하는 Dockerfile이 포함된 빌드는 자동으로 실패한다는 점에서 CI 파이프라인에 통합할 수 있습니다.

스크래치에서 이미지 빌드

이미지를 빌드할 때 컨테이너 이미지의 공격 표면을 줄이는 것이 기본 조준이어야 합니다. 이를 위한 이상적인 방법은 취약성을 악용하는 데 사용할 수 있는 바이너리가 없는 최소한의 이미지를 생성하는 것입니다. 다행히 Docker에는에서 이미지를 생성하는 메커니즘이 있습니다`scratch`. Go와 같은 언어를 사용하면 정적 연결 바이너리를 생성하고이 예제와 같이 Dockerfile에서 참조할 수 있습니다.

```
############################
# STEP 1 build executable binary
############################
FROM golang:alpine AS builder# Install git.
# Git is required for fetching the dependencies.
RUN apk update && apk add --no-cache gitWORKDIR $GOPATH/src/mypackage/myapp/COPY . . #
 Fetch dependencies.
# Using go get.
RUN go get -d -v# Build the binary.
RUN go build -o /go/bin/hello

############################
# STEP 2 build a small image
############################
FROM scratch# Copy our static executable.
COPY --from=builder /go/bin/hello /go/bin/hello# Run the hello binary.
ENTRYPOINT ["/go/bin/hello"]
```

이렇게 하면 애플리케이션만으로 구성된 컨테이너 이미지가 생성되므로 매우 안전합니다.

ECR에서 변경 불가능한 태그 사용

변경 불가능한 태그는 이미지 리포지토리로 푸시할 때마다 이미지 태그를 업데이트하도록 강제합니다. 이렇게 하면 공격자가 이미지의 태그를 변경하지 않고도 악성 버전으로 이미지를 덮어쓰지 못하게 할 수 있습니다. 또한 이미지를 쉽고 고유하게 식별할 수 있는 방법을 제공합니다.

이미지, SBOMs, 파이프라인 실행 및 취약성 보고서에 서명

Docker가 처음 도입되었을 때 컨테이너 이미지를 확인하기 위한 암호화 모델은 없었습니다. v2에서는 Docker가 이미지 매니페스트에 다이제스트를 추가했습니다. 이렇게 하면 이미지의 구성을 해시하고 해시를 사용하여 이미지의 ID를 생성할 수 있습니다. 이미지 서명이 활성화되면 Docker 엔진은 매니페스트의 서명을 확인하여 콘텐츠가 신뢰할 수 있는 소스에서 생성되었고 변조가 발생하지 않았는지 확인합니다. 각 계층을 다운로드한 후 엔진은 계층의 다이제스트를 확인하여 콘텐츠가 매니페스트에 지정된 콘텐츠와 일치하는지 확인합니다. 이미지 서명은 이미지와 연결된 디지털 서명을 확인하여 보안 공급망을 효과적으로 생성할 수 있습니다.

AWS Signer 또는 Sigstore Cosign을 사용하여 컨테이너 이미지에 서명하고, SBOMs, 취약성 스캔 보고서 및 파이프라인 실행 보고서에 대한 증명을 생성할 수 있습니다. 이러한 증명은 이미지의 신뢰성과 무결성을 보장하고, 실제로 이미지는 간섭이나 변조 없이 신뢰할 수 있는 파이프라인에서 생성되며, 이미지 게시자가 검증하고 신뢰할 수 있는 (SBOM에서) 문서화된 소프트웨어 구성 요소만 포함한다는 것을 보장합니다. 이러한 증명은 컨테이너 이미지에 연결하고 리포지토리로 푸시할 수 있습니다.

다음 섹션에서는 감사 및 승인 컨트롤러 확인에 대해 증명된 아티팩트를 사용하는 방법을 알아봅니다.

Kubernetes 승인 컨트롤러를 사용한 이미지 무결성 확인

동적 승인 컨트롤러를 사용하여 대상 Kubernetes 클러스터에 이미지를 배포하기 전에 이미지 서명, 증명된 아티팩트를 자동으로 확인하고 아티팩트의 보안 메타데이터가 승인 컨트롤러 정책을 준수하는 경우에만 배포를 허용할 수 있습니다.

예를 들어 이미지의 서명, 증명된 SBOM, 증명된 파이프라인 실행 보고서 또는 증명된 CVE 스캔 보고서를 암호화 방식으로 확인하는 정책을 작성할 수 있습니다. 정책의 조건을 작성하여 보고서의 데이터를 확인할 수 있습니다. 예를 들어 CVE 스캔에는 중요한 CVEs 없어야 합니다. 배포는 이러한 조건을 충족하는 이미지에만 허용되며 다른 모든 배포는 승인 컨트롤러에 의해 거부됩니다.

승인 컨트롤러의 예는 다음과 같습니다.

- 키버노
- OPA 게이트키퍼
- 포티에리스
- Ratify
- 크라티스
- Grafeas 자습서

- ™

컨테이너 이미지의 패키지 업데이트

이미지`apt-get update && apt-get upgrade`의 패키지를 업그레이드하려면 Dockerfiles에 RUN을 포함해야 합니다. 업그레이드하려면 루트로를 실행해야 하지만 이는 이미지 빌드 단계에서 발생합니다. 애플리케이션은 루트로 실행할 필요가 없습니다. 업데이트를 설치한 다음 사용자 지시문을 사용하여 다른 사용자로 전환할 수 있습니다. 기본 이미지가 루트가 아닌 사용자로 실행되는 경우 루트 및 백으로 전환합니다. 최신 보안 업데이트를 설치하기 위해 기본 이미지의 유지 관리에만 의존하지 마세요.

를 실행`apt-get clean`하여에서 설치 관리자 파일을 삭제합니다`/var/cache/apt/archives/`. 패키지를 설치`rm -rf /var/lib/apt/lists/*`한 후를 실행할 수도 있습니다. 그러면 인덱스 파일 또는 설치할 수 있는 패키지 목록이 제거됩니다. 이러한 명령은 패키지 관리자마다 다를 수 있습니다. 예제:

```
RUN apt-get update && apt-get install -y \
    curl \
    git \
    libsqlite3-dev \
    && apt-get clean && rm -rf /var/lib/apt/lists/*
```

도구 및 리소스

- Amazon EKS Security Immersion 워크숍 - 이미지 보안
- docker-slim 안전한 최소 이미지 빌드
- dockle Dockerfile이 보안 이미지 생성 모범 사례에 부합하는지 확인
- Dockerfile용 dockerfile-lint 규칙 기반 린터
- hadolint 스마트 dockerfile 린터
- 게이트키퍼 및 OPA A 정책 기반 승인 컨트롤러
- Kyverno Kubernetes 네이티브 정책 엔진
- in-toto 사용자가 공급망의 단계를 수행할 계획인지, 올바른 액터가 단계를 수행했는지 확인할 수 있도록 허용합니다.
- 공증 컨테이너 이미지에 서명하기 위한 프로젝트
- 공증 v2

- Grafeas 소프트웨어 공급망을 감사하고 관리하기 위한 개방형 아티팩트 메타데이터 API
- SUSE 오픈 소스, 제로 트러스트 컨테이너 보안 플랫폼의 NeuVector는 취약성, 보안 암호 및 규정 준수에 대한 컨테이너, 이미지 및 레지스트리 스캔을 제공합니다.

다중 계정 전략

AWS는 다중 계정 전략과 AWS 조직을 사용하여 비즈니스 애플리케이션과 데이터를 격리하고 관리할 것을 권장합니다. 다중 계정 전략을 사용하면 다음과 같은 많은 이점이 있습니다.

- AWS API 서비스 할당량이 증가했습니다. 할당량은 AWS 계정에 적용되며 워크로드에 여러 계정을 사용하면 워크로드에 사용할 수 있는 전체 할당량이 증가합니다.
- 더 간단한 Identity and Access Management(IAM) 정책. 워크로드와 워크로드를 지원하는 운영자가 자신의 AWS 계정에만 액세스할 수 있도록 허용하면 최소 권한 원칙을 달성하기 위해 세분화된 IAM 정책을 작성하는 시간이 단축됩니다.
- AWS 리소스 격리 개선. 설계상 계정 내에 프로비저닝된 모든 리소스는 다른 계정에 프로비저닝된 리소스와 논리적으로 격리됩니다. 이 격리 경계는 애플리케이션 관련 문제, 잘못된 구성 또는 악의적인 작업의 위험을 제한하는 방법을 제공합니다. 한 계정 내에서 문제가 발생하면 다른 계정에 포함된 워크로드의 영향을 줄이거나 제거할 수 있습니다.
- AWS 다중 계정 전략 백서에 설명된 추가 이점

다음 섹션에서는 중앙 집중식 또는 분산형 EKS 클러스터 접근 방식을 사용하여 EKS 워크로드에 대한 다중 계정 전략을 구현하는 방법을 설명합니다.

다중 테넌트 클러스터에 대한 다중 워크로드 계정 전략 계획

다중 계정 AWS 전략에서 S3 버킷, ElastiCache 클러스터 및 DynamoDB 테이블과 같은 특정 워크로드에 속하는 리소스는 모두 해당 워크로드에 대한 모든 리소스가 포함된 AWS 계정에 생성됩니다. 이를 워크로드 계정이라고 하며 EKS 클러스터는 클러스터 계정이라고 하는 계정에 배포됩니다. 클러스터 계정은 다음 섹션에서 살펴볼 것입니다. 전용 워크로드 계정에 리소스를 배포하는 것은 kubernetes 리소스를 전용 네임스페이스에 배포하는 것과 유사합니다.

그런 다음 소프트웨어 개발 수명 주기 또는 적절한 경우 기타 요구 사항에 따라 워크로드 계정을 추가로 분류할 수 있습니다. 예를 들어 지정된 워크로드에는 프로덕션 계정, 개발 계정 또는 특정 리전에서 해당 워크로드의 인스턴스를 호스팅하는 계정이 있을 수 있습니다. 자세한 내용은이 AWS 백서에서 확인할 수 있습니다.

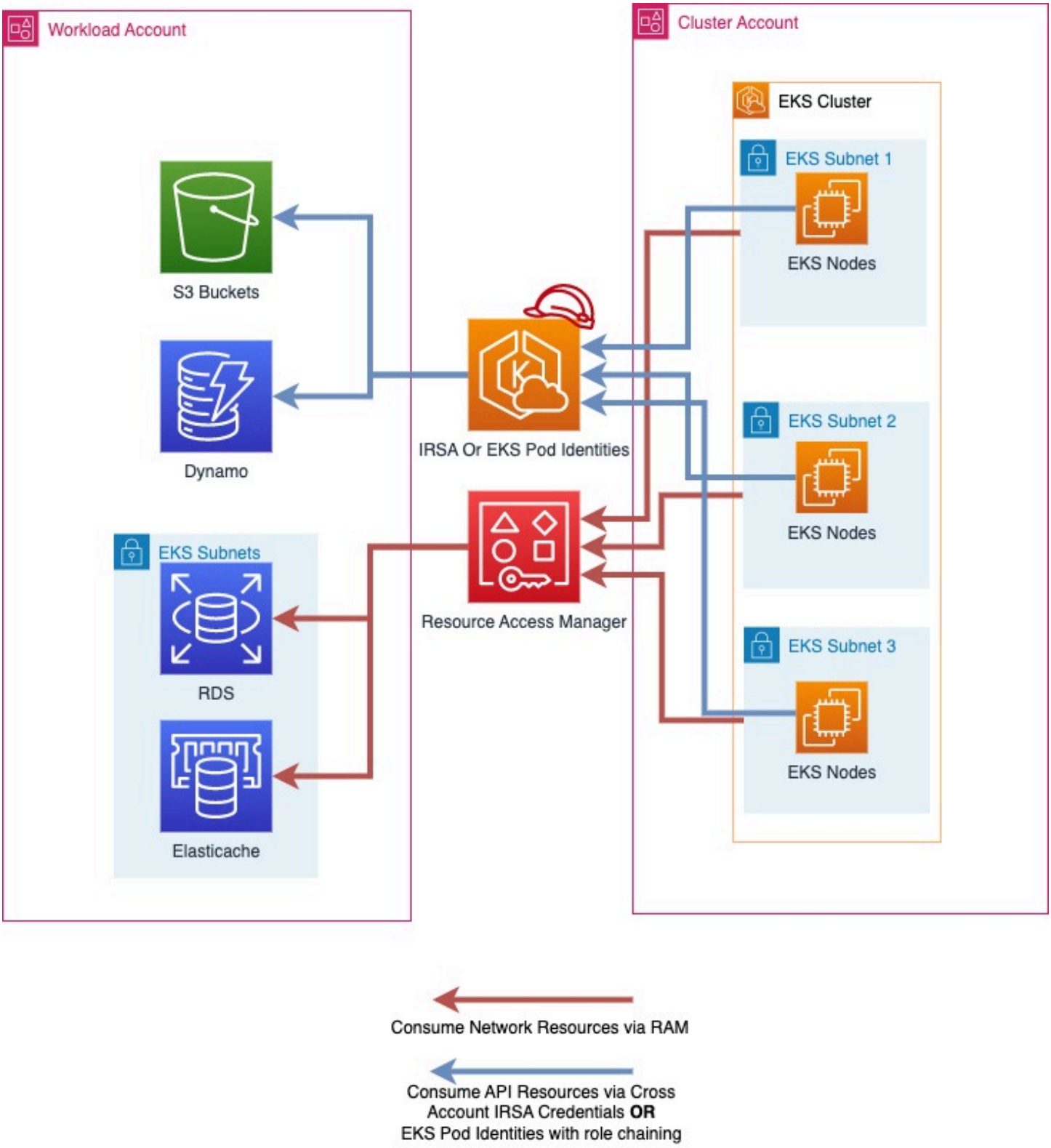
EKS 다중 계정 전략을 구현할 때 다음 접근 방식을 채택할 수 있습니다.

중앙 집중식 EKS 클러스터

이 접근 방식에서는 EKS 클러스터가 라는 단일 AWS 계정에 배포됩니다`Cluster Account`. 서비스 계정(IRSA) 또는 EKS Pod Identity에 대한 IAM 역할을 사용하여 임시 AWS 자격 증명을 제공하고 AWS Resource Access Manager(RAM)를 사용하여 네트워크 액세스를 간소화하면 다중 테넌트 EKS 클러스터에 대한 다중 계정 전략을 채택할 수 있습니다. https://docs.aws.amazon.com/eks/latest/userguide/pod-identities.html 클러스터 계정에는 VPC, 서브넷, EKS 클러스터, EC2/Fargate 컴퓨팅 리소스(작업자 노드) 및 EKS 클러스터를 실행하는 데 필요한 추가 네트워킹 구성이 포함됩니다.

다중 테넌트 클러스터에 대한 다중 워크로드 계정 전략에서 AWS 계정은 일반적으로 리소스 그룹을 격리하기 위한 메커니즘으로 kubernetes 네임스페이스와 일치합니다. 다중 테넌트 EKS 클러스터에 대한 다중 계정 전략을 구현할 때는 EKS 클러스터 내의 테넌트 격리 모범 사례를 계속 따라야 합니다.

`Cluster Accounts` AWS 조직에 여러가 있을 수 있으며 소프트웨어 개발 수명 주기 요구 사항에 `Cluster Accounts` 맞는 여러가 있는 것이 모범 사례입니다. 대규모로 작동하는 워크로드의 경우 모든 워크로드에 사용할 수 있는 충분한 kubernetes 및 AWS 서비스 할당량이 있는지 확인하기 `Cluster Accounts` 위해 여러 개가 필요할 수 있습니다.


Workload Account
Cluster Account
EKS Cluster
EKS Subnet 1
EKS Nodes
S3 Buckets
Dynamo
IRSA Or EKS Pod Identities
EKS Subnet 2
EKS Nodes
EKS Subnets
Resource Access Manager
RDS
EKS Subnet 3
EKS Nodes
Elasticache
Consume Network Resources via RAM
Consume API Resources via Cross Account IRSA Credentials OR EKS Pod Identities with role chaining

|위 다이어그램에서 AWS RAM은 클러스터 계정의 서브넷을 워크로드 계정으로 공유하는 데 사용됩니다. 그런 다음 EKS 포드에서 실행되는 워크로드는 IRSA 또는 EKS 포드 자격 증명 및 역할 체인을 사용하여 워크로드 계정에서 역할을 수임하고 AWS 리소스에 액세스합니다.

다중 테넌트 클러스터에 대한 다중 워크로드 계정 전략 구현

AWS Resource Access Manager와 서브넷 공유

AWS Resource Access Manager(RAM)를 사용하면 AWS 계정 간에 리소스를 공유할 수 있습니다.

AWS 조직에 RAM이 활성화된 경우 클러스터 계정의 VPC 서브넷을 워크로드 계정과 공유할 수 있습니다. 이렇게 하면 Amazon ElastiCache 클러스터 또는 Amazon Relational Database Service(RDS) 데이터베이스와 같은 워크로드 계정이 소유한 AWS 리소스를 EKS 클러스터와 동일한 VPC에 배포하고 EKS 클러스터에서 실행되는 워크로드에서 사용할 수 있습니다.

RAM을 통해 리소스를 공유하려면 클러스터 계정의 AWS 콘솔에서 RAM을 열고 "리소스 공유" 및 "리소스 공유 생성"을 선택합니다. 리소스 공유의 이름을 지정하고 공유하려는 서브넷을 선택합니다. 다음을 다시 선택하고 서브넷을 공유하려는 워크로드 계정의 12자리 계정 IDs를 입력하고 다음을 다시 선택한 다음 리소스 공유 생성을 클릭하여 완료합니다. 이 단계 후 워크로드 계정은 리소스를 해당 서브넷에 배포할 수 있습니다.

RAM 공유는 프로그래밍 방식으로 또는 코드형 인프라를 사용하여 생성할 수도 있습니다.

EKS Pod Identity와 IRSA 중에서 선택

re:Invent 2023에서 AWS는 임시 AWS 자격 증명을 EKS의 포드에 전달하는 더 간단한 방법으로 EKS Pod Identity를 출시했습니다. IRSA 및 EKS Pod Identity는 모두 임시 AWS 자격 증명을 EKS 포드에 전달하는 데 유효한 방법이며 계속 지원됩니다. 필요에 가장 적합한 제공 방법을 고려해야 합니다.

EKS 클러스터 및 여러 AWS 계정으로 작업할 때 IRSA는 EKS 클러스터가 직접 호스팅되는 계정 이외의 AWS 계정에서 역할을 직접 수임할 수 있지만, EKS Pod 자격 증명을 사용하려면 역할 체인을 구성해야 합니다. 심층 비교는 EKS 설명서를 참조하세요.

서비스 계정에 대한 IAM 역할을 사용하여 AWS API 리소스에 액세스

IAM 서비스 계정 역할(IRSA)을 사용하면 EKS에서 실행되는 워크로드에 임시 AWS 자격 증명을 제공할 수 있습니다. IRSA를 사용하여 클러스터 계정에서 워크로드 계정의 IAM 역할에 대한 임시 자격 증명을 가져올 수 있습니다. 이렇게 하면 클러스터 계정의 EKS 클러스터에서 실행되는 워크로드가 워크로드 계정에 호스팅된 S3 버킷과 같은 AWS API 리소스를 자동으로 사용하고 Amazon RDS 데이터베이스 또는 Amazon EFS FileSystems.

워크로드 계정에서 IAM 인증을 사용하는 AWS API 리소스 및 기타 리소스는 교차 계정 액세스가 가능하고 명시적으로 활성화된 경우를 제외하고 동일한 워크로드 계정의 IAM 역할에 대한 자격 증명으로만 액세스할 수 있습니다.

교차 계정 액세스를 위한 IRSA 활성화

클러스터 계정의 워크로드에 대해 IRSA가 워크로드 계정의 리소스에 액세스할 수 있도록 하려면 먼저 워크로드 계정에서 IAM OIDC 자격 증명 공급자를 생성해야 합니다. 이는 워크로드 계정에 자격 증명 공급자가 생성된다는 점을 제외하고 IRSA를 설정하는 동일한 절차로 수행할 수 있습니다.

그런 다음 EKS에서 워크로드에 대해 IRSA를 구성할 때 설명서와 동일한 단계를 따르지만 "다른 계정의 클러스터에서 자격 증명 공급자 생성 예제" 섹션에 언급된 대로 워크로드 계정의 12자리 계정 ID를 사용할 수 있습니다.

이를 구성한 후 EKS에서 실행되는 애플리케이션은 서비스 계정을 직접 사용하여 워크로드 계정에서 역할을 수임하고 해당 계정 내의 리소스를 사용할 수 있습니다.

EKS Pod Identity를 사용하여 AWS API 리소스에 액세스

EKS Pod Identity는 EKS에서 실행되는 워크로드에 AWS 자격 증명을 제공하는 새로운 방법입니다. EKS 포드 ID는 AWS 자격 증명을 EKS의 포드에 제공하기 위해 더 이상 OIDC 구성을 관리할 필요가 없으므로 AWS 리소스 구성을 간소화합니다.

교차 계정 액세스를 위한 EKS Pod Identity 활성화

IRSA와 달리 EKS Pod Identity는 EKS 클러스터와 동일한 계정의 역할에 대한 액세스 권한을 직접 부여하는 데만 사용할 수 있습니다. 다른 AWS 계정의 역할에 액세스하려면 EKS Pod Identity를 사용하는 포드가 역할 연결을 수행해야 합니다.

다양한 AWS SDKs에서 사용할 수 있는 프로세스 자격 증명 공급자를 사용하여 aws 구성 파일을 사용하여 애플리케이션 프로파일에서 역할 체인을 구성할 수 있습니다.는 다음과 같은 프로파일을 구성할 때 자격 증명 소스로 사용할 `credential_process` 수 있습니다.

```
# Content of the AWS Config file
[profile account_b_role]
source_profile = account_a_role
role_arn = arn:aws:iam::444455556666:role/account-b-role

[profile account_a_role]
credential_process = /eks-credential-processrole.sh
```

credentials_process에서 호출한 스크립트의 소스:

```
#!/bin/bash
# Content of the eks-credential-processrole.sh
# This will retreive the credential from the pod identities agent,
# and return it to the AWS SDK when referenced in a profile
curl -H "Authorization: $(cat $AWS_CONTAINER_AUTHORIZATION_TOKEN_FILE)"
 $AWS_CONTAINER_CREDENTIALS_FULL_URI | jq -c '{AccessKeyId: .AccessKeyId,
 SecretAccessKey: .SecretAccessKey, SessionToken: .Token, Expiration: .Expiration,
 Version: 1}'
```

위와 같이 계정 A 및 B 역할을 모두 사용하여 aws 구성 파일을 생성하고 포드 사양에서 AWS_CONFIG_FILE 및 AWS_PROFILE env vars를 지정할 수 있습니다. env vars가 포드 사양에 이미 있는 경우 EKS Pod Identity Webhook는 재정의하지 않습니다.

```
# Snippet of the PodSpec
containers:
  - name: container-name
    image: container-image:version
    env:
    - name: AWS_CONFIG_FILE
      value: path-to-customer-provided-aws-config-file
    - name: AWS_PROFILE
      value: account_b_role
```

EKS 포드 ID와의 역할 체인에 대한 역할 신뢰 정책을 구성할 때 EKS 특정 속성을 세션 태그로 참조하고 속성 기반 액세스 제어(ABAC)를 사용하여 IAM 역할에 대한 액세스를 포드가 속한 Kubernetes 서비스 계정과 같은 특정 EKS 포드 ID 세션으로만 제한할 수 있습니다.

이러한 속성 중 일부는 범용으로 고유하지 않을 수 있습니다. 예를 들어 EKS 클러스터 2개는 네임스페이스가 동일하고 클러스터 1개는 네임스페이스 간에 이름이 동일한 서비스 계정을 가질 수 있습니다. 따라서 EKS Pod Identity 및 ABAC를 통해 액세스 권한을 부여할 때는 서비스 계정에 액세스 권한을 부여할 때 항상 클러스터 ARN과 네임스페이스를 고려하는 것이 좋습니다.

교차 계정 액세스를 위한 ABAC 및 EKS Pod Identity

EKS Pod Identity를 사용하여 다중 계정 전략의 일부로 다른 계정의 역할(역할 체인)을 수임하는 경우 다른 계정에 액세스해야 하는 각 서비스 계정에 고유한 IAM 역할을 할당하거나 여러 서비스 계정에서 공통 IAM 역할을 사용하고 ABAC를 사용하여 액세스할 수 있는 계정을 제어할 수 있습니다.

ABAC를 사용하여 역할 체인이 있는 다른 계정에 역할을 수임할 수 있는 서비스 계정을 제어하려면 예상 값이 있을 때 역할 세션에서만 역할을 수임하도록 허용하는 역할 신뢰 정책 설명을 생성합니다.

다음 역할 신뢰 정책은 kubernetes-service-account, eks-cluster-arn 및 kubernetes-namespace 태그에 모두 예상 값이 있는 경우에만 EKS 클러스터 계정(계정 ID 111122223333)의 역할이 역할을 수임하도록 허용합니다.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:root"
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                    "aws:PrincipalTag/kubernetes-service-account":
 "PayrollApplication",
                    "aws:PrincipalTag/eks-cluster-arn": "arn:aws:eks:us-
east-1:111122223333:cluster/ProductionCluster",
                    "aws:PrincipalTag/kubernetes-namespace": "PayrollNamespace"
                }
            }
        }
    ]
}
```

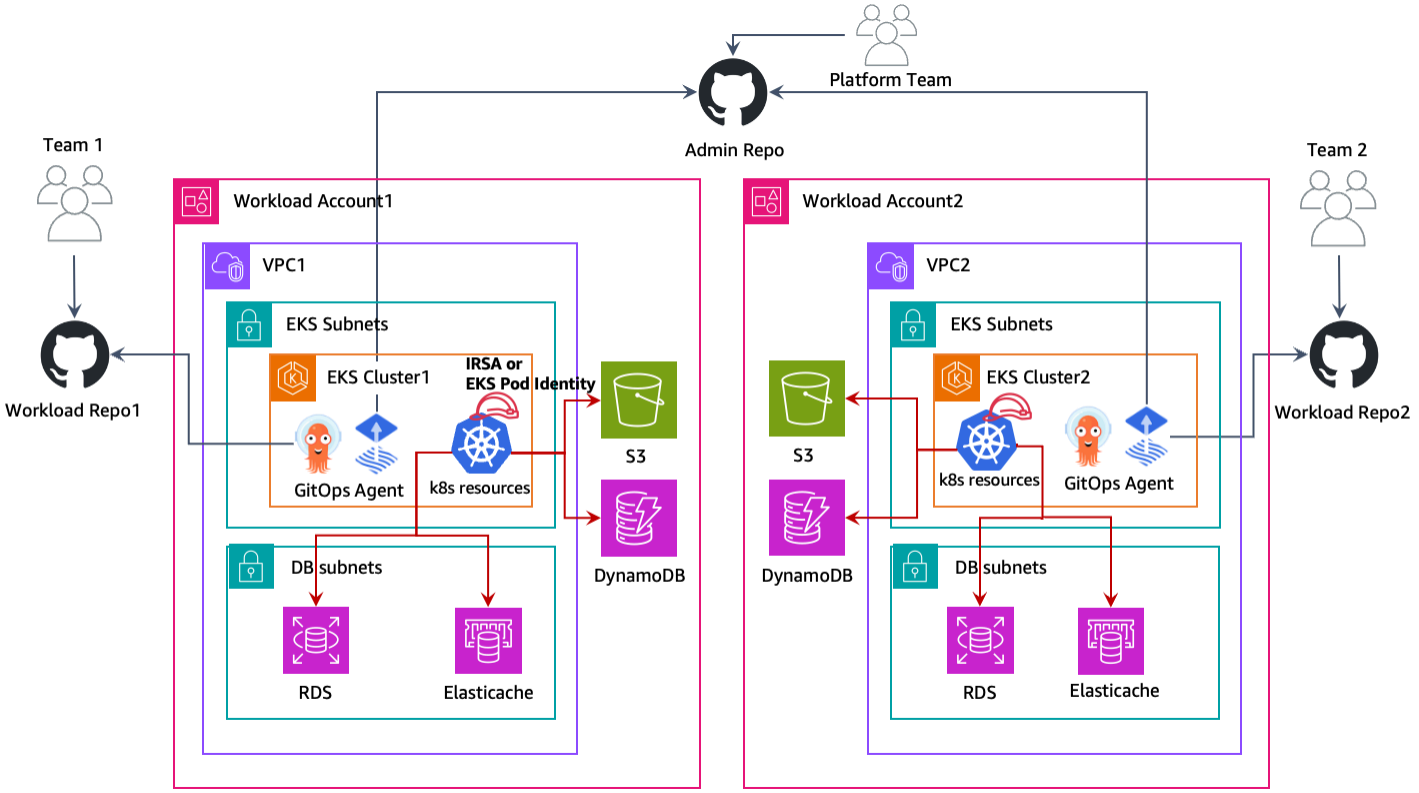
이 전략을 사용할 때는 공통 IAM 역할에 sts:AssumeRole 권한만 있고 다른 AWS 액세스 권한은 없는지 확인하는 것이 가장 좋습니다.

ABAC를 사용할 때는 IAM 역할 및 사용자를 엄격한 필요가 있는 사용자에게만 태그를 지정할 수 있는 사용자를 제어하는 것이 중요합니다. IAM 역할 또는 사용자에게 태그를 지정할 수 있는 사용자는 EKS Pod Identity에서 설정하는 것과 동일한 역할/사용자에 태그를 설정할 수 있으며 권한을 에스컬레이션할 수 있습니다. IAM 정책 또는 서비스 제어 정책(SCP)을 사용하여 IAM 역할 및 사용자에게 kubernetes- eks- 태그를 설정할 수 있는 액세스 권한을 가진 사용자를 제한할 수 있습니다.

분산형 EKS 클러스터

이 접근 방식에서 EKS 클러스터는 각 워크로드 AWS 계정에 배포되고 Amazon S3 버킷, VPCs, Amazon DynamoDB 테이블 등과 같은 다른 AWS 리소스와 함께 작동합니다. 각 워크로드 계정은 독립적이고 자체적이며 각 사업부/애플리케이션 팀에서 운영합니다. 이 모델을 사용하면 AI/ML 클러스터, 배치 처리, 범용 등 다양한 클러스터 기능에 대해 재사용 가능한 블루프린트를 생성하고 애플리케이션

팀 요구 사항에 따라 클러스터를 판매할 수 있습니다. 애플리케이션 팀과 플랫폼 팀 모두 해당 GitOps 리포지토리에서 운영하여 워크로드 클러스터에 대한 배포를 관리합니다.



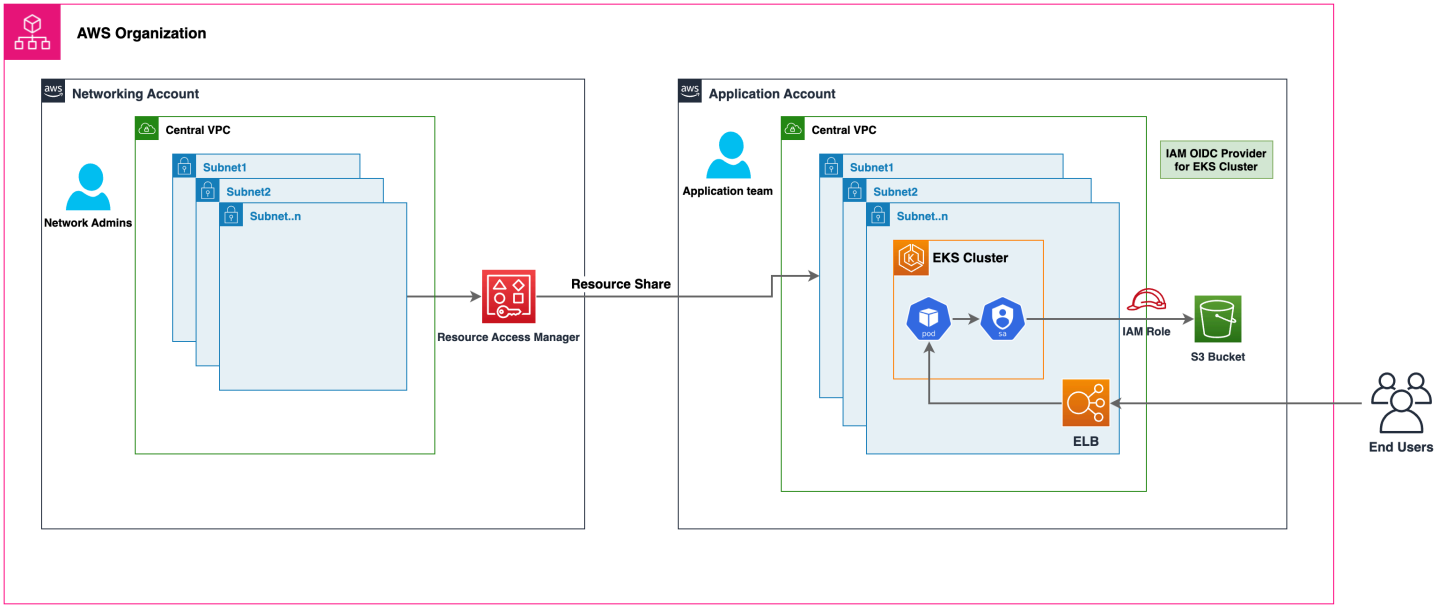
위 다이어그램에서 Amazon EKS 클러스터 및 기타 AWS 리소스는 각 워크로드 계정에 배포됩니다. 그런 다음 EKS 포드에서 실행되는 워크로드는 IRSA 또는 EKS 포드 ID를 사용하여 AWS 리소스에 액세스합니다.

GitOps는 전체 시스템이 Git 리포지토리에서 선언적으로 설명되도록 애플리케이션 및 인프라 배포를 관리하는 방법입니다. 버전 관리, 변경 불가능한 아티팩트 및 자동화의 모범 사례를 사용하여 여러 Kubernetes 클러스터의 상태를 관리할 수 있는 기능을 제공하는 운영 모델입니다. 이 다중 클러스터 모델에서 각 워크로드 클러스터는 여러 Git 리포지토리로 부트스트랩되어 각 팀(애플리케이션, 플랫폼, 보안 등)이 클러스터에 각 변경 사항을 배포할 수 있습니다.

각 계정의 서비스 계정(IRSA) 또는 EKS Pod Identity에 대한 IAM 역할을 활용하여 EKS 워크로드가 임시 aws 자격 증명을 가져와 다른 AWS 리소스에 안전하게 액세스할 수 있도록 합니다. https://docs.aws.amazon.com/eks/latest/userguide/pod-identities.html IAM 역할은 각 워크로드 AWS 계정에서 생성되어 k8s 서비스 계정에 매핑되어 임시 IAM 액세스를 제공합니다. 따라서이 접근 방식에서는 교차 계정 액세스가 필요하지 않습니다. IRSA의 각 워크로드에서를 설정하는 방법에 대한 서비스 계정용 IAM 역할 설명서와 각 계정에서 EKS 포드 ID를 설정하는 방법에 대한 EKS 포드 ID 설명서를 따릅니다.

중앙 집중식 네트워킹

또한 AWS RAM을 사용하여 VPC 서브넷을 워크로드 계정에 공유하고 Amazon EKS 클러스터 및 해당 계정의 기타 AWS 리소스를 시작할 수 있습니다. 이를 통해 중앙 집중식 네트워크 관리, 간소화된 네트워크 연결 및 분산형 EKS 클러스터를 사용할 수 있습니다. 이 접근 방식에 대한 자세한 안내 및 고려 사항은이 AWS 블로그를 참조하세요.



위 다이어그램에서 AWS RAM은 중앙 네트워킹 계정의 서브넷을 워크로드 계정으로 공유하는 데 사용됩니다. 그런 다음 EKS 클러스터 및 기타 AWS 리소스가 각 워크로드 계정의 해당 서브넷에서 시작됩니다. EKS 포드는 IRSA 또는 EKS 포드 ID를 사용하여 AWS 리소스에 액세스합니다.

중앙 집중식 및 비중앙 집중식 EKS 클러스터

중앙 집중식 또는 비중앙식으로를 실행하는 결정은 요구 사항에 따라 달라집니다. 이 표는 각 전략의 주요 차이점을 보여줍니다.

#	중앙 집중식 EKS 클러스터	분산형 EKS 클러스터
클러스터 관리:	단일 EKS 클러스터를 관리하는 것이 여러 클러스터를 관리하는 것보다 더 쉽습니다.	여러 EKS 클러스터 관리의 운영 오버헤드를 줄이려면 효율적인 클러스터 관리 자동화가 필요합니다.

#	중앙 집중식 EKS 클러스터	분산형 EKS 클러스터
비용 효율성:	비용 효율성을 높이는 EKS 클러스터 및 네트워크 리소스 재사용 허용	워크로드당 네트워킹 및 클러스터 설정이 필요하며 추가 리소스가 필요합니다.
복원력:	클러스터가 손상된 경우 중앙 집중식 클러스터의 여러 워크로드가 영향을 받을 수 있습니다.	클러스터가 손상된 경우 손상은 해당 클러스터에서 실행되는 워크로드로만 제한됩니다. 다른 모든 워크로드는 영향을 받지 않습니다.
격리 및 보안:	격리/소프트 다중 테넌시는와 같은 k8s 네이티브 구문을 사용하여 수행됩니다Namespaces . 워크로드는 CPU, 메모리 등과 같은 기본 리소스를 공유할 수 있습니다. AWS 리소스는 기본적으로 다른 AWS 계정에서 액세스할 수 없는 자체 워크로드 계정으로 격리됩니다.	워크로드가 리소스를 공유하지 않는 개별 클러스터 및 노드에서 실행됨에 따라 컴퓨팅 리소스에 대한 격리가 강화됩니다. AWS 리소스는 기본적으로 다른 AWS 계정에서 액세스할 수 없는 자체 워크로드 계정으로 격리됩니다.
성능 및 확장성:	워크로드가 매우 대규모로 증가함에 따라 클러스터 계정에서 kubernetes 및 AWS 서비스 할당량이 발생할 수 있습니다. 추가 클러스터 계정을 배포하여 더 확장할 수 있습니다.	더 많은 클러스터와 VPCs가 존재하면 각 워크로드에 더 많은 k8 및 AWS 서비스 할당량이 있습니다.
네트워킹:	클러스터당 단일 VPC가 사용되므로 해당 클러스터의 애플리케이션에 더 간단하게 연결할 수 있습니다.	분산형 EKS 클러스터 VPCs

#	중앙 집중식 EKS 클러스터	분산형 EKS 클러스터
Kubernetes 액세스 관리:	모든 워크로드 팀에 대한 액세스를 제공하고 kubernetes 리소스가 적절하게 분리되도록 하려면 클러스터에서 다양한 역할과 사용자를 유지해야 합니다.	각 클러스터가 워크로드/팀 전용이므로 간소화된 액세스 관리
AWS 액세스 관리:	AWS 리소스는 워크로드 계정의 IAM 역할로만 기본적으로 액세스할 수 있는 자체 계정에 배포됩니다. 워크로드 계정의 IAM 역할은 IRSA 또는 EKS Pod Identity를 사용하는 교차 계정으로 간주됩니다.	AWS 리소스는 워크로드 계정의 IAM 역할로만 기본적으로 액세스할 수 있는 자체 계정에 배포됩니다. 워크로드 계정의 IAM 역할은 IRSA 또는 EKS Pod Identity를 사용하여 포드에 직접 전달됩니다.

클러스터 액세스 관리

Amazon EKS 클러스터의 보안 및 무결성을 유지하려면 효과적인 액세스 관리가 중요합니다. 이 가이드에서는 AWS IAM Identity Center(이전 AWS SSO) 사용에 중점을 두고 EKS 액세스 관리를 위한 다양한 옵션을 살펴봅니다. 다양한 접근 방식을 비교하고, 장단점을 설명하고, 알려진 제한 사항과 고려 사항을 강조해 보겠습니다.

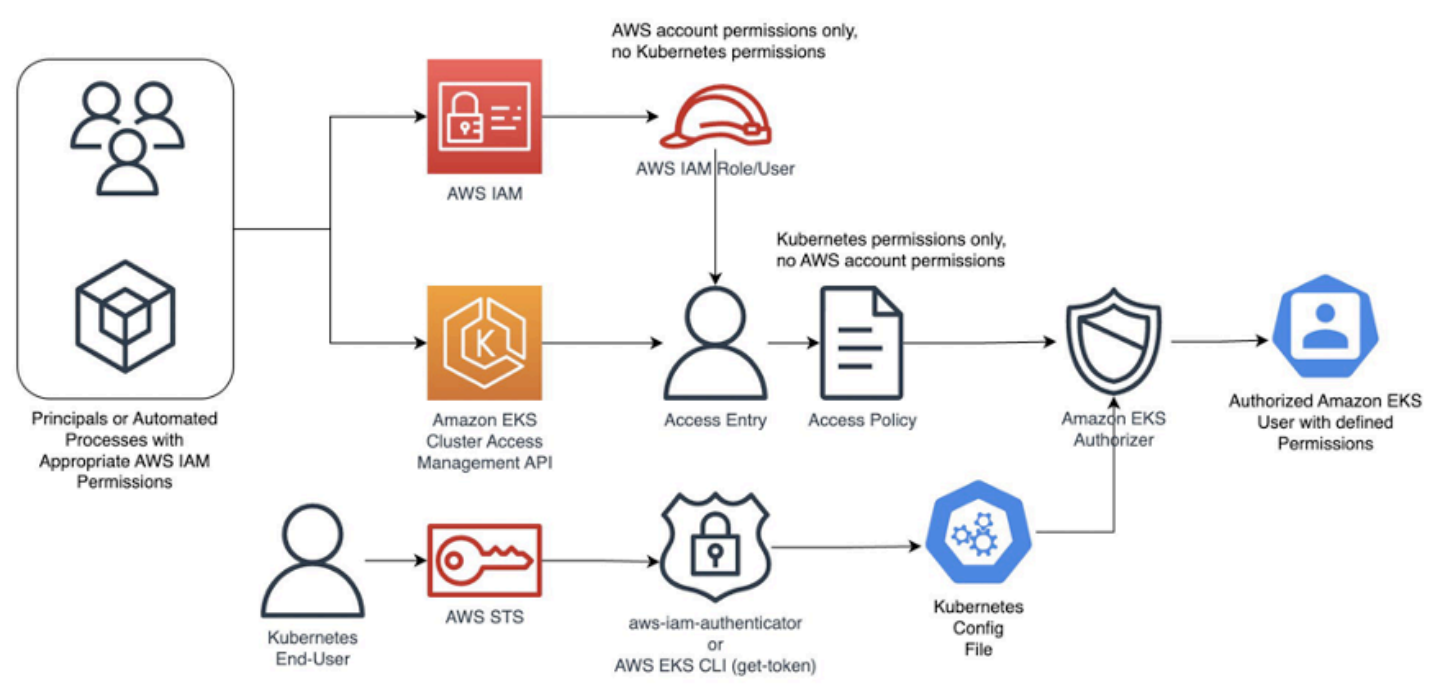
EKS 액세스 관리 옵션

> **Note**
>
> ConfigMap 기반 액세스 관리(aws-auth ConfigMap)는 더 이상 사용되지 않으며 Cluster Access Management(CAM) API로 대체됩니다. 새 EKS 클러스터의 경우 CAM API를 구현하여 클러스터 액세스를 관리합니다. aws-auth ConfigMap을 사용하는 기존 클러스터의 경우 CAM API를 사용하여 로 마이그레이션합니다.

옵션 1: 클러스터 액세스 관리(CAM) API를 사용하는 AWS IAM Identity Center

- 중앙 집중식 사용자 및 권한 관리
- 기존 자격 증명 공급자와의 통합(예: Microsoft AD,Okta, PingId 등)
- CAM API는 액세스 항목을 사용하여 AWS IAM 보안 주체(사용자 또는 역할)를 EKS 클러스터에 연결합니다. 이러한 항목은 IAM Identity Center의 관리형 ID와 함께 작동하므로 관리자가 Identity Center에 정의된 사용자 및 그룹에 대한 클러스터 액세스를 제어할 수 있습니다.

EKS 클러스터 인증 흐름:



1. 보안 주체(인간 사용자) 또는 자동화된 프로세스는 적절한 AWS 계정 권한을 제공하여 AWS IAM을 통해 인증합니다. 이 단계에서는 적절한 AWS IAM 보안 주체(역할 또는 사용자)에 매핑됩니다.
2. 다음으로 EKS 액세스 항목은 Kubernetes 권한만 포함하는 적절한 액세스 정책을 정의하여이 IAM 보안 주체를 Kubernetes RBAC 보안 주체(사용자 또는 그룹)에 매핑합니다.
3. Kubernetes 최종 사용자가 클러스터에 액세스하려고 하면 aws-iam-authenticator 또는 AWS EKS CLI에서 인증 요청을 처리하고 kubeconfig 파일의 클러스터 컨텍스트에 대해 검증합니다.
4. 마지막으로 EKS 권한 부여자는 인증된 사용자의 액세스 항목과 연결된 권한을 확인하고 그에 따라 액세스를 부여하거나 거부합니다.

- API는 Amazon EKS별 액세스 정책을 사용하여 각 액세스 항목에 대한 권한 부여 수준을 정의합니다. 이러한 정책을 IAM Identity Center에 설정된 역할 및 권한에 매핑하여 AWS 서비스 및 EKS 클러스터에서 일관된 액세스 제어를 보장할 수 있습니다.

ConfigMap 기반 액세스 관리의 이점:

1. 잘못된 구성의 위험 감소: 직접 API 기반 관리는 수동 ConfigMap 편집과 관련된 일반적인 오류를 제거합니다. 이렇게 하면 클러스터에서 사용자를 잠글 수 있는 실수로 인한 삭제 또는 구문 오류를 방지하는 데 도움이 됩니다.
2. 향상된 최소 권한 원칙: 클러스터 생성자 자격 증명에서 클러스터 관리자 권한이 필요하지 않으며 보다 세분화되고 적절한 권한 할당을 허용합니다. 휴지통 사용 사례에 대해이 권한을 추가하도록 선택할 수 있습니다.
3. 향상된 보안 모델: 액세스 항목이 적용되기 전에 액세스 항목에 대한 기본 제공 검증을 제공합니다. 또한는 인증을 위해 AWS IAM과의 통합을 강화합니다.
4. 간소화된 작업: AWS 네이티브 도구를 통해 권한을 관리하는 보다 직관적인 방법을 제공합니다.

모범 사례:

1. AWS Organizations를 사용하여 여러 계정을 관리하고 서비스 제어 정책(SCPs 적용합니다.
2. 다양한 EKS 역할(예: 관리자, 개발자, 읽기 전용)에 대한 특정 권한 세트를 생성하여 최소 권한 원칙을 구현합니다.
3. ABAC(속성 기반 액세스 제어)를 활용하여 사용자 속성을 기반으로 포드에 권한을 동적으로 할당합니다.
4. 액세스 권한을 정기적으로 감사하고 검토합니다.

고려 사항/제한 사항:

1. Identity Center에서 생성한 역할 ARNs에는 임의의 접미사가 있으므로 정적 구성에서 사용하기가 어렵습니다.
2. Kubernetes 리소스 수준에서 세분화된 권한에 대한 제한된 지원. 사용자 지정 Kubernetes RBAC 역할에는 추가 구성이 필요합니다. Kubernetes 네이티브 RBAC와 함께 EKS 클러스터의 고급 권한 관리에 Kyverno를 사용하는 것이 좋습니다.

옵션 2: Kubernetes 그룹에 매핑된 AWS IAM 사용자/역할

장점:

1. IAM 권한에 대한 세분화된 제어.
2. 예측 가능한 정적 역할 ARNs

단점:

1. 사용자 계정의 관리 오버헤드 증가
2. 중앙 집중식 자격 증명 관리 부족
3. IAM 개체의 확산 가능성

모범 사례:

1. 보안 및 관리 용이성을 높이기 위해 IAM 사용자 대신 IAM 역할 사용
2. 역할에 대한 이름 지정 규칙을 구현하여 일관성과 관리 용이성 보장
3. IAM 정책 조건을 활용하여 태그 또는 기타 속성을 기반으로 액세스를 제한합니다.
4. 액세스 키를 정기적으로 교체하고 권한을 검토합니다.

고려 사항/제한 사항:

1. 많은 수의 사용자 또는 역할 관리 시 확장성 문제
2. 기본 제공 Single Sign-On 기능 없음

옵션 3: OIDC 공급자

장점:

1. 기존 자격 증명 관리 시스템과의 통합
2. 사용자 계정의 관리 오버헤드 감소

단점:

1. 추가 구성 복잡성
2. 인증 중 지연 시간 증가 가능성
3. 외부 자격 증명 공급자에 대한 종속성

모범 사례:

1. 보안 토큰 검증을 보장하도록 OIDC 공급자를 신중하게 구성합니다.
2. 수명이 짧은 토큰을 사용하고 토큰 새로 고침 메커니즘을 구현합니다.
3. OIDC 구성을 정기적으로 감사하고 업데이트합니다.

외부 Single Sign-On 공급자를 Amazon EKS와 통합하는 참조 구현을 보려면이 가이드를 검토하세요.

고려 사항/제한 사항:

1. IAM에 비해 AWS 서비스와의 기본 통합이 제한적입니다.
2. EKS가 서명 키를 검색하려면 OIDC 공급자의 발급자 URL에 공개적으로 액세스할 수 있어야 합니다.

워크로드에 대한 AWS EKS Pod Identity와 IRSA 비교

Amazon EKS는 Amazon EKS 클러스터에서 실행되는 워크로드에 AWS IAM 권한을 부여하는 두 가지 방법, 즉 서비스 계정에 대한 IAM 역할(IRSA)과 EKS Pod Identity를 제공합니다.

IRSA와 EKS Pod Identity 모두 최소 권한 액세스, 자격 증명 격리 및 감사 가능성의 이점을 제공하지만 EKS Pod Identity는 워크로드에 권한을 부여하는 권장 방법입니다.

EKS 포드의 자격 증명 및 자격 증명에 대한 자세한 지침은 보안 모범 사례의 자격 증명 및 자격 증명 섹션을 참조하세요.

권장 사항

IAM Identity Center와 CAM API 결합

- 간소화된 관리: 클러스터 액세스 관리 API를 IAM Identity Center와 함께 사용하면 관리자가 다른 AWS 서비스와 함께 EKS 클러스터 액세스를 관리할 수 있으므로 다른 인터페이스 간에 전환하거나 ConfigMaps 수동으로 편집할 필요가 줄어듭니다.
- 액세스 항목을 사용하여 클러스터 외부에서 IAM 위탁자의 Kubernetes 권한을 관리합니다. EKS API, AWS 명령줄 인터페이스, AWS SDKs, AWS CloudFormation 및 AWS Management Console을 사용하여 클러스터에 대한 액세스를 추가하고 관리할 수 있습니다. 즉, 클러스터를 생성할 때 사용한 것과 동일한 도구로 사용자를 관리할 수 있습니다.
- 세분화된 Kubernetes 권한은 액세스 항목 및 액세스 정책을 통해 Kubernetes 사용자 또는 그룹을 SSO 자격 증명과 연결된 IAM 보안 주체와 매핑하여 적용할 수 있습니다.
- 시작하려면 인증 모드를 액세스 항목을 사용하도록 변경에 따라 기존 aws-auth ConfigMap 항목을 액세스 항목으로 마이그레이션합니다.

클러스터 Autoscaling 모범 사례

이 가이드에서는 Auto Mode, Karpenter 및 Kubernetes Cluster Autoscaler에 대한 지침을 포함하여 클러스터 Autoscaling에 대한 조언을 제공합니다.

주제

- EKS Auto Mode
- Karpenter
- Cluster Autoscaler

EKS Auto Mode

> **Tip**
>
> Amazon EKS Auto Mode를 직접 사용해 보고 싶으신가요? AWS Connected Community에 가입하여 AWS 전문가가 진행하는 예정된 가상 Kubernetes 워크숍에 참여하세요.

Amazon EKS Auto Mode는 안전하고 확장 가능한 클러스터 인프라를 AWS에서 관리하는 통합 Kubernetes 기능과 결합하여 Kubernetes 인프라 관리의 상당한 발전을 나타냅니다. 이 서비스는 완전 관리형 작업자 노드 작업을 제공하므로 고객이 관리형 노드 그룹 또는 AutoScaling 그룹를 설정할 필요가 없습니다.

주요 아키텍처 차이점은 EKS Auto Mode가 포드 요청에 대한 응답으로 EC2 인스턴스를 자동으로 프로비저닝하는 Karpenter 기반 시스템을 사용한다는 것입니다. 이러한 인스턴스는 EBS CSI 드라이버와 같은 추가 기능이 사전 설치된 Bottlerocket AMIs에서 실행되므로 AWS에서 인프라를 완벽하게 관리할 수 있습니다. 기존 조정 방법과 달리:

- 기존 Cluster Autoscaler(CAS)는 수동 노드 그룹 관리가 필요하며 노드 그룹당 단일 인스턴스 유형으로만 노드를 생성할 수 있습니다.
- 자체 관리형 Karpenter는 EC2 플릿 API로 작업하여 유연성을 높이고 다양한 인스턴스 유형을 프로비저닝할 수 있지만 고객 관리가 필요합니다.
- EKS Auto Mode는 관리형 NodePools 및 NodeClasses를 통해 모든 조정 작업을 자동으로 처리합니다.

새 시스템에는 다음과 같은 몇 가지 운영 개선 사항이 도입되었습니다.

- 수동 노드 그룹 구성이 없는 자동 포드 기반 조정
- 수신 리소스를 기반으로 ALB/NLB를 자동으로 생성하는 기본 제공 관리형 로드 밸런서 컨트롤러
- 사전 구성된 포드 자격 증명이 포함된 통합 보안 기능
- 자동 교체 시 최대 노드 런타임 21일

비용 관점에서 EKS Auto Mode는 표준 EC2 요금을 유지하는 동시에 Auto Mode 관리형 노드에 대해서만 관리 요금을 추가합니다. 고객은 여전히 Auto Mode 관리형 노드를 동일한 클러스터의 자체 관리형 노드와 혼합할 수 있습니다.

AWS는 대부분의 운영 측면을 처리하지만 고객은 클러스터 버전 관리에 대한 책임을 지며 작업자 노드의 롤링 업데이트를 트리거하는 제어 업그레이드를 수행할 수 있습니다.

Auto Mode를 사용하는 이유

Auto Mode는 Kubernetes 및 EKS의 이점을 원하지만 Auto Scaling, 로드 밸런싱 및 스토리지와 같은 중요한 플랫폼 부분의 업그레이드 및 설치/유지 관리와 같은 Kubernetes 관련 운영 부담을 최소화해야 하는 사용자를 대상으로 합니다. Auto Mode는 Kubernetes 유지 관리와 함께 발생하는 차별화되지 않은 과중한 작업을 최소화하는 데 EKS를 한 단계 더 나아갑니다.

FAQ

EKS Auto Mode와 오픈 소스 Karpenter의 차이점은 무엇인가요?

EKS Auto Mode는 프로덕션급 Kubernetes 실행을 간소화하는 대규모 기능 모음입니다. 이러한 기능 중 하나는 완전 관리형 Karpenter의 자동 크기 조정 이점입니다. 작업 관점에서 유일하게 차이점은 EKS Auto Mode에서는 Karpenter 포드의 배포, 조정 및 업그레이드를 직접 관리할 필요가 없다는 것입니다. 관리형 NodeClasses 및 NodePools와 같은 다른 모든 작업은 오픈 소스 Karpenter와 동일하게 작동합니다.

Auto Mode 관리형 노드와 함께 관리형 노드 그룹을 실행할 수 있나요?

예, Auto Mode와 함께 제공되는 Auto Scaling 노드와 함께 관리형 노드 그룹을 통해 정적 노드를 실행할 수 있습니다.

클러스터를 표준 EKS에서 EKS Auto Mode로 마이그레이션할 수 있나요?

예, 기존 클러스터에서 EKS Auto Mode를 활성화하는 지침은 공식 AWS 설명서에서 확인할 수 있습니다.

참고 사항: 1. Auto Mode를 활성화한 후 Karpenter 또는 AWS Load Balancer Controller 2와 같이 이제 Auto Mode에서 관리하는 설치한 구성 요소를 모두 제거해야 합니다. 설치된 추가 기능이 up-to-date 상태인지 확인해야 합니다. 설명서를 참조하세요.

EKS Auto Mode에서 NodePools를 구성하려면 어떻게 해야 합니까?

새 클러스터는 두 개의 NodePools 사전 구성되어 제공됩니다.

범용

```yaml
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  generation: 1
  labels:
    app.kubernetes.io/managed-by: eks
  name: general-purpose
spec:
  disruption:
    budgets:
      - nodes: 10%
    consolidateAfter: 30s
    consolidationPolicy: WhenEmptyOrUnderutilized
  template:
    spec:
      expireAfter: 336h
      nodeClassRef:
        group: eks.amazonaws.com
        kind: NodeClass
        name: default
      requirements:
        - key: karpenter.sh/capacity-type
          operator: In
          values:
            - on-demand
        - key: eks.amazonaws.com/instance-category
          operator: In
          values:
            - c
            - m
            - r
        - key: eks.amazonaws.com/instance-generation
          operator: Gt
          values:
            - '4'
        - key: kubernetes.io/arch
          operator: In
          values:
            - amd64
        - key: kubernetes.io/os
          operator: In
          values:
            - linux
      terminationGracePeriod: 24h0m0s
```

이 NodePool은 Karpenter에 다음 특성을 가진 노드를 시작하도록 지시합니다.

1. “온디맨드”의 용량 유형
2. 인스턴스 유형 C, M 또는 R
3. 인스턴스 생성 4
4. AMD 아키텍처
5. Linux OS

또한 지정된 시간에 모든 노드의 10%만 중단 상태에 있을 수 있으며 노드가 비어 있거나 사용률이 낮은 경우에만 통합이 발생한다고 선언하여 스케일 다운 로직이 무엇인지 정의합니다.

시스템

```yaml
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  generation: 1
  labels:
    app.kubernetes.io/managed-by: eks
  name: system
spec:
  disruption:
    budgets:
      - nodes: 10%
    consolidateAfter: 30s
    consolidationPolicy: WhenEmptyOrUnderutilized
  template:
    spec:
      expireAfter: 336h
      nodeClassRef:
        group: eks.amazonaws.com
        kind: NodeClass
        name: default
      requirements:
        - key: karpenter.sh/capacity-type
          operator: In
          values:
            - on-demand
        - key: eks.amazonaws.com/instance-category
          operator: In
          values:
            - c
            - m
            - r
        - key: eks.amazonaws.com/instance-generation
          operator: Gt
          values:
            - '4'
        - key: kubernetes.io/arch
          operator: In
          values:
            - amd64
            - arm64
        - key: kubernetes.io/os
          operator: In
          values:
            - linux
      taints:
        - effect: NoSchedule
```

이 NodePool은 다음과 같은 차이점을 제외하고 “일반 용도”와 유사합니다.

1. ARM 아키텍처와 AMD 아키텍처가 있는 노드를 허용합니다.
2. “CriticalAddonsOnly”에 대한 허용 오차가 없는 한 NoSchedule을 사용하여 이러한 노드를 테인팅합니다. 이는 EKS 추가 기능에서 내부적으로 사용하기 위한 것입니다.

사용자 지정

필요에 따라 사용자 지정 NodePools 생성할 수 있습니다. NodePools에 대한 자세한 내용은 Karpenter 설명서를 참조하세요.

새 노드가 시작될 때 Auto Mode에서 사용하는 AMI를 사용자 지정할 수 있습니까?

아니요, 현재 지원되는 유일한 AMIs는 Amazon 제공 Bottlerocket용입니다.

Kubernetes 호스트에 사용자 지정 도구 또는 에이전트를 설치하려면 어떻게 해야 하나요?

AMI 사용자 지정은 지원되지 않으므로 보안 스캔과 같은 작업을 위해 호스트 수준 소프트웨어가 필요한 경우 워크로드를 Kubernetes DaemonSet로 배포해야 합니다.

새 EKS Auto Mode 클러스터를 프로비저닝할 때 클러스터 데이터 영역에서 실행 중인 구성 요소는 무엇입니까?

eksctl 또는 AWS 콘솔을 사용하여 클러스터를 생성하는 경우 EKS Auto Mode 클러스터에서 실행되는 유일한 포드는 Kubernetes Metrics Server 포드입니다. Karpenter, AWS Load Balancer Controller 및 EBS CSI 드라이버와 같은 EKS Auto Mode의 다른 구성 요소는 모두 클러스터 외부에서 실행되고 관리됩니다.

새 EKS Auto Mode 클러스터를 지원하기 위해 실행 중인 관리형 구성 요소는 무엇입니까?

EKS Auto Mode는 프로덕션급 Kubernetes에 필요한 데이터 영역의 대부분의 배포를 완전히 자동화합니다. 여기에는 다음이 포함됩니다.

- Karpenter, 클러스터의 컴퓨팅 자동 크기 조정용
- 자동화된 Elastic Load Balancer 통합을 통해 Kubernetes 서비스를 쉽게 노출할 수 있는 AWS Load Balancer 컨트롤러

- EBS CSI
- VPC CNI
- EKS Pod Identity 에이전트

가 클러스터에서 포드로 실행하던 Auto Mode 구성 요소의 문제를 해결하려면 어떻게 해야 하나요?

EKS Auto Mode를 사용하면 AWS Load Balancer Controller 및 Karpenter와 같은 많은 구성 요소가 클러스터 외부에서 관리되므로 자체 관리 시 사용되는 로그를 동일하게 볼 수 없습니다. Auto Mode 기능의 기능 문제를 해결해야 하는 경우 AWS Support 티켓을 생성합니다.

Karpenter

Karpenter는 Kubernetes 클러스터 내에서 노드 수명 주기 관리를 개선하도록 설계된 오픈 소스 프로젝트입니다. 포드의 특정 일정 요구 사항에 따라 노드의 프로비저닝 및 프로비저닝 해제를 자동화하여 효율적인 규모 조정 및 비용 최적화를 지원합니다. 주요 기능은 다음과 같습니다.

- 리소스 제약으로 인해 Kubernetes 스케줄러가 예약할 수 없는 포드를 모니터링합니다.
- 예약 불가능한 포드의 예약 요구 사항(리소스 요청, 노드 선택기, 친화도, 허용치 등)을 평가합니다.
- 해당 포드의 요구 사항을 충족하는 새 노드를 프로비저닝합니다.
- 더 이상 필요하지 않은 노드를 제거합니다.

Karpenter를 사용하면 테인트, 레이블, 요구 사항(인스턴스 유형, 영역 등) 및 총 프로비저닝된 리소스에 대한 제한과 같은 노드 프로비저닝에 대한 제약 조건으로 NodePools를 정의할 수 있습니다. 워크로드를 배포할 때 포드 사양에서 리소스 요청/제한, 노드 선택기, 노드/포드 친화도, 허용 오차 및 토폴로지 분산 제약 조건과 같은 다양한 예약 제약 조건을 지정할 수 있습니다. 그러면 Karpenter는 이러한 사양에 따라 적절한 크기의 노드를 프로비저닝합니다.

Karpenter를 사용하는 이유

Karpenter를 시작하기 전에 Kubernetes 사용자는 주로 Amazon EC2 Auto Scaling 그룹과 Kubernetes Cluster Autoscaler(CAS)를 사용하여 클러스터의 컴퓨팅 용량을 동적으로 조정했습니다. Karpenter를 사용하면 Karpenter로 얻을 수 있는 유연성과 다양성을 달성하기 위해 수십 개의 노드 그룹을 생성할 필요가 없습니다. CAS와 달리 Karpenter는 Kubernetes 버전과 긴밀하게 결합되지 않으며 AWS와 Kubernetes APIs 간에 점프할 필요가 없습니다.

Karpenter는 인스턴스 오케스트레이션 책임을 단일 시스템 내에 통합하여 더 간단하고 안정적이며 클러스터를 인식합니다. Karpenter는 다음과 같은 간소화된 방법을 제공하여 Cluster Autoscaler에서 발생하는 몇 가지 문제를 극복하도록 설계되었습니다.

- 워크로드 요구 사항에 따라 노드를 프로비저닝합니다.
- 유연한 NodePool 옵션을 사용하여 인스턴스 유형별로 다양한 노드 구성을 생성합니다. Karpenter는 많은 특정 사용자 지정 노드 그룹을 관리하는 대신 유연한 단일 NodePool을 사용하여 다양한 워크로드 용량을 관리할 수 있습니다.
- 노드를 빠르게 시작하고 포드를 예약하여 대규모로 포드 예약을 개선합니다.

Karpenter 사용에 대한 자세한 내용과 설명서는 karpenter.sh 사이트를 참조하십시오.

권장 사항

모범 사례는 Karpenter 자체, NodePools 및 포드 예약에 대한 섹션으로 나뉩니다.

Karpenter 모범 사례

다음 모범 사례에서는 Karpenter 자체와 관련된 주제를 다룹니다.

프로덕션 클러스터에서 AMIs 잠금

Karpenter가 프로덕션 클러스터에 사용하는 잘 알려진 Amazon Machine Image(AMIs)를 고정시키는 것이 좋습니다. 별칭이 로 설정된 `amiSelector` 상태에서를 사용하거나 `@latest`테스트되지 않은 AMIs가 릴리스될 때 배포되는 다른 방법을 사용하면 프로덕션 클러스터에서 워크로드 장애 및 가동 중지가 발생할 위험이 있습니다. 따라서 비프로덕션 클러스터에서 최신 버전을 테스트하는 동안 프로덕션 클러스터에 대해 테스트된 작동 버전의 AMIs를 고정시키는 것이 좋습니다. 예를 들어 다음과 같이 NodeClass에서 별칭을 설정할 수 있습니다.

```
amiSelectorTerms
  - alias: al2023@v20240807
```

Karpenter에서 AMIs 관리하고 고정하는 방법에 대한 자세한 내용은 Karpenter 설명서의 AMIs 관리를 참조하세요.

용량 요구 사항이 변화하는 워크로드에 Karpenter 사용

Karpenter는 Autoscaling Groups(ASGs) 및 Managed Node Groups(MNG)보다 Kubernetes 네이티브 APIs에 더 가깝게 조정 관리를 제공합니다.MNGs ASGs 및 MNGs는 EC2 CPU 로드와 같은 AWS 수준

지표를 기반으로 조정이 트리거되는 AWS 네이티브 추상화입니다. Cluster Autoscaler는 Kubernetes 추상화를 AWS 추상화에 연결하지만 특정 가용 영역에 대한 예약과 같은 유연성 때문에 약간의 유연성을 잃습니다.

Karpenter는 AWS 추상화 계층을 제거하여 일부 유연성을 Kubernetes에 직접 제공합니다. Karpenter는 수요가 높고 급증하는 기간이 발생하거나 컴퓨팅 요구 사항이 다양한 워크로드가 있는 클러스터에 가장 적합합니다. MNGs 및 ASGs는 보다 정적이고 일관된 경향이 있는 워크로드를 실행하는 클러스터에 적합합니다. 요구 사항에 따라 동적 및 정적 관리형 노드를 혼합하여 사용할 수 있습니다.

다음과 같은 경우 다른 Auto Scaling 프로젝트를 고려하세요.

Karpenter에서 아직 개발 중인 기능이 필요합니다. Karpenter는 비교적 새로운 프로젝트이므로 Karpenter의 일부가 아닌 기능이 필요한 경우 당분간 다른 Auto Scaling 프로젝트를 고려하세요.

EKS Fargate 또는 노드 그룹에 속하는 작업자 노드에서 Karpenter 컨트롤러 실행

Karpenter는 차트 Helm을 사용하여 설치됩니다. Helm 차트는 Karpenter 컨트롤러와 Webhook 포드를 배포로 설치합니다. 배포는 컨트롤러를 클러스터 조정에 사용하기 전에 실행해야 합니다. 작업자 노드가 하나 이상 있는 작은 노드 그룹을 하나 이상 사용하는 것이 좋습니다. 또는 `karpenter` 네임스페이스에 대한 Fargate 프로파일을 생성하여 EKS Fargate에서 이러한 포드를 실행할 수 있습니다. 이렇게 하면이 네임스페이스에 배포된 모든 포드가 EKS Fargate에서 실행됩니다. Karpenter에서 관리하는 노드에서는 Karpenter를 실행하지 마십시오.

Karpenter에서는 사용자 지정 시작 템플릿이 지원되지 않습니다.

v1 APIs. 사용자 지정 사용자 데이터를 사용하거나 EC2NodeClass에서 사용자 지정 AMIs를 직접 지정할 수 있습니다. 이 작업을 수행하는 방법에 대한 자세한 내용은 NodeClasses에서 확인할 수 있습니다.

워크로드에 맞지 않는 인스턴스 유형 제외

클러스터에서 실행되는 워크로드에 필요하지 않은 경우 `node.kubernetes.io/instance-type` 키가 있는 특정 인스턴스 유형을 제외하는 것이 좋습니다.

다음 예제에서는 대용량 Graviton 인스턴스 프로비저닝을 방지하는 방법을 보여줍니다.

```
- key: node.kubernetes.io/instance-type
  operator: NotIn
  values:
  - m6g.16xlarge
```

```
  - m6gd.16xlarge
  - r6g.16xlarge
  - r6gd.16xlarge
  - c6g.16xlarge
```

스팟 사용 시 중단 처리 활성화

Karpenter는 기본 중단 처리를 지원하며 스팟 인스턴스 중단, 예약된 유지 관리 이벤트, 워크로드를 중단시킬 수 있는 인스턴스 종료/중지 이벤트와 같은 비자발적 중단 이벤트를 처리할 수 있습니다. Karpenter는 노드에 대해 이러한 이벤트를 감지하면 영향을 받는 노드를 자동으로 테인트, 드레이닝 및 종료하여 중단 전에 워크로드를 정상적으로 정리합니다. 2분 알림이 포함된 스팟 중단의 경우 Karpenter는 인스턴스를 회수하기 전에 포드를 이동할 수 있도록 새 노드를 빠르게 시작합니다. 중단 처리를 활성화하려면이 목적으로 프로비저닝된 SQS 대기열의 이름으로 `--interruption-queue` CLI 인수를 구성합니다. 여기에 설명된 대로 노드 종료 핸들러와 함께 Karpenter 중단 처리를 사용하지 않는 것이 좋습니다.

체크포인트 또는 기타 형태의 정상적인 드레이닝이 필요하고 종료 전에 2분이 필요한 포드는 클러스터에서 Karpenter 중단 처리를 활성화해야 합니다.

아웃바운드 인터넷 액세스가 없는 Amazon EKS 프라이빗 클러스터

인터넷 경로 없이 EKS 클러스터를 VPC에 프로비저닝할 때는 EKS 설명서에 표시된 프라이빗 클러스터 요구 사항에 따라 환경을 구성해야 합니다. 또한 VPC에서 STS VPC 리전 엔드포인트를 생성했는지 확인해야 합니다. 그렇지 않으면 아래에 표시된 것과 유사한 오류가 표시됩니다.

```
{"level":"FATAL","time":"2024-02-29T14:28:34.392Z","logger":"controller","message":"Checking
 EC2 API connectivity, WebIdentityErr: failed to retrieve credentials
\ncaused by: RequestError: send request failed\ncaused by: Post
 \"https://sts.<region>.amazonaws.com/\": dial tcp 54.239.32.126:443: i/o
 timeout","commit":"596ea97"}
```

Karpenter Controller는 서비스 계정에 대한 IAM 역할(IRSA)을 사용하기 때문에 프라이빗 클러스터에서 이러한 변경이 필요합니다. IRSA로 구성된 포드는 AWS Security Token Service(AWS STS) API를 호출하여 자격 증명을 획득합니다. 아웃바운드 인터넷 액세스가 없는 경우 VPC에서 AWS STS VPC 엔드포인트를 생성하고 사용해야 합니다.

프라이빗 클러스터에서는 SSM에 대한 VPC 엔드포인트를 생성해야 합니다. Karpenter가 새 노드를 프로비저닝하려고 하면 시작 템플릿 구성과 SSM 파라미터를 쿼리합니다. VPC에 SSM VPC 엔드포인트가 없는 경우 다음 오류가 발생합니다.

```
{"level":"ERROR","time":"2024-02-29T14:28:12.889Z","logger":"controller","message":"Unable
 to hydrate the AWS launch template cache, RequestCanceled: request context canceled
\ncaused by: context canceled","commit":"596ea97","tag-key":"karpenter.k8s.aws/
cluster","tag-value":"eks-workshop"}
...
{"level":"ERROR","time":"2024-02-29T15:08:58.869Z","logger":"controller.nodeclass","message":"d
 amis from ssm, getting ssm parameter \"/aws/service/eks/optimized-ami/1.27/amazon-
linux-2/recommended/image_id\", RequestError: send request failed\ncaused by:
 Post \"https://ssm.<region>.amazonaws.com/\": dial tcp 67.220.228.252:443: i/o
 timeout","commit":"596ea97","ec2nodeclass":"default","query":"/aws/service/eks/
optimized-ami/1.27/amazon-linux-2/recommended/image_id"}
```

Price List Query API 에는 VPC 엔드포인트가 없습니다. 따라서 시간 경과에 따라 요금 데이터가 오래됩니다. Karpenter는 바이너리에 온디맨드 요금 데이터를 포함시켜이 문제를 해결하지만 Karpenter가 업그레이드될 때만 해당 데이터를 업데이트합니다. 요금 데이터에 대한 요청이 실패하면 다음과 같은 오류 메시지가 표시됩니다.

```
{"level":"ERROR","time":"2024-02-29T15:08:58.522Z","logger":"controller.pricing","message":"ret
 on-demand pricing data, RequestError: send request failed\ncaused by: Post
 \"https://api.pricing.<region>.amazonaws.com/\": dial tcp 18.196.224.8:443:
 i/o timeout; RequestError: send request failed\ncaused by: Post \"https://
api.pricing.<region>.amazonaws.com/\": dial tcp 18.185.143.117:443: i/o
 timeout","commit":"596ea97"}
```

완전 프라이빗 EKS 클러스터에서 Karpenter를 사용하고 생성할 VPC 엔드포인트를 알아보려면이 설명서를 참조하세요.

NodePools 생성

다음 모범 사례에서는 NodePools 생성과 관련된 주제를 다룹니다.

다음과 같은 경우 여러 NodePools 생성합니다.

다른 팀이 클러스터를 공유하고 다른 작업자 노드에서 워크로드를 실행해야 하거나 OS 또는 인스턴스 유형 요구 사항이 다른 경우 여러 NodePools 생성합니다. 예를 들어 한 팀은 Bottlerocket을 사용하고 다른 팀은 Amazon Linux를 사용할 수 있습니다. 마찬가지로 한 팀은 다른 팀에 필요하지 않은 비용이 많이 드는 GPU 하드웨어에 액세스할 수 있습니다. 여러 NodePools를 사용하면 각 팀이 가장 적합한 자산을 사용할 수 있습니다.

상호 배타적이거나 가중치가 적용되는 NodePools 생성

일관된 예약 동작을 제공하기 위해 상호 배타적이거나 가중치가 적용되는 NodePools를 생성하는 것이 좋습니다. 일치하지 않고 여러 NodePools 일치하는 경우 Karpenter는 사용할를 무작위로 선택하여 예기치 않은 결과를 초래합니다. 여러 NodePools 생성하는 데 유용한 예는 다음과 같습니다.

GPU를 사용하여 NodePool을 생성하고 이러한 (저가) 노드에서 특수 워크로드만 실행하도록 허용:

```
# NodePool for GPU Instances with Taints
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: gpu
spec:
  disruption:
    consolidateAfter: 1m
    consolidationPolicy: WhenEmptyOrUnderutilized
  template:
    metadata: {}
    spec:
      nodeClassRef:
        group: karpenter.k8s.aws
        kind: EC2NodeClass
        name: default
      expireAfter: Never
      requirements:
      - key: node.kubernetes.io/instance-type
        operator: In
        values:
        - p3.8xlarge
        - p3.16xlarge
      - key: kubernetes.io/os
        operator: In
        values:
        - linux
      - key: kubernetes.io/arch
        operator: In
        values:
        - amd64
      - key: karpenter.sh/capacity-type
        operator: In
        values:
        - on-demand
```

```
      taints:
      - effect: NoSchedule
        key: nvidia.com/gpu
        value: "true"
```

테인트에 대한 허용 오차가 있는 배포:

```
# Deployment of GPU Workload will have tolerations defined
apiVersion: apps/v1
kind: Deployment
metadata:
  name: inflate-gpu
spec:
    spec:
      tolerations:
      - key: "nvidia.com/gpu"
        operator: "Exists"
        effect: "NoSchedule"
```

다른 팀을 위한 일반 배포의 경우 NodePool 사양에 nodeAffinity가 포함될 수 있습니다. 그러면 배포에서 nodeSelectorTerms를 사용하여를 일치시킬 수 있습니다`billing-team`.

```
# NodePool for regular EC2 instances
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: generalcompute
spec:
  template:
    metadata:
      labels:
        billing-team: my-team
    spec:
      nodeClassRef:
        group: karpenter.k8s.aws
        kind: EC2NodeClass
        name: default
      expireAfter: Never
      requirements:
      - key: node.kubernetes.io/instance-type
        operator: In
        values:
        - m5.large
```

```
        - m5.xlarge
        - m5.2xlarge
        - c5.large
        - c5.xlarge
        - c5a.large
        - c5a.xlarge
        - r5.large
        - r5.xlarge
      - key: kubernetes.io/os
        operator: In
        values:
        - linux
      - key: kubernetes.io/arch
        operator: In
        values:
        - amd64
      - key: karpenter.sh/capacity-type
        operator: In
        values:
        - on-demand
```

nodeAffinity를 사용한 배포:

```
# Deployment will have spec.affinity.nodeAffinity defined
kind: Deployment
metadata:
  name: workload-my-team
spec:
  replicas: 200
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                - key: "billing-team"
                  operator: "In"
                  values: ["my-team"]
```

타이머(TTL)를 사용하여 클러스터에서 노드 자동 삭제

프로비저닝된 노드의 타이머를 사용하여 워크로드 포드가 없거나 만료 시간에 도달한 노드를 삭제할 시기를 설정할 수 있습니다. 노드 만료를 업그레이드 수단으로 사용할 수 있으므로 노드가 사용 중지

되고 업데이트된 버전으로 대체됩니다. 를 사용하여 노드 만료를 구성하는 방법에 대한 자세한 내용은 Karpenter 설명서의 만료를 참조`spec.template.spec`하세요.

특히 스팟을 활용할 때 Karpenter가 프로비저닝할 수 있는 인스턴스 유형을 지나치게 제한하지 마세요.

스팟을 사용할 때 Karpenter는 Price Capacity Optimized allocation 전략을 사용하여 EC2 인스턴스를 프로비저닝합니다. 이 전략은 시작하는 인스턴스 수에 대해 가장 깊은 풀에서 인스턴스를 프로비저닝하고 중단 위험이 가장 낮도록 EC2에 지시합니다. 그런 다음 EC2 플릿은 이러한 풀의 최저 가격에서 스팟 인스턴스를 요청합니다. Karpenter가 사용할 수 있는 인스턴스 유형이 많을수록 EC2가 스팟 인스턴스의 런타임을 최적화할 수 있습니다. 기본적으로 Karpenter는 클러스터가 배포된 리전 및 가용 영역에서 EC2가 제공하는 모든 인스턴스 유형을 사용합니다. Karpenter는 보류 중인 포드를 기반으로 모든 인스턴스 유형 세트 중에서 지능적으로 선택하여 적절한 크기의 탑재된 인스턴스에 포드가 예약되도록 합니다. 예를 들어 포드에 GPU가 필요하지 않은 경우 Karpenter는 포드를 GPU를 지원하는 EC2 인스턴스 유형으로 예약하지 않습니다. 사용할 인스턴스 유형이 확실하지 않은 경우 Amazon ec2-instance-selector를 실행하여 컴퓨팅 요구 사항에 맞는 인스턴스 유형 목록을 생성할 수 있습니다. 예를 들어 CLI는 메모리 vCPU, 아키텍처 및 리전을 입력 파라미터로 받아 해당 제약 조건을 충족하는 EC2 인스턴스 목록을 제공합니다.

```
$ ec2-instance-selector --memory 4 --vcpus 2 --cpu-architecture x86_64 -r ap-
southeast-1
c5.large
c5a.large
c5ad.large
c5d.large
c6i.large
t2.medium
t3.medium
t3a.medium
```

스팟 인스턴스를 사용할 때 Karpenter에 너무 많은 제약 조건을 두면 애플리케이션의 가용성에 영향을 미칠 수 있으므로 그렇게 하면 안 됩니다. 예를 들어 특정 유형의 모든 인스턴스가 회수되고 이를 대체할 수 있는 적절한 대안이 없다고 가정해 보겠습니다. 구성된 인스턴스 유형의 스팟 용량이 보충될 때까지 포드는 보류 상태로 유지됩니다. 스팟 풀은 AZs. 즉, 스팟을 사용할 때 Karpenter가 다양한 인스턴스 유형을 사용하도록 허용하는 것이 일반적인 모범 사례입니다.

포드 예약

다음 모범 사례는 노드 프로비저닝을 위해 Karpenter를 사용하여 클러스터에 포드를 배포하는 것과 관련이 있습니다.

고가용성을 위한 EKS 모범 사례 준수

고가용성 애플리케이션을 실행해야 하는 경우 일반적인 EKS 모범 사례 권장 사항을 따르세요. 노드와 영역에 포드를 분산하는 방법에 대한 자세한 내용은 Karpenter 설명서의 토폴로지 분산을 참조하세요. 중단 예산을 사용하여 포드를 제거하거나 삭제하려는 시도가 있는 경우 유지 관리해야 하는 사용 가능한 최소 포드를 설정합니다.

계층화된 제약 조건을 사용하여 클라우드 공급자가 제공하는 컴퓨팅 기능 제한

Karpenter의 계층화된 제약 조건 모델을 사용하면 복잡한 NodePool 및 포드 배포 제약 조건 세트를 생성하여 포드 일정에 가장 적합한 일치 항목을 얻을 수 있습니다. 포드 사양이 요청할 수 있는 제약 조건의 예는 다음과 같습니다.

- 특정 애플리케이션만 사용할 수 있는 가용 영역에서 실행해야 합니다. 예를 들어 특정 가용 영역에 있는 EC2 인스턴스에서 실행되는 다른 애플리케이션과 통신해야 하는 포드가 있다고 가정해 보겠습니다. VPC에서 교차 AZ 트래픽을 줄이는 것이 목표인 경우 EC2 인스턴스가 위치한 AZ에서 포드를 공동 배치할 수 있습니다. 이러한 유형의 대상 지정은 노드 선택기를 사용하여 수행되는 경우가 많습니다. 노드 선택기에 대한 자세한 내용은 Kubernetes 설명서를 참조하세요.
- 특정 종류의 프로세서 또는 기타 하드웨어가 필요합니다. GPU에서 포드를 실행해야 하는 포드 사양 예제는 Karpenter 문서의 액셀러레이터 섹션을 참조하세요.

결제 경보를 생성하여 컴퓨팅 지출 모니터링

클러스터를 자동으로 확장하도록 구성할 때 지출이 임계값을 초과하면 경고하는 결제 경보를 생성하고 Karpenter 구성에 리소스 제한을 추가해야 합니다. Karpenter로 리소스 제한을 설정하는 것은 Karpenter NodePool에서 인스턴스화할 수 있는 컴퓨팅 리소스의 최대 양을 나타낸다는 점에서 AWS Autoscaling 그룹의 최대 용량을 설정하는 것과 유사합니다.

> **Note**
>
> 전체 클러스터에 대한 전역 제한을 설정할 수 없습니다. 제한은 특정 NodePools에 적용됩니다.

아래 코드 조각은 Karpenter에게 최대 1,000개의 CPU 코어와 1000Gi의 메모리만 프로비저닝하도록 지시합니다. Karpenter는 한도가 충족되거나 초과된 경우에만 용량 추가를 중지합니다. 한도를 초과하면 Karpenter 컨트롤러가 컨트롤러의 로그에 `memory resource usage of 1001 exceeds limit of 1000` 또는 유사한 메시지를 씁니다. 컨테이너 로그를 CloudWatch 로그로 라우팅하는 경우 지표 필터를 생성하여 로그에서 특정 패턴 또는 용어를 찾은 다음 CloudWatch 경보를 생성하여 구성된 지표 임계값이 위반되면 알림을 받을 수 있습니다.

Karpenter에서 제한을 사용하는 자세한 내용은 Karpenter 설명서의 리소스 제한 설정을 참조하세요.

```
spec:
  limits:
    cpu: 1000
    memory: 1000Gi
```

제한을 사용하지 않거나 Karpenter가 프로비저닝할 수 있는 인스턴스 유형을 제한하지 않으면 Karpenter는 필요에 따라 클러스터에 컴퓨팅 용량을 계속 추가합니다. 이러한 방식으로 Karpenter를 구성하면 클러스터를 자유롭게 확장할 수 있지만 비용에 상당한 영향을 미칠 수도 있습니다. 따라서 결제 경보를 구성하는 것이 좋습니다. 결제 경보를 사용하면 계정(들)에서 계산된 예상 요금이 정의된 임계값을 초과할 때 알림을 받고 사전 알림을 받을 수 있습니다. 자세한 내용은 예상 요금을 선제적으로 모니터링하기 위한 Amazon CloudWatch 결제 경보 설정을 참조하세요.

기계 학습을 사용하여 비용 및 사용량을 지속적으로 모니터링하여 비정상적인 지출을 감지하는 AWS 비용 관리 기능인 비용 이상 탐지를 활성화할 수도 있습니다. 자세한 내용은 AWS 비용 이상 탐지 시작 안내서에서 확인할 수 있습니다. 지금까지 AWS Budgets에서 예산을 생성한 경우 특정 임계값을 위반했을 때 알리도록 작업을 구성할 수도 있습니다. 예산 작업을 사용하면 이메일을 보내거나 SNS 주제에 메시지를 게시하거나 Slack과 같은 챗봇에 메시지를 보낼 수 있습니다. 자세한 내용은 AWS Budgets 작업 구성을 참조하세요.

karpenter.sh/do-not-disrupt 주석을 사용하여 Karpenter가 노드 프로비저닝을 해제하지 못하도록 합니다.

장기 실행 배치 작업 또는 상태 저장 애플리케이션과 같은 Karpenter 프로비저닝 노드에서 중요한 애플리케이션을 실행 중이고 노드의 TTL이 만료된 경우 인스턴스가 종료되면 애플리케이션이 중단됩니다. 포드에 `karpenter.sh/do-not-disrupt` 주석을 추가하면 포드가 종료되거나 `karpenter.sh/do-not-disrupt` 주석이 제거될 때까지 노드를 보존하도록 Karpenter에 지시합니다. 자세한 내용은 Distruption 설명서를 참조하세요.

노드에 남아 있는 유일한 비 데몬 세트 포드가 작업과 연결된 포드인 경우 Karpenter는 작업 상태가 성공 또는 실패인 한 해당 노드를 대상으로 지정하고 종료할 수 있습니다.

통합을 사용할 때 CPU가 아닌 모든 리소스에 대한 요청=제한 구성

포드 리소스 요청과 노드에서 할당 가능한 리소스의 양을 비교하여 일반적인 작업의 통합 및 예약. 리소스 제한은 고려되지 않습니다. 예를 들어 메모리 한도가 메모리 요청보다 큰 포드는 요청 위로 버스트할 수 있습니다. 동일한 노드의 여러 포드가 동시에 버스트되면 메모리 부족(OOM) 조건으로 인해 일부 포드가 종료될 수 있습니다. 통합은 요청을 고려할 때만 노드에 포드를 패키징하는 작업을 수행하므로 이러한 일이 발생할 가능성을 높일 수 있습니다.

LimitRanges를 사용하여 리소스 요청 및 제한에 대한 기본값 구성

Kubernetes는 기본 요청 또는 제한을 설정하지 않으므로 컨테이너가 기본 호스트, CPU 및 메모리에서 리소스를 소비하는 것은 제한되지 않습니다. Kubernetes 스케줄러는 포드의 총 요청 수(포드 컨테이너의 총 요청 수 또는 포드의 Init 컨테이너의 총 리소스 수 중 더 높은 쪽)를 확인하여 포드를 예약할 작업자 노드를 결정합니다. 마찬가지로 Karpenter는 포드의 요청을 고려하여 프로비저닝하는 인스턴스 유형을 결정합니다. 일부 포드에서 리소스 요청을 지정하지 않은 경우 제한 범위를 사용하여 네임스페이스에 적절한 기본값을 적용할 수 있습니다.

네임스페이스에 대한 기본 메모리 요청 및 제한 구성을 참조하세요.

모든 워크로드에 정확한 리소스 요청 적용

Karpenter는 워크로드 요구 사항에 대한 정보가 정확할 때 워크로드에 가장 적합한 노드를 시작할 수 있습니다. 이는 Karpenter의 통합 기능을 사용하는 경우 특히 중요합니다.

모든 워크로드에 대한 리소스 요청/제한 구성 및 크기 조정을 참조하세요.

CoreDNS 권장 사항

CoreDNS 구성을 업데이트하여 신뢰성 유지

Karpenter에서 관리하는 노드에 CoreDNS 포드를 배포할 때 수요에 맞게 새 노드를 빠르게 종료/생성하는 Karpenter의 동적 특성을 고려할 때 다음 모범 사례를 준수하는 것이 좋습니다.

CoreDNS lameduck 기간

CoreDNS 준비 프로브

이렇게 하면 DNS 쿼리가 아직 준비되지 않았거나 종료된 CoreDNS 포드로 전달되지 않습니다.

Karpenter 청사진

Karpenter는에 대한 컴퓨팅 용량을 Kubernetes 데이터 영역으로 프로비저닝하기 위해 애플리케이션 우선 접근 방식을 취하므로 이를 올바르게 구성하는 방법이 궁금할 수 있는 일반적인 워크로드 시나리오가 있습니다. Karpenter Blueprints는 여기에 설명된 모범 사례에 따른 일반적인 워크로드 시나리오 목록을 포함하는 리포지토리입니다. Karpenter가 구성된 EKS 클러스터를 생성하고 리포지토리에 포함된 각 블루프린트를 테스트하는 데 필요한 모든 리소스를 확보할 수 있습니다. 다양한 블루프린트를 결합하여 워크로드에 필요한 블루프린트(들)를 생성할 수 있습니다.

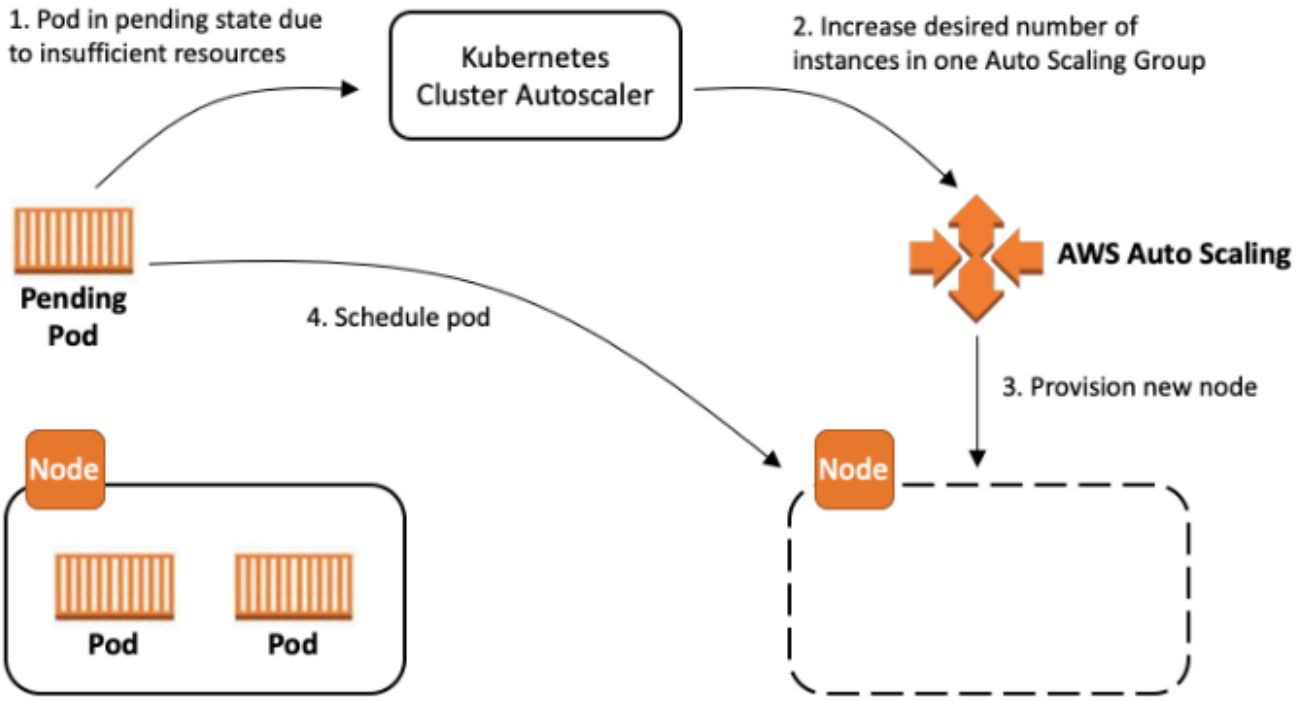
추가 리소스

- Karpenter Immersion Day 워크숍
- Karpenter 비용 최적화 워크숍
- EKS 워크숍 - Karpenter
- Karpenter와 Cluster Autoscaler 비교
- re:Invent 2023의 Karpenter 세션
- 자습서: Amazon EC2 스팟 및 Karpenter를 사용하여 Kubernetes 클러스터를 더 적게 실행

Cluster Autoscaler

개요

Kubernetes Cluster Autoscaler는 SIG Autoscaling에서 유지 관리하는 인기 있는 클러스터 Autoscaling 솔루션입니다. 이는 클러스터에 리소스를 낭비하지 않고 포드를 예약할 수 있는 충분한 노드가 있는지 확인하는 역할을 합니다. 예약에 실패한 포드와 사용률이 낮은 노드를 감시합니다. 그런 다음 클러스터에 변경 사항을 적용하기 전에 노드의 추가 또는 제거를 시뮬레이션합니다. Cluster Autoscaler 내의 AWS 클라우드 공급자 구현은 EC2 Auto Scaling 그룹의 `.DesiredReplicas` 필드를 제어합니다.



이 가이드에서는 Cluster Autoscaler를 구성하고 조직의 요구 사항에 맞는 최적의 장단점을 선택하기 위한 멘탈 모델을 제공합니다. 최상의 단일 구성은 없지만 성능, 확장성, 비용 및 가용성을 상쇄할 수 있는 구성 옵션 세트가 있습니다. 또한 이 가이드에서는 AWS에 대한 구성을 최적화하기 위한 팁과 모범 사례를 제공합니다.

용어집

이 문서 전체에서 다음 용어가 자주 사용됩니다. 이러한 용어는 광범위한 의미를 가질 수 있지만 이 문서의 목적상 아래 정의로 제한됩니다.

확장성은 Kubernetes 클러스터의 포드 및 노드 수가 증가함에 따라 Cluster Autoscaler가 얼마나 잘 작동하는지를 나타냅니다. 확장성 한도에 도달하면 Cluster Autoscaler의 성능과 기능이 저하됩니다. Cluster Autoscaler가 확장성 제한을 초과하면 클러스터에서 더 이상 노드를 추가하거나 제거하지 못할 수 있습니다.

성능은 Cluster Autoscaler가 조정 결정을 내리고 실행할 수 있는 속도를 나타냅니다. 완벽하게 작동하는 Cluster Autoscaler는 즉시 결정을 내리고 포드를 예약할 수 없게 되는 등 자극에 대응하여 조정 작업을 트리거합니다.

가용성이란 중단 없이 신속하게 포드를 예약할 수 있음을 의미합니다. 여기에는 새로 생성된 포드를 예약해야 하는 경우와 축소된 노드가 예약된 나머지 포드를 종료하는 경우가 포함됩니다.

비용은 스케일 아웃 및 스케일 인 이벤트의 결정에 따라 결정됩니다. 기존 노드가 사용률이 낮거나 수신 포드에 비해 너무 큰 새 노드가 추가되면 리소스가 낭비됩니다. 사용 사례에 따라 공격적인 스케일 다운 결정으로 인해 포드를 조기에 종료하는 데 따른 비용이 발생할 수 있습니다.

노드 그룹은 클러스터 내의 노드 그룹에 대한 추상적인 Kubernetes 개념입니다. 이는 실제 Kubernetes 리소스는 아니지만 Cluster Autoscaler, Cluster API 및 기타 구성 요소에 추상화로 존재합니다. 노드 그룹 내의 노드는 레이블 및 테인트와 같은 속성을 공유하지만 여러 가용 영역 또는 인스턴스 유형으로 구성될 수 있습니다.

EC2 Auto Scaling 그룹은 EC2에서 노드 그룹의 구현으로 사용할 수 있습니다. EC2 Auto Scaling 그룹은 Kubernetes 클러스터에 자동으로 조인하는 인스턴스를 시작하고 Kubernetes API의 해당 노드 리소스에 레이블과 테인트를 적용하도록 구성됩니다.

EC2 관리형 노드 그룹은 EC2에서 노드 그룹의 또 다른 구현입니다. EC2 Autoscaling 조정 그룹을 수동으로 구성하는 복잡성을 추상화하고 노드 버전 업그레이드 및 정상적인 노드 종료와 같은 추가 관리 기능을 제공합니다.

Cluster Autoscaler 작동

Cluster Autoscaler는 일반적으로 클러스터에 배포로 설치됩니다. 리더 선출을 사용하여 고가용성을 보장하지만 작업은 한 번에 단일 복제본에 의해 수행됩니다. 수평적으로 확장할 수 없습니다. 기본 설정의 경우 기본값은 제공된 설치 지침을 사용하여 즉시 사용할 수 있지만 몇 가지 유의해야 할 사항이 있습니다.

다음을 확인하세요.

- Cluster Autoscaler의 버전은 클러스터의 버전과 일치합니다. 교차 버전 호환성은 테스트되거나 지원되지 않습니다.
- 이 모드를 사용할 수 없는 특정 고급 사용 사례가 없는 한 자동 검색이 활성화됩니다.

IAM 역할에 대한 최소 권한 액세스 사용

Auto Discovery를 사용하는 경우 작업 `autoscaling:SetDesiredCapacity` 및를 현재 클러스터로 범위가 지정된 Auto Scaling 그룹`autoscaling:TerminateInstanceInAutoScalingGroup`으로 제한하여 최소 권한 액세스를 사용하는 것이 좋습니다.

이렇게 하면 `--node-group-auto-discovery` 인수가 태그를 사용하여 클러스터의 노드 그룹으로 범위가 축소되지 않은 경우에도 한 클러스터에서 실행되는 Cluster Autoscaler가 다른 클러스터의 노드 그룹을 수정하는 것을 방지할 수 있습니다(예: `k8s.io/cluster-autoscaler/<cluster-name>`).

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "autoscaling:SetDesiredCapacity",
                "autoscaling:TerminateInstanceInAutoScalingGroup"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/k8s.io/cluster-autoscaler/enabled": "true",
                    "aws:ResourceTag/k8s.io/cluster-autoscaler/my-cluster": "owned"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "autoscaling:DescribeAutoScalingGroups",
                "autoscaling:DescribeAutoScalingInstances",
                "autoscaling:DescribeLaunchConfigurations",
                "autoscaling:DescribeScalingActivities",
                "autoscaling:DescribeTags",
                "ec2:DescribeImages",
                "ec2:DescribeInstanceTypes",
                "ec2:DescribeLaunchTemplateVersions",
                "ec2:GetInstanceTypesFromInstanceRequirements",
                "eks:DescribeNodegroup"
            ],
            "Resource": "*"
        }
    ]
}
```

노드 그룹 구성

효과적인 오토 스케일링은 클러스터에 대한 노드 그룹 세트를 올바르게 구성하는 것으로 시작됩니다. 올바른 노드 그룹 세트를 선택하는 것은 가용성을 극대화하고 워크로드 전반의 비용을 절감하는 데 중요합니다. AWS는 많은 사용 사례에 유연한 EC2 Auto Scaling 그룹을 사용하여 노드 그룹을 구현합니

다. 하지만 Cluster Autoscaler는 노드 그룹에 대해 몇 가지 가정을 합니다. EC2 Auto Scaling 그룹 구성을 이러한 가정과 일관되게 유지하면 원치 않는 동작이 최소화됩니다.

다음을 확인하세요.

- 노드 그룹의 각 노드에는 레이블, 테인트 및 리소스와 같은 동일한 예약 속성이 있습니다.
  - MixedInstancePolicies의 경우 인스턴스 유형은 CPU, 메모리 및 GPU에 대해 동일한 셰이프여야 합니다.
  - 정책에 지정된 첫 번째 인스턴스 유형은 예약을 시뮬레이션하는 데 사용됩니다.
  - 정책에 리소스가 더 많은 추가 인스턴스 유형이 있는 경우 확장 후 리소스가 낭비될 수 있습니다.
  - 정책에 리소스가 적은 추가 인스턴스 유형이 있는 경우 포드가 인스턴스에서 예약하지 못할 수 있습니다.
- 노드가 많은 노드 그룹은 노드가 적은 많은 노드 그룹보다 선호됩니다. 이는 확장성에 가장 큰 영향을 미칩니다.
- 가능하면 두 시스템 모두 지원을 제공할 때 EC2 기능을 선호합니다(예: 리전, MixedInstancePolicy).

> **Note**
>
> EKS 관리형 노드 그룹을 사용하는 것이 좋습니다. 관리형 노드 그룹에는 자동 EC2 Auto Scaling 그룹 검색 및 정상적인 노드 종료와 같은 Cluster Autoscaler 기능을 비롯한 강력한 관리 기능이 제공됩니다.

성능 및 확장성 최적화

Autoscaling 알고리즘의 런타임 복잡성을 이해하면 노드가 1,000개 이상인 대규모 클러스터에서 원활하게 작동하도록 Cluster Autoscaler를 조정할 수 있습니다.

Cluster Autoscaler의 확장성을 조정하기 위한 기본 노브는 프로세스에 제공되는 리소스, 알고리즘의 스캔 간격, 클러스터의 노드 그룹 수입니다. 플러그인 복잡성 및 포드 수를 예약하는 등이 알고리즘의 실제 런타임 복잡성과 관련된 다른 요인이 있습니다. 이러한 파라미터는 클러스터의 워크로드에 자연스럽게 적용되며 쉽게 튜닝할 수 없으므로 구성할 수 없는 파라미터로 간주됩니다.

Cluster Autoscaler는 포드, 노드 및 노드 그룹을 포함하여 전체 클러스터의 상태를 메모리에 로드합니다. 각 스캔 간격에서 알고리즘은 예약할 수 없는 포드를 식별하고 각 노드 그룹에 대한 일정을 시뮬레이션합니다. 이러한 요소를 조정하면 사용 사례에 맞게 신중하게 고려해야 하는 다양한 장단점이 있습니다.

Cluster Autoscaler 수직 자동 조정

Cluster Autoscaler를 더 큰 클러스터로 확장하는 가장 간단한 방법은 배포에 대한 리소스 요청을 늘리는 것입니다. 대용량 클러스터의 경우 메모리와 CPU를 모두 늘려야 하지만 클러스터 크기에 따라 크게 다릅니다. Autoscaling 알고리즘은 모든 포드와 노드를 메모리에 저장하므로 경우에 따라 메모리 공간이 1기가바이트보다 클 수 있습니다. 리소스 증가는 일반적으로 수동으로 수행됩니다. 일정한 리소스 튜닝으로 인해 운영 부담이 발생하는 경우 Addon Resizer 또는 Vertical Pod Autoscaler를 사용하는 것이 좋습니다.

노드 그룹 수 줄이기

노드 그룹 수를 최소화하는 것은 Cluster Autoscaler가 대규모 클러스터에서 계속 잘 작동하도록 하는 한 가지 방법입니다. 이는 팀 또는 애플리케이션별로 노드 그룹을 구성하는 일부 조직에게는 어려울 수 있습니다. 이는 Kubernetes API에서 완전히 지원되지만 확장성을 위해 영향을 미치는 Cluster Autoscaler 안티 패턴으로 간주됩니다. 여러 노드 그룹(예: 스팟 또는 GPUs)을 사용해야 하는 이유는 많지만, 많은 경우 소수의 그룹을 사용하면서 동일한 효과를 내는 대체 설계가 있습니다.

다음을 확인하세요.

- 포드 격리는 노드 그룹이 아닌 네임스페이스를 사용하여 수행됩니다.
  - 이는 저신뢰 멀티 테넌트 클러스터에서는 가능하지 않을 수 있습니다.
  - Pod ResourceRequests 및 ResourceLimits는 리소스 경합을 방지하기 위해 적절하게 설정됩니다.
  - 인스턴스 유형이 클수록 빈 패킹이 최적화되고 시스템 포드 오버헤드가 줄어듭니다.
- NodeTaints NodeSelectors 포드를 규칙이 아닌 예외로 예약하는 데 사용됩니다.
- 리전 리소스는 여러 가용 영역이 있는 단일 EC2 Auto Scaling 그룹으로 정의됩니다.

스캔 간격 줄이기

스캔 간격이 짧으면(예: 10초) 포드를 예약할 수 없을 때 Cluster Autoscaler가 최대한 빨리 응답합니다. 그러나 스캔할 때마다 Kubernetes API 및 EC2 Auto Scaling 그룹 또는 EKS 관리형 노드 그룹 APIs 많이 발생합니다. 이러한 API 호출로 인해 Kubernetes 컨트롤 플레인의 속도가 제한되거나 심지어 서비스를 사용할 수 없게 될 수 있습니다.

기본 스캔 간격은 10초이지만 AWS에서 노드를 시작하는 데 새 인스턴스를 시작하는 데 훨씬 더 오래 걸립니다. 즉, 전체 확장 시간을 크게 늘리지 않고 간격을 늘릴 수 있습니다. 예를 들어 노드를 시작하는 데 2분이 걸리는 경우 간격을 1분으로 변경하면 38% 느린 스케일 업에서 API 호출이 6배 감소합니다.

노드 그룹 간 샤딩

Cluster Autoscaler는 특정 노드 그룹 집합에서 작동하도록 구성할 수 있습니다. 이 기능을 사용하면 각각 다른 노드 그룹 세트에서 작동하도록 구성된 Cluster Autoscaler의 여러 인스턴스를 배포할 수 있습니다. 이 전략을 사용하면 임의로 많은 수의 노드 그룹을 사용할 수 있으며, 확장성을 위한 거래 비용입니다. 성능 향상을 위한 최후의 수단으로만 사용하는 것이 좋습니다.

Cluster Autoscaler는 원래이 구성을 위해 설계되지 않았으므로 몇 가지 부작용이 있습니다. 샤드가 통신하지 않으므로 여러 오토 스케일러가 예약할 수 없는 포드를 예약하려고 시도할 수 있습니다. 이로 인해 여러 노드 그룹에서 불필요한 스케일 아웃이 발생할 수 있습니다. 이러한 추가 노드는 이후에 다시 축소됩니다`scale-down-delay`.

```
metadata:
  name: cluster-autoscaler
  namespace: cluster-autoscaler-1

...

--nodes=1:10:k8s-worker-asg-1
--nodes=1:10:k8s-worker-asg-2

---

metadata:
  name: cluster-autoscaler
  namespace: cluster-autoscaler-2

...

--nodes=1:10:k8s-worker-asg-3
--nodes=1:10:k8s-worker-asg-4
```

다음을 확인하세요.

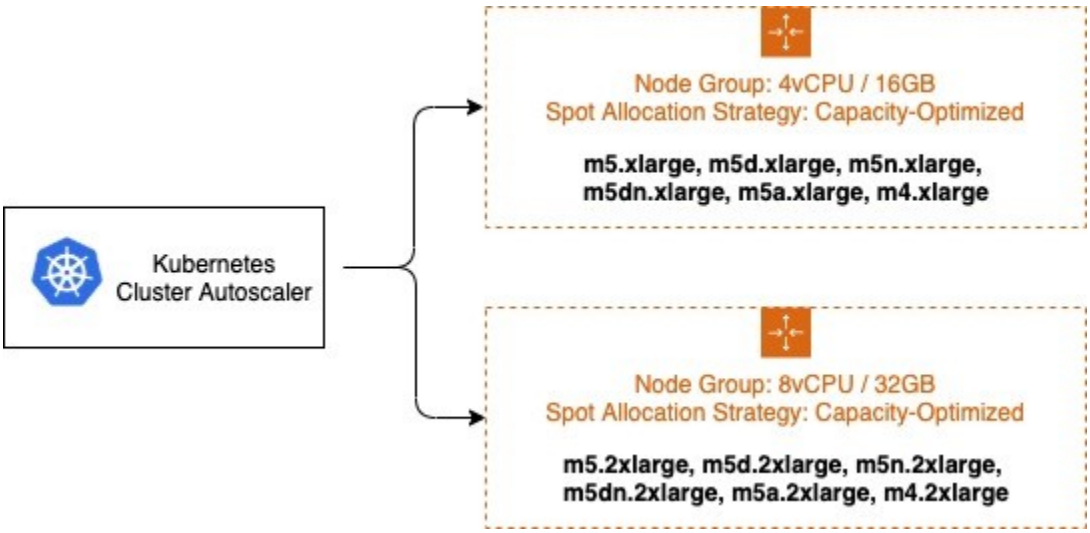
- 각 샤드는 고유한 EC2 Auto Scaling 그룹 세트를 가리키도록 구성됩니다.
- 각 샤드는 리더 선출 충돌을 방지하기 위해 별도의 네임스페이스에 배포됩니다.

비용 및 가용성 최적화

스팟 인스턴스

노드 그룹에서 스팟 인스턴스를 사용하고 온디맨드 가격에서 최대 90% 할인할 수 있습니다. 단, EC2에 용량이 다시 필요한 경우 언제든지 스팟 인스턴스가 중단될 수 있습니다. 사용 가능한 용량이 부족하여 EC2 Auto Scaling 그룹을 확장할 수 없는 경우 용량 부족 오류가 발생합니다. 여러 인스턴스 패밀리를 선택하여 다양성을 극대화하면 여러 스팟 용량 풀을 활용하여 원하는 규모를 달성할 가능성이 높아지고 스팟 인스턴스 중단이 클러스터 가용성에 미치는 영향이 줄어들 수 있습니다. 스팟 인스턴스와 혼합 인스턴스 정책은 노드 그룹 수를 늘리지 않고도 다양성을 높일 수 있는 좋은 방법입니다. 보장된 리소스가 필요한 경우 스팟 인스턴스 대신 온디맨드 인스턴스를 사용해야 합니다.

혼합 인스턴스 정책을 구성할 때 모든 인스턴스 유형에 유사한 리소스 용량이 있어야 합니다. 오토 스케일러의 스케줄링 시뮬레이터는 MixedInstancePolicy의 첫 번째 InstanceType을 사용합니다. MixedInstancePolicy 후속 인스턴스 유형이 더 크면 스케일 업 후 리소스가 낭비될 수 있습니다. 더 작으면 용량이 부족하여 새 인스턴스에서 포드를 예약하지 못할 수 있습니다. 예를 들어 M4, M5, M5a 및 M5n 인스턴스는 모두 CPU와 메모리 양이 비슷하며 MixedInstancePolicy에 적합한 후보입니다. EC2 인스턴스 선택기 도구는 유사한 인스턴스 유형을 식별하는 데 도움이 될 수 있습니다.



온디맨드 및 스팟 용량을 별도의 EC2 Auto Scaling 그룹으로 분리하는 것이 좋습니다. 이는 기본 용량 전략을 사용하는 것보다 선호됩니다. 예약 속성이 근본적으로 다르기 때문입니다. 스팟 인스턴스는 언제든지 중단되므로(EC2에 용량이 다시 필요한 경우) 사용자는 종종 선점 가능한 노드를 테인트하므로 선점 동작에 대한 명시적 포드 허용이 필요합니다. 이러한 테인트는 노드의 예약 속성이 다르므로 여러 EC2 Auto Scaling 그룹으로 분리해야 합니다.

Cluster Autoscaler에는 확장기의 개념이 있습니다. 확장기는 확장할 노드 그룹을 선택하기 위한 다양한 전략을 제공합니다. 전략`--expander=least-waste`은 좋은 범용 기본값이며 스팟 인스턴스 다각

화에 여러 노드 그룹을 사용하려는 경우(위 이미지에 설명된 대로) 조정 활동 후 가장 잘 활용될 그룹을 조정하여 노드 그룹을 추가로 비용 최적화하는 데 도움이 될 수 있습니다.

노드 그룹/ASG 우선 순위 지정

Priority 확장기를 사용하여 우선 순위 기반 Autoscaling을 구성할 수도 있습니다.를 `--expander=priority` 사용하면 클러스터가 노드 그룹/ASG의 우선 순위를 지정할 수 있으며, 어떤 이유로든 확장할 수 없는 경우 우선 순위가 지정된 목록에서 다음 노드 그룹을 선택합니다. 이는 예를 들어 GPU가 워크로드에 최적의 성능을 제공하기 때문에 P3 인스턴스 유형을 사용하려는 상황에서 유용하지만, 두 번째 옵션으로 P2 인스턴스 유형을 사용할 수도 있습니다.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-autoscaler-priority-expander
  namespace: kube-system
data:
  priorities: |-
    10:
      - .*p2-node-group.*
    50:
      - .*p3-node-group.*
```

Cluster Autoscaler는 p3-node-group 이름과 일치하는 EC2 Auto Scaling 그룹을 확장하려고 시도합니다. 이 작업이 내에서 성공하지 못하면 p2-node-group 이름과 일치하는 EC2 Auto Scaling 그룹을 조정하려고 시도`--max-node-provision-time`합니다. 이 값은 기본적으로 15분이며 응답성이 뛰어난 노드 그룹 선택을 위해 줄일 수 있지만 값이 너무 낮으면 불필요한 스케일 아웃이 발생할 수 있습니다.

오버프로비저닝

Cluster Autoscaler는 노드가 필요한 경우에만 클러스터에 추가되고 사용하지 않으면 제거되도록 하여 비용을 최소화합니다. 이는 많은 포드가 노드 스케일 업을 기다렸다가 예약할 수 있기 때문에 배포 지연 시간에 상당한 영향을 미칩니다. 노드를 사용할 수 있게 되려면 몇 분 정도 걸릴 수 있으며, 이로 인해 포드 예약 지연 시간이 크게 늘어날 수 있습니다.

예약 대기 시간이 늘어나는 것을 감수하고 오버프로비저닝을 사용하여 이를 완화할 수 있습니다. 오버프로비저닝은 클러스터의 공간을 차지하는 음의 우선 순위가 있는 임시 포드를 사용하여 구현됩니다. 새로 생성된 포드를 예약할 수 없고 우선 순위가 높으면 임시 포드가 미리 선점되어 공간을 만듭니다. 그런 다음 임시 포드를 예약할 수 없게 되어 Cluster Autoscaler가 과도하게 프로비저닝된 새 노드를 확장하도록 트리거합니다.

오버프로비저닝에는 덜 명확한 다른 이점이 있습니다. 오버프로비저닝이 없으면 활용도가 높은 클러스터의 부작용 중 하나는 포드가 포드 또는 노드 선호도 `preferredDuringSchedulingIgnoredDuringExecution` 규칙을 사용하여 최적의 스케줄링 결정을 내리는 것입니다. 이에 대한 일반적인 사용 사례는 AntiAffinity를 사용하여 가용 영역에서 가용성이 높은 애플리케이션의 포드를 분리하는 것입니다. 오버프로비저닝은 올바른 영역의 노드를 사용할 수 있는 가능성을 크게 높일 수 있습니다.

과다 프로비저닝된 용량의 양은 조직의 신중한 비즈니스 결정입니다. 핵심은 성능과 비용의 장단점입니다. 이 결정을 내리는 한 가지 방법은 평균 스케일 업 빈도를 결정하고 이를 새 노드를 스케일 업하는 데 걸리는 시간으로 나누는 것입니다. 예를 들어 평균적으로 30초마다 새 노드가 필요하고 EC2가 새 노드를 프로비저닝하는 데 30초가 걸리는 경우 오버프로비저닝의 단일 노드를 사용하면 항상 추가 노드를 사용할 수 있으므로 단일 추가 EC2 인스턴스의 비용으로 예약 지연 시간을 30초 단축할 수 있습니다. 영역 예약 결정을 개선하려면 EC2 Auto Scaling 그룹의 가용 영역 수와 동일한 수의 노드를 오버프로비저닝하여 스케줄러가 수신 포드에 가장 적합한 영역을 선택할 수 있도록 합니다.

축소 제거 방지

일부 워크로드는 제거 비용이 많이 듭니다. 빅 데이터 분석, 기계 학습 작업 및 테스트 실행기는 결국 완료되지만 중단되면 다시 시작해야 합니다. Cluster Autoscaler는 scale-down-utilization-threshold 아래의 노드를 스케일 다운하려고 시도하며, 이로 인해 노드에 남아 있는 포드가 중단됩니다. 제거 비용이 많이 드는 포드가 Cluster Autoscaler에서 인식하는 레이블로 보호되도록 하면 이를 방지할 수 있습니다.

다음을 확인하세요.

- 제거 비용이 많이 드는 포드에는 주석이 있습니다. `cluster-autoscaler.kubernetes.io/safe-to-evict=false`

고급 사용 사례

EBS 볼륨

영구 스토리지는 데이터베이스 또는 분산 캐시와 같은 상태 저장 애플리케이션을 구축하는 데 매우 중요합니다. EBS 볼륨은 Kubernetes에서이 사용 사례를 활성화하지만 특정 영역으로 제한됩니다. 이러한 애플리케이션은 각 AZs 사용하여 여러 AZ에 샤딩되는 경우 가용성이 높을 수 있습니다. 그런 다음 Cluster Autoscaler는 EC2 Autoscaling 그룹의 조정 균형을 맞출 수 있습니다.

다음을 확인하세요.

- 노드 그룹 밸런싱은 `balance-similar-node-groups=true`를 설정하여 활성화합니다.
- 노드 그룹은 서로 다른 가용 영역과 EBS 볼륨을 제외하고 동일한 설정으로 구성됩니다.

공동 예약

기계 학습 분산 훈련 작업은 동일한 영역 노드 구성의 최소화된 지연 시간을 통해 상당한 이점을 얻을 수 있습니다. 이러한 워크로드는 특정 영역에 여러 포드를 배포합니다. 이는를 사용하여 공동 예약된 모든 포드에 대해 포드 선호도를 설정하거나 노드 선호도를 설정하여 달성할 수 있습니다`topologyKey: failure-domain.beta.kubernetes.io/zone`. 그런 다음 Cluster Autoscaler는 수요에 맞게 특정 영역을 확장합니다. 여러 EC2 Auto Scaling 그룹을 가용 영역당 하나씩 할당하여 공동 예약된 전체 워크로드에 대한 장애 조치를 활성화할 수 있습니다.

다음을 확인하세요.

- 노드 그룹 밸런싱은 `balance-similar-node-groups=false`를 설정하여 활성화합니다.
- 노드 선호도 및/또는 포드 선점은 클러스터에 리전 노드 그룹과 영역 노드 그룹이 모두 포함된 경우에 사용됩니다.
  - 노드 선호도를 사용하여 리전 포드가 영역 노드 그룹을 피하도록 강제하거나 장려하거나 그 반대의 경우도 마찬가지입니다.
  - 영역 포드가 리전 노드 그룹에 예약되면 리전 포드의 용량이 불균형해집니다.
  - 영역 워크로드가 중단 및 재배치를 허용할 수 있는 경우 리전별로 규모가 조정된 포드가 경쟁이 덜한 영역에서 선점 및 일정을 강제로 조정할 수 있도록 포드 선점을 구성합니다.

액셀러레이터

일부 클러스터는 GPU와 같은 특수 하드웨어 액셀러레이터를 활용합니다. 스케일 아웃 시 액셀러레이터 디바이스 플러그인이 리소스를 클러스터에 알리는 데 몇 분 정도 걸릴 수 있습니다. Cluster Autoscaler는이 노드에 액셀러레이터가 있다고 시뮬레이션했지만 액셀러레이터가 준비되고 노드의 사용 가능한 리소스를 업데이트할 때까지 보류 중인 포드는 노드에서 예약할 수 없습니다. 이로 인해 반복적 불필요한 확장이 발생할 수 있습니다.

또한 액셀러레이터가 사용되지 않더라도 액셀러레이터가 있고 CPU 또는 메모리 사용률이 높은 노드는 스케일 다운에 고려되지 않습니다. 이 동작은 액셀러레이터의 상대적 비용 때문에 비용이 많이 들 수 있습니다. 대신 Cluster Autoscaler는 액셀러레이터가 점유되지 않은 경우 스케일 다운할 노드를 고려하는 특수 규칙을 적용할 수 있습니다.

이러한 경우에 올바른 동작을 보장하기 위해 클러스터에 조인하기 전에 노드에 레이블을 지정하도록 액셀러레이터 노드의 kubelet을 구성할 수 있습니다. Cluster Autoscaler는 이 레이블 선택기를 사용하여 액셀러레이터 최적화 동작을 트리거합니다.

다음을 확인하세요.

- GPU 노드용 Kubelet은 로 구성됩니다. `--node-labels k8s.amazonaws.com/accelerator=$ACCELERATOR_TYPE`
- Accelerator가 있는 노드는 위에서 언급한 것과 동일한 예약 속성 규칙을 준수합니다.

0부터 조정

Cluster Autoscaler는 노드 그룹을 0에서 0으로 조정하여 상당한 비용 절감 효과를 얻을 수 있습니다. LaunchConfiguration 또는 LaunchTemplate에 지정된 InstanceType을 검사하여 Auto Scaling 그룹의 CPU, 메모리 및 GPU 리소스를 감지합니다. 일부 포드에는 `WindowsENI` LaunchConfiguration에서 검색할 수 없는 `PrivateIPv4Address` 또는 특정 NodeSelectors 또는 Taints와 같은 추가 리소스가 필요합니다. Cluster Autoscaler는 EC2 Auto Scaling 그룹의 태그에서 이러한 요소를 검색하여 이러한 요소를 설명할 수 있습니다. 예제:

```
Key: k8s.io/cluster-autoscaler/node-template/resources/$RESOURCE_NAME
Value: 5
Key: k8s.io/cluster-autoscaler/node-template/label/$LABEL_KEY
Value: $LABEL_VALUE
Key: k8s.io/cluster-autoscaler/node-template/taint/$TAINT_KEY
Value: NoSchedule
```

> **Note**
>
> 0으로 조정하면 용량이 EC2로 반환되고 향후 사용할 수 없게 될 수 있습니다.

추가 파라미터

Cluster Autoscaler의 동작과 성능을 조정하는 데 사용할 수 있는 많은 구성 옵션이 있습니다. 전체 파라미터 목록은 GitHub에서 확인할 수 있습니다.

파라미터	설명	기본값
스캔 간격	클러스터가 스케일 업 또는 스케일 다운을 위해 재평가되는 빈도	10초
max-empty-bulk-delete	동시에 삭제할 수 있는 최대 빈 노드 수입니다.	10
scale-down-delay-after-add	스케일 업 후 스케일 다운 평가가 재개되는 기간	10분
scale-down-delay-after-delete	축소 평가가 재개된 후 노드 삭제가 시작되는 기간은 기본적으로 scan-interval로 설정됩니다.	스캔 간격
scale-down-delay-after-failure	축소 실패 후 축소 평가가 재개되는 기간	3분
scale-down-unneeded-time	스케일 다운에 적합하기 전에 노드가 필요하지 않은 기간	10분
scale-down-unready-time	스케일 다운에 적합하기 전에 준비되지 않은 노드가 필요하지 않은 기간	20분
scale-down-utilization-threshold	노드 사용률 수준 - 요청된 리소스의 합계를 용량으로 나눈 값으로 정의되며 노드를 축소 대상으로 고려할 수 있습니다.	0.5
scale-down-non-empty-candidates-count	한 번의 반복에서 드레이닝을 통한 스케일 다운 후보로 간주되는 비어 있지 않은 노드의 최대 수입니다. 값이 낮을수록 CA 응답성이 향상되지만 스케일 다운 지연 시간이 느려질 수 있습니다. 값이 높을수록 큰 클	30

파라미터	설명	기본값
	러스터(수백 개의 노드)의 CA 성능에 영향을 미칠 수 있습니다. 이 휴리스틱을 끄려면 양수가 아닌 값으로 설정합니다. CA는 고려하는 노드 수를 제한하지 않습니다.”	
scale-down-candidates-pool-ratio	이전 반복의 일부 후보가 더 이상 유효하지 않을 때 스케일 다운을 위해 비어 있지 않은 추가 후보로 간주되는 노드의 비율입니다. 값이 낮을수록 CA 응답성이 향상되지만 스케일 다운 지연 시간이 느려질 수 있습니다. 값이 높을수록 큰 클러스터(수백 개의 노드)의 CA 성능에 영향을 미칠 수 있습니다. 이 휴리스틱을 끄려면 1.0으로 설정합니다. CA는 모든 노드를 추가 후보로 사용합니다.	0.1
scale-down-candidates-pool-min-count	이전 반복의 일부 후보가 더 이상 유효하지 않을 때 스케일 다운을 위해 비어 있지 않은 추가 후보로 간주되는 최소 노드 수입니다. 추가 후보의 풀 크기를 계산할 때 `max(#nodes * scale-down-candidates-pool-ratio, scale-down-candidates-pool-min-count)`	50

추가 리소스

이 페이지에는 Cluster Autoscaler 프레젠테이션 및 데모 목록이 포함되어 있습니다. 여기에 프레젠테이션 또는 데모를 추가하려면 풀 요청을 보내세요.

프레젠테이션/데모	발표자
Kubernetes의 오토 스케일링 및 비용 최적화: 0~100	Guy Templeton, Skyscanner & Jiaxin Shan, Amazon
SIG-Autoscaling 심층 분석	Maciek Pytel 및 Marcin Wielgus

참조

- https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md
- https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/aws/README.md
- https://github.com/aws/amazon-ec2-instance-selector
- https://github.com/aws/aws-node-termination-handler

신뢰성 모범 사례

이 섹션에서는 EKS에서 실행되는 워크로드의 복원력과 가용성을 높이는 방법에 대한 지침을 제공합니다.

이 설명서의 사용법

이 가이드는 EKS에서 가용성과 내결함성이 뛰어난 서비스를 개발하고 운영하려는 개발자와 아키텍트를 대상으로 합니다. 이 가이드는 더 쉽게 사용할 수 있도록 다양한 주제 영역으로 구성되어 있습니다. 각 주제는 간략한 개요로 시작하여 EKS 클러스터의 신뢰성을 위한 권장 사항 및 모범 사례 목록으로 이어집니다.

소개

EKS의 신뢰성 모범 사례는 다음 주제로 그룹화되었습니다.

- 애플리케이션
- 제어 플레인
- 데이터 영역

시스템을 안정적으로 만드는 요소는 무엇입니까? 일정 기간 동안 환경이 변경되더라도 시스템이 일관되게 작동하고 수요를 충족할 수 있는 경우 신뢰할 수 있다고 할 수 있습니다. 이를 위해서는 시스템에서 장애를 감지하고 자동으로 복구하며 수요에 따라 확장할 수 있어야 합니다.

고객은 Kubernetes를 기반으로 미션 크리티컬 애플리케이션 및 서비스를 안정적으로 운영할 수 있습니다. 그러나 컨테이너 기반 애플리케이션 설계 원칙을 통합하는 것 외에도 워크로드를 안정적으로 실행하려면 신뢰할 수 있는 인프라도 필요합니다. Kubernetes에서 인프라는 컨트롤 플레인과 데이터 플레인으로 구성됩니다.

EKS는 가용성과 내결함성을 높이도록 설계된 프로덕션급 Kubernetes 컨트롤 플레인을 제공합니다.

EKS에서 AWS는 Kubernetes 컨트롤 플레인의 신뢰성을 담당합니다. EKS는 AWS 리전의 세 가지 가용 영역에서 Kubernetes 컨트롤 플레인을 실행합니다. Kubernetes API 서버 및 etcd 클러스터의 가용성과 확장성을 자동으로 관리합니다.

데이터 영역의 신뢰성에 대한 책임은 사용자, 고객 및 AWS 간에 공유됩니다. EKS는 Kubernetes 데이터 영역을 배포하기 위한 네 가지 작업자 노드 옵션을 제공합니다.

가장 관리형 옵션인 EKS Auto Mode는 관리형 컴퓨팅, 네트워킹 및 스토리지 기능을 제공하는 동시에 데이터 영역의 프로비저닝, 규모 조정 및 업데이트를 처리합니다. 자동 모드 AMIs 자주 릴리스되고 클러스터는 CVE 수정 및 보안 패치를 배포하기 위해 자동으로 최신 AMI로 업데이트됩니다. Auto Mode NodePools에서 중단 제어를 구성하여 이러한 상황이 발생하는 시기를 제어할 수 있습니다.

Fargate는 노드당 하나의 포드를 실행하여 데이터 영역의 프로비저닝 및 크기 조정을 처리합니다. 세 번째 옵션인 관리형 노드 그룹은 데이터 영역의 프로비저닝 및 업데이트를 처리합니다. 마지막으로 자체 관리형 노드는 데이터 영역의 최소 관리형 옵션입니다. AWS 관리형 데이터 영역을 많이 사용할수록 책임이 줄어듭니다.

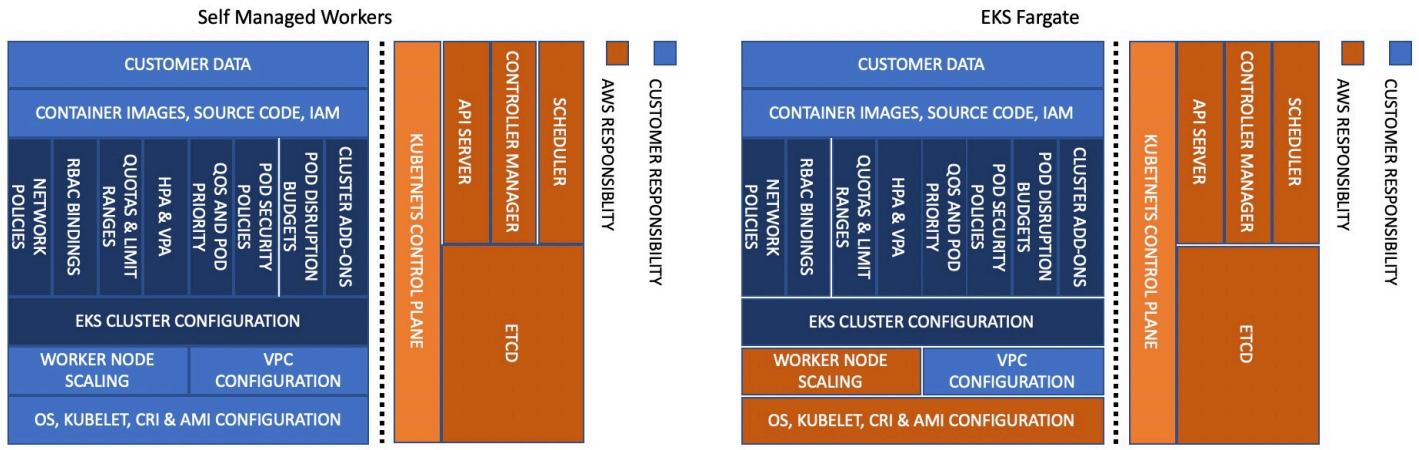
관리형 노드 그룹은 EC2 노드의 프로비저닝 및 수명 주기 관리를 자동화합니다. EKS API(EKS 콘솔, AWS API, AWS CLI, CloudFormation, Terraform 또는 사용`eksctl`)를 사용하여 관리형 노드를 생성, 확장 및 업그레이드할 수 있습니다. 관리형 노드는 계정에서 EKS 최적화 Amazon Linux 2 EC2 인스턴스를 실행하며 SSH 액세스를 활성화하여 사용자 지정 소프트웨어 패키지를 설치할 수 있습니다. 관리형 노드를 프로비저닝하면 여러 가용 영역에 걸쳐 있을 수 있는 EKS 관리형 Auto Scaling 그룹의 일부로 실행됩니다. 관리형 노드를 생성할 때 제공하는 서브넷을 통해 이를 제어합니다. 또한 EKS는 관리형 노드에 자동으로 태그를 지정하여 Cluster Autoscaler와 함께 사용할 수 있습니다.

Amazon EKS는 관리형 노드 그룹의 CVE 및 보안 패치에 대한 공동 책임 모델을 따릅니다. 관리형 노드는 Amazon EKS 최적화 AMIs를 실행하므로 Amazon EKS는 버그 수정 시 이러한 AMIs의 패치 버전을 빌드할 책임이 있습니다. 그러나 이렇게 패치된 AMI 버전은 사용자가 관리형 노드 그룹에 배포해야 합니다.

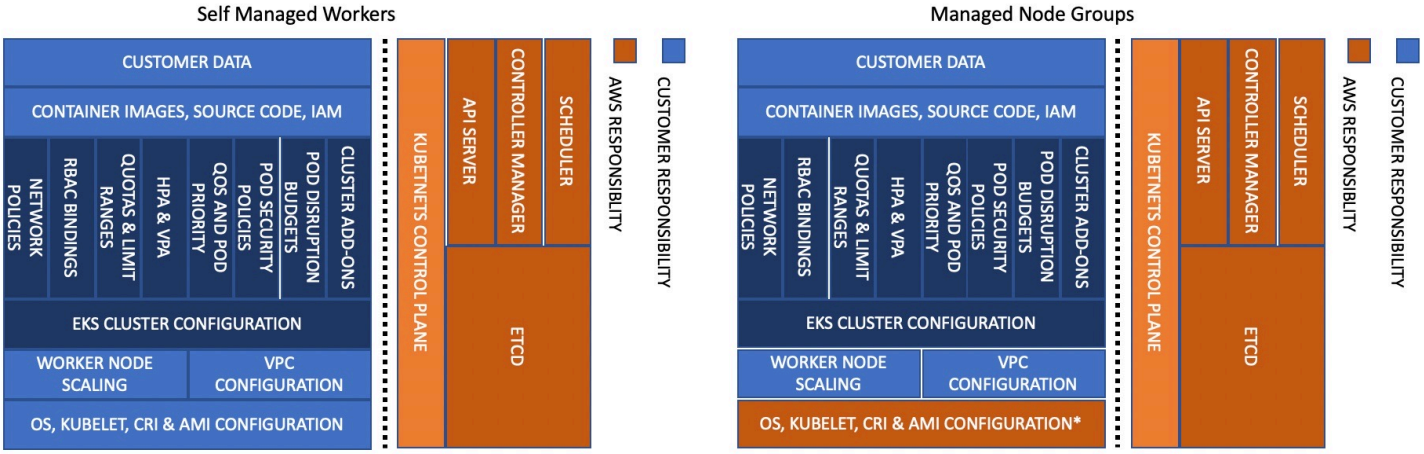
또한 EKS는 노드 업데이트를 관리하지만 업데이트 프로세스를 시작해야 합니다. 관리형 노드 업데이트 프로세스는 EKS 설명서에 설명되어 있습니다.

자체 관리형 노드를 실행하는 경우 Amazon EKS 최적화 Linux AMI를 사용하여 작업자 노드를 생성할 수 있습니다. AMI와 노드를 패치하고 업그레이드하는 것은 사용자의 책임입니다. `eksctl`, CloudFormation 또는 인프라를 코드 도구로 사용하여 자체 관리형 노드를 프로비저닝하는 것이 가장 좋습니다. 이렇게 하면 자체 관리형 노드를 쉽게 업그레이드할 수 있기 때문입니다. 마이그레이션 프로세스는 이전 노드 그룹을 로 테인팅`NoSchedule`하고 새 스택이 기존 포드 워크로드를 수락할 준비가 된 후 노드를 비우므로 작업자 노드를 업데이트할 때 새 노드로 마이그레이션하는 것이 좋습니다. 그러나 자체 관리형 노드의 현재 위치 업그레이드를 수행할 수도 있습니다.

공동 책임 모델 - Fargate



공동 책임 모델 - MNG



이 가이드에는 EKS 데이터 영역, Kubernetes 코어 구성 요소 및 애플리케이션의 신뢰성을 개선하는 데 사용할 수 있는 권장 사항 세트가 포함되어 있습니다.

Feedback

이 가이드는 광범위한 EKS/Kubernetes 커뮤니티에서 직접적인 피드백과 제안을 수집하기 위해 GitHub에서 릴리스됩니다. 가이드에 포함해야 한다고 생각되는 모범 사례가 있는 경우 GitHub 리포지토리에 문제를 제기하거나 PR을 제출하세요. 새로운 기능이 서비스에 추가되거나 새로운 모범 사례가 발전할 때 가이드를 주기적으로 업데이트하려고 합니다.

가용성이 높은 애플리케이션 실행

고객은 변경 시, 특히 트래픽이 급증하는 경우를 포함하여 애플리케이션을 항상 사용할 수 있기를 기대합니다. 확장 가능하고 복원력이 뛰어난 아키텍처는 중단 없이 애플리케이션과 서비스를 실행하여 사용자를 만족시킵니다. 확장 가능한 인프라는 비즈니스 요구 사항에 따라 확장 및 축소됩니다. 단일 장애 지점을 제거하는 것은 애플리케이션의 가용성을 개선하고 복원력을 높이기 위한 중요한 단계입니다.

Kubernetes를 사용하면 가용성과 복원력이 뛰어난 방식으로 애플리케이션을 운영하고 실행할 수 있습니다. 선언적 관리를 통해 애플리케이션을 설정한 후 Kubernetes는 현재 상태를 원하는 상태와 지속적으로 일치시키려고 시도합니다.

권장 사항

포드 중단 예산 구성

포드 중단 예산은 애플리케이션에서 발생할 수 있는 동시 중단의 양을 제한하는 데 사용됩니다. 워크로드의 일부를 항상 사용할 수 있는 것이 중요한 경우 워크로드에 맞게 구성해야 합니다. EKS Auto Mode, Karpenter 및 Cluster Autoscaler는 스케일 다운 시 구성된 포드 중단 예산을 인식하고 준수합니다. EKS Auto Mode, Karpenter 및 Managed Node Groups는 노드 업데이트 시 포드 중단 예산도 준수합니다.

singleton 포드 실행 방지

전체 애플리케이션이 단일 포드에서 실행되는 경우 해당 포드가 종료되면 애플리케이션을 사용할 수 없습니다. 개별 포드를 사용하여 애플리케이션을 배포하는 대신 배포를 생성합니다. 배포에 의해 생성된 포드가 실패하거나 종료되면 배포 컨트롤러는 지정된 수의 복제본 포드가 항상 실행되도록 새 포드를 시작합니다.

여러 복제본 실행

배포를 사용하여 앱의 여러 복제본 포드를 실행하면 가용성이 높은 방식으로 실행하는 데 도움이 됩니다. 한 복제본에 장애가 발생하면 Kubernetes가 손실을 보충하기 위해 다른 포드를 생성할 때까지 용량을 줄이더라도 나머지 복제본은 계속 작동합니다. 또한 Horizontal Pod Autoscaler를 사용하여 워크로드 수요에 따라 복제본을 자동으로 확장할 수 있습니다.

노드 간 복제본 예약

모든 복제본이 동일한 노드에서 실행 중이고 노드를 사용할 수 없게 되는 경우 여러 복제본을 실행하는 것은 매우 유용하지 않습니다. 포드 안티 친화도 또는 포드 토폴로지 분산 제약 조건을 사용하여 여러 작업자 노드에 배포의 복제본을 분산하는 것이 좋습니다.

여러 AZs.

포드 안티 친화도 규칙 사용

아래 매니페스트는 Kubernetes 스케줄러가 별도의 노드와 AZ에 포드를 배치하는 것을 선호하도록 지시합니다. AZs 개별 노드 또는 AZ가 필요하지 않습니다. 그럴 경우 각 AZ에서 실행 중인 포드가 있으면 Kubernetes가 포드를 예약할 수 없기 때문입니다. 애플리케이션에 복제본이 3개만 필요한 경우 `requiredDuringSchedulingIgnoredDuringExecution`에를 사용할 수 있으며`topologyKey: topology.kubernetes.io/zone`, Kubernetes 스케줄러는 동일한 AZ에서 포드 2개를 예약하지 않습니다.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spread-host-az
  labels:
    app: web-server
spec:
  replicas: 4
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
    spec:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
```

```
                - web-server
            topologyKey: topology.kubernetes.io/zone
          weight: 100
        - podAffinityTerm:
            labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - web-server
            topologyKey: kubernetes.io/hostname
          weight: 99
    containers:
    - name: web-app
      image: nginx:1.16-alpine
```

포드 토폴로지 분산 제약 조건 사용

포드 친화도 방지 규칙과 마찬가지로 포드 토폴로지 분산 제약 조건을 사용하면 호스트 또는 AZs와 같은 다양한 장애(또는 토폴로지) 도메인에서 애플리케이션을 사용할 수 있습니다. 이 접근 방식은 내결함성 및 가용성을 보장하려고 할 때 서로 다른 각 토폴로지 도메인에 여러 복제본이 있어 매우 효과적입니다. 반면에 포드 안티 친화도 규칙은 토폴로지 도메인에 단일 복제본이 있는 결과를 쉽게 생성할 수 있습니다. 서로에 대한 안티 친화도가 있는 포드는 반발 효과가 있기 때문입니다. 이러한 경우 전용 노드의 단일 복제본은 내결함성이나 리소스 사용에 적합하지 않습니다. 토폴로지 분산 제약 조건을 사용하면 스케줄러가 토폴로지 도메인 전체에 적용해야 하는 분산 또는 배포를 더 잘 제어할 수 있습니다. 이 접근 방식에 사용할 몇 가지 중요한 속성은 다음과 같습니다.

1. maxSkew는 토폴로지 도메인에서 사물이 고르지 않을 수 있는 최대 지점을 제어하거나 결정하는 데 사용됩니다. 예를 들어 애플리케이션에 10개의 복제본이 있고 3AZs에 배포된 경우 균등한 분산을 얻을 수 없지만 배포가 얼마나 고르지 않은지에 영향을 미칠 수 있습니다. 이 경우는 1에서 10 사이의 값이 될 maxSkew 수 있습니다. 값이 1이면와 같은 스프레드 4,3,3 3,4,3 또는 3AZs3,3,4로 끝날 수 있습니다. 반대로 값이 10이면와 같은 스프레드 10,0,0 0,10,0 또는 0,0,10 3AZs로 끝날 수 있습니다.
2. topologyKey는 노드 레이블 중 하나의 키이며 포드 배포에 사용해야 하는 토폴로지 도메인 유형을 정의합니다. 예를 들어 영역 분산에는 다음과 같은 키-값 페어가 있습니다.

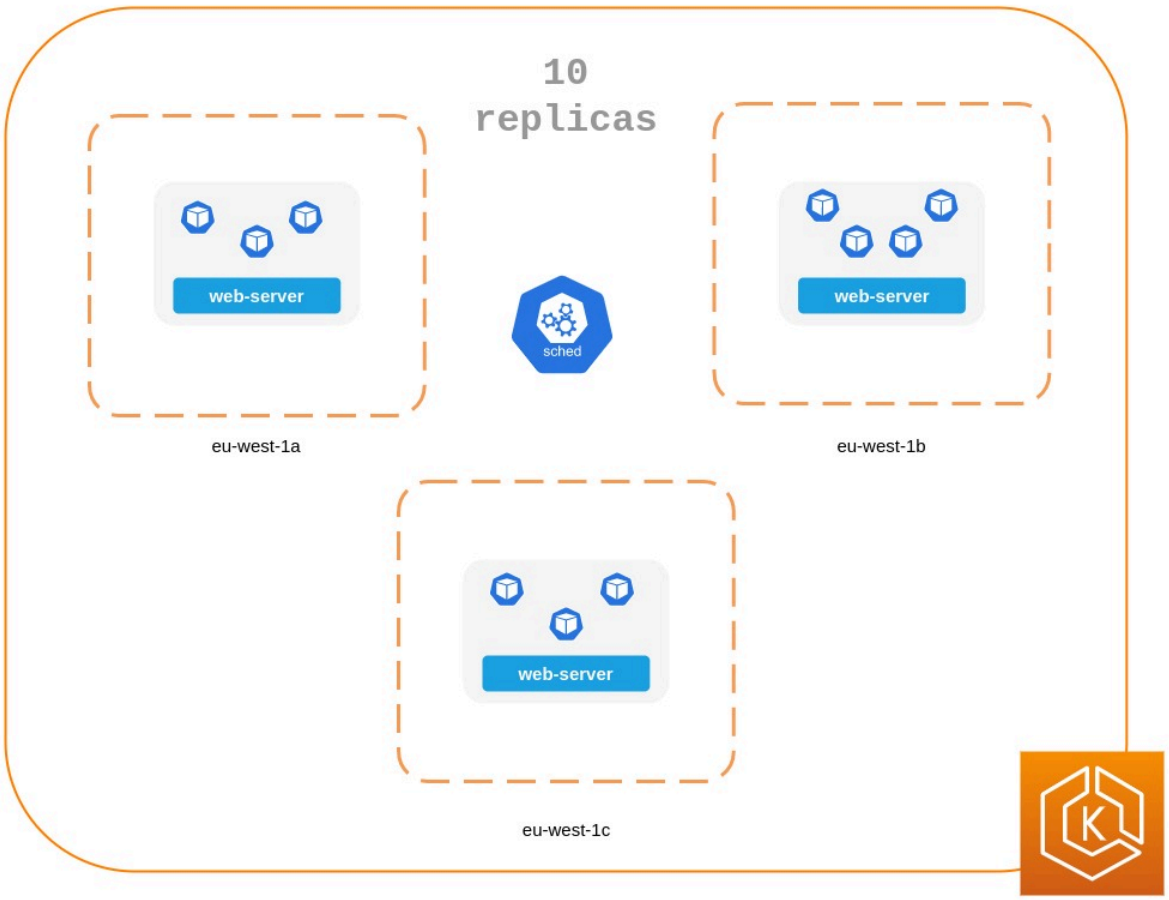
   ```
   topologyKey: "topology.kubernetes.io/zone"
   ```

3. whenUnsatisfiable 속성은 원하는 제약 조건을 충족할 수 없는 경우 스케줄러가 응답할 방법을 결정하는 데 사용됩니다.

4. `labelSelector`는 지정한 제약 조건에 따라 포드를 배치할 위치를 결정할 때 스케줄러가 이를 인식할 수 있도록 일치하는 포드를 찾는 데 사용됩니다.

위의 필드 외에도 Kubernetes 설명서에서 자세히 읽을 수 있는 다른 필드가 있습니다.

3AZs에 포드 토폴로지 분산 제약 조건




```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spread-host-az
  labels:
    app: web-server
spec:
  replicas: 10
  selector:
```

```
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
    spec:
      topologySpreadConstraints:
      - maxSkew: 1
        topologyKey: "topology.kubernetes.io/zone"
        whenUnsatisfiable: ScheduleAnyway
        labelSelector:
          matchLabels:
            app: express-test
      containers:
      - name: web-app
        image: nginx:1.16-alpine
```

Kubernetes 지표 서버 실행

애플리케이션을 확장하는 데 도움이 되도록 Kubernetes 지표 서버를 설치합니다. HPA 및 VPA와 같은 Kubernetes Autoscaler 추가 기능은 애플리케이션의 지표를 추적하여 크기를 조정해야 합니다. 지표 서버는 조정 결정을 내리는 데 사용할 수 있는 리소스 지표를 수집합니다. 지표는 kubelet에서 수집되어 지표 API 형식으로 제공됩니다.

지표 서버는 데이터를 보관하지 않으며 모니터링 솔루션이 아닙니다. 그 목적은 CPU 및 메모리 사용량 지표를 다른 시스템에 노출하는 것입니다. 시간 경과에 따라 애플리케이션의 상태를 추적하려면 Prometheus 또는 Amazon CloudWatch와 같은 모니터링 도구가 필요합니다.

EKS 설명서에 따라 EKS 클러스터에 metrics-server를 설치합니다.

Horizontal Pod Autoscaler(HPA)

HPA는 수요에 따라 애플리케이션을 자동으로 확장할 수 있으며 트래픽이 많을 때 고객에게 영향을 주지 않도록 도와줍니다. 리소스 지표를 제공하는 APIs에서 지표를 주기적으로 쿼리하는 Kubernetes의 제어 루프로 구현됩니다.

HPA는 다음 APIs에서 지표를 검색할 수 있습니다. 1. 리소스 지표 API라고`metrics.k8s.io`도 함 - 포드 2에 대한 CPU 및 메모리 사용량을 제공합니다. `custom.metrics.k8s.io` - Prometheus와 같은 다른 지표 수집기의 지표를 제공합니다. 이러한 지표는 Kubernetes 클러스터 내부에 있습니다. 3. `external.metrics.k8s.io` - Kubernetes 클러스터 외부에 있는 지표를 제공합니다(예: SQS 대기열 깊이, ELB 지연 시간).

이러한 세 가지 APIs 중 하나를 사용하여 애플리케이션 규모 조정을 위한 지표를 제공해야 합니다.

사용자 지정 또는 외부 지표를 기반으로 애플리케이션 규모 조정

사용자 지정 또는 외부 지표를 사용하여 CPU 또는 메모리 사용률 이외의 지표에서 애플리케이션을 확장할 수 있습니다. 사용자 지정 지표 API 서버는 HPA가 애플리케이션을 자동 조정하는 데 사용할 수 있는 `custom-metrics.k8s.io` API를 제공합니다.

Kubernetes 지표 APIs용 Prometheus 어댑터를 사용하여 Prometheus에서 지표를 수집하고 HPA와 함께 사용할 수 있습니다. 이 경우 Prometheus 어댑터는 Prometheus 지표를 지표 API 형식으로 표시합니다.

Prometheus 어댑터를 배포한 후에는 kubectl을 사용하여 사용자 지정 지표를 쿼리할 수 있습니다. `kubectl get –raw /apis/custom.metrics.k8s.io/v1beta1/`

외부 지표는 이름에서 알 수 있듯이 Horizontal Pod Autoscaler에 Kubernetes 클러스터 외부의 지표를 사용하여 배포를 확장할 수 있는 기능을 제공합니다. 예를 들어 배치 처리 워크로드에서는 SQS 대기열에서 진행 중인 작업 수에 따라 복제본 수를 조정하는 것이 일반적입니다.

Kubernetes 워크로드를 자동 조정하려면 여러 사용자 지정 이벤트를 기반으로 컨테이너 조정을 구동할 수 있는 오픈 소스 프로젝트인 KEDA(Kubernetes 이벤트 기반 Autoscaling)를 사용할 수 있습니다. 이 AWS 블로그에서는 Kubernetes용 Amazon Managed Service for Prometheus 워크로드 Auto Scaling을 사용하는 방법을 간략하게 설명합니다.

Vertical Pod Autoscaler(VPA)

VPA는 포드의 CPU 및 메모리 예약을 자동으로 조정하여 애플리케이션을 "적합하게" 조정할 수 있도록 지원합니다. 리소스 할당을 늘려 수직으로 확장해야 하는 애플리케이션의 경우 VPA를 사용하여 포드 복제본을 자동으로 조정하거나 조정 권장 사항을 제공할 수 있습니다.

VPA의 현재 구현이 포드에 대한 현재 위치 조정을 수행하지 않으므로 VPA가 확장해야 하는 경우 애플리케이션을 일시적으로 사용할 수 없게 됩니다. 대신 확장해야 하는 포드를 다시 생성합니다.

EKS 설명서에는 VPA 설정에 대한 연습이 포함되어 있습니다.

Fairwinds Goldilocks 프로젝트는 CPU 및 메모리 요청과 제한에 대한 VPA 권장 사항을 시각화하는 대시보드를 제공합니다. VPA 업데이트 모드를 사용하면 VPA 권장 사항에 따라 포드를 자동으로 조정할 수 있습니다.

애플리케이션 업데이트

최신 애플리케이션에는 높은 수준의 안정성과 가용성으로 신속한 혁신이 필요합니다. Kubernetes는 고객을 방해하지 않고 애플리케이션을 지속적으로 업데이트할 수 있는 도구를 제공합니다.

가용성을 저하시키지 않고 변경 사항을 신속하게 배포할 수 있는 몇 가지 모범 사례를 살펴보겠습니다.

롤백을 수행하는 메커니즘 보유

실행 취소 버튼이 있으면 재해를 피할 수 있습니다. 프로덕션 클러스터를 업데이트하기 전에 별도의 하위 환경(테스트 또는 개발 환경)에서 배포를 테스트하는 것이 좋습니다. CI/CD 파이프라인을 사용하면 배포를 자동화하고 테스트하는 데 도움이 될 수 있습니다. 지속적 배포 파이프라인을 사용하면 업그레이드에 결함이 있는 경우 이전 버전으로 빠르게 되돌릴 수 있습니다.

배포를 사용하여 실행 중인 애플리케이션을 업데이트할 수 있습니다. 이는 일반적으로 컨테이너 이미지를 업데이트하여 수행됩니다. `kubectl`를 사용하여 다음과 같이 배포를 업데이트할 수 있습니다.

```
kubectl --record deployment.apps/nginx-deployment set image nginx-deployment
 nginx=nginx:1.16.1
```

`--record` 인수는 배포에 대한 변경 사항을 기록하고 롤백을 수행해야 하는 경우 도움이 됩니다. 는 클러스터의 배포에 대해 기록된 변경 사항을 `kubectl rollout history deployment` 보여줍니다. 를 사용하여 변경 사항을 롤백할 수 있습니다`kubectl rollout undo deployment <DEPLOYMENT_NAME>`.

기본적으로 포드를 다시 생성해야 하는 배포를 업데이트하면 배포가 롤링 업데이트를 수행합니다. 즉, Kubernetes는 배포에서 실행 중인 포드의 일부만 업데이트하고 모든 포드를 한 번에 업데이트하지는 않습니다. Kubernetes가 `RollingUpdateStrategy` 속성을 통해 롤링 업데이트를 수행하는 방법을 제어할 수 있습니다.

배포의 롤링 업데이트를 수행할 때 `Max Unavailable` 속성을 사용하여 업데이트 중에 사용할 수 없는 최대 포드 수를 지정할 수 있습니다. 배포의 `Max Surge` 속성을 사용하면 원하는 포드 수를 초과하여 생성할 수 있는 최대 포드 수를 설정할 수 있습니다.

롤아웃이 고객을 방해하지 `max unavailable` 않도록 조정하는 것이 좋습니다. 예를 들어 Kubernetes는 `max unavailable` 기본적으로 25%를 설정합니다. 즉, 포드가 100개인 경우 롤아웃 중에 활발하게 작동하는 포드가 75개뿐일 수 있습니다. 애플리케이션에 최소 80개의 포드가 필요한 경우이 롤아웃은 중단될 수 있습니다. 대신를 20%`max unavailable`로 설정하여 롤아웃 전체에 최소 80개의 기능 포드가 있는지 확인할 수 있습니다.

블루/그린 배포 사용

변경 사항은 본질적으로 위험하지만 실행 취소할 수 없는 변경 사항은 잠재적으로 치명적일 수 있습니다. 롤백을 통해 시간을 효과적으로 되돌릴 수 있는 변경 절차를 통해 개선 및 실험을 더 안전하게 수행할 수 있습니다. 블루/그린 배포를 사용하면 문제가 발생할 경우 변경 사항을 신속하게 취소할 수 있습니다. 이 배포 전략에서는 새 버전에 대한 환경을 생성합니다. 이 환경은 업데이트 중인 애플리케이션의 현재 버전과 동일합니다. 새 환경이 프로비저닝되면 트래픽이 새 환경으로 라우팅됩니다. 새 버전이 오류를 생성하지 않고 원하는 결과를 생성하면 이전 환경이 종료됩니다. 그렇지 않으면 트래픽이 이전 버전으로 복원됩니다.

기존 버전의 배포와 동일한 새 배포를 생성하여 Kubernetes에서 블루/그린 배포를 수행할 수 있습니다. 새 배포의 포드가 오류 없이 실행되고 있는지 확인한 후 애플리케이션의 포드로 트래픽을 라우팅하는 서비스의 `selector` 사양을 변경하여 새 배포로 트래픽 전송을 시작할 수 있습니다.

Flux, Jenkins, Chef와 같은 많은 지속적 통합 도구를 사용하면 블루/그린 배포를 자동화할 수 있습니다. AWS Containers 블로그에는 AWS Load Balancer Controller를 사용한 연습, 블루/그린 배포에 AWS Load Balancer Controller 사용, canary 배포 및 A/B 테스트가 포함되어 있습니다.

Canary 배포 사용

카나리아 배포는 변경으로 인한 위험을 크게 제거할 수 있는 블루/그린 배포의 변형입니다. 이 배포 전략에서는 이전 배포와 함께 포드 수가 적은 새 배포를 생성하고 소량의 트래픽을 새 배포로 전환합니다. 지표에서 새 버전이 기존 버전과 비슷하거나 더 나은 성능을 보이고 있는 경우 모든 트래픽이 새 배포로 전환될 때까지 확장하면서 새 배포로 가는 트래픽을 점진적으로 증가시킵니다. 문제가 있는 경우 모든 트래픽을 이전 배포로 라우팅하고 새 배포로 트래픽 전송을 중지할 수 있습니다.

Kubernetes는 카나리아 배포를 수행하는 기본 방법을 제공하지 않지만 Istio와 함께 Flagger와 같은 도구를 사용할 수 있습니다.

상태 확인 및 자가 복구

버그가 없는 소프트웨어는 없지만 Kubernetes는 소프트웨어 장애의 영향을 최소화하는 데 도움이 될 수 있습니다. 과거에는 애플리케이션이 충돌한 경우 애플리케이션을 수동으로 다시 시작하여 상황을 해결해야 했습니다. Kubernetes를 사용하면 포드에서 소프트웨어 장애를 감지하고 자동으로 새 복제본으로 교체할 수 있습니다. Kubernetes를 사용하면 애플리케이션의 상태를 모니터링하고 비정상 인스턴스를 자동으로 교체할 수 있습니다.

Kubernetes는 세 가지 유형의 상태 확인을 지원합니다.

1. Liveness 프로브

2. 시작 프로브(Kubernetes 버전 1.16 이상에서 지원됨)
3. 준비 프로브

Kubernetes 에이전트인 Kubelet은 위에서 언급한 모든 검사를 실행할 책임이 있습니다. Kubelet은 다음과 같은 세 가지 방법으로 포드의 상태를 확인할 수 있습니다. kubelet은 포드의 컨테이너 내에서 셸 명령을 실행하거나, HTTP GET 요청을 컨테이너로 보내거나, 지정된 포트에서 TCP 소켓을 열 수 있습니다.

컨테이너 내에서 셸 스크립트를 실행하는 `exec`기반 프로브를 선택하는 경우 `timeoutSeconds` 값이 만료되기 전에 셸 명령이 종료되는지 확인합니다. 그렇지 않으면 노드에 `<defunct>` 프로세스가 있어 노드 오류가 발생합니다.

권장 사항

Liveness 프로브를 사용하여 비정상 포드 제거

Liveness 프로브는 프로세스가 계속 실행되지만 애플리케이션이 응답하지 않는 교착 상태를 감지할 수 있습니다. 예를 들어 포트 80에서 수신 대기하는 웹 서비스를 실행하는 경우 포드의 포트 80에서 HTTP GET 요청을 보내도록 Liveness 프로브를 구성할 수 있습니다. Kubelet은 주기적으로 포드에 GET 요청을 보내고 응답을 예상합니다. 포드가 200~399 사이에 응답하면 kubelet은 포드가 정상이라고 간주합니다. 그렇지 않으면 포드가 비정상으로 표시됩니다. 포드가 상태 확인에 지속적으로 실패하면 kubelet이 해당 포드를 종료합니다.

`initialDelaySeconds`를 사용하여 첫 번째 프로브를 지연시킬 수 있습니다.

Liveness 프로브를 사용할 때는 Kubernetes가 모든 포드를 교체하려고 하므로 애플리케이션이 모든 포드가 동시에 Liveness 프로브에 실패하는 상황이 되지 않도록 해야 합니다. 그러면 애플리케이션이 오프라인으로 렌더링됩니다. 또한 Kubernetes는 Liveness 프로브에도 실패하여 컨트롤 플레인에 불필요한 부담을 주는 새 포드를 계속 생성합니다. 외부 데이터베이스와 같이 포드 외부에 있는 요인에 의존하도록 Liveness 프로브를 구성하지 마세요. 즉, 응답하지 않는 external-to-your-Pod 데이터베이스로 인해 포드가 Liveness 프로브에 실패해서는 안 됩니다.

Sandor Szücs의 게시물 LIVENESS PROBES IS DANGEROUS는 잘못 구성된 프로브로 인해 발생할 수 있는 문제를 설명합니다.

시작 시간이 더 오래 걸리는 애플리케이션에 Startup 프로브 사용

앱을 시작하는 데 추가 시간이 필요한 경우 Startup 프로브를 사용하여 Liveness 및 Readiness 프로브를 지연시킬 수 있습니다. 예를 들어 데이터베이스에서 캐시를 수화해야 하는 Java 앱이 완전히 작동하

려면 최대 2분이 걸릴 수 있습니다. 완전히 작동할 때까지 모든 Liveness 또는 Readiness 프로브가 실패할 수 있습니다. Startup Probe를 구성하면 Liveness 또는 Readiness Probe가 실행되기 전에 Java 앱이 정상 상태가 될 수 있습니다.

시작 프로브가 성공할 때까지 다른 모든 프로브는 비활성화됩니다. Kubernetes가 애플리케이션 시작을 기다려야 하는 최대 시간을 정의할 수 있습니다. 최대 구성 시간 후에도 포드가 여전히 시작 프로브에 실패하면 포드가 종료되고 새 포드가 생성됩니다.

Startup 프로브는 Liveness 프로브와 유사합니다. 프로브가 실패하면 포드가 다시 생성됩니다. Ricardo A.가 게시물 Fantastic Probes and How To Configure Them에서 설명하는 것처럼 애플리케이션의 시작 시간을 예측할 수 없는 경우 Startup Probes를 사용해야 합니다. 애플리케이션을 시작하는 데 10초가 필요하다는 것을 알고 있는 경우 Liveness/Readiness 프로브를 `initialDelaySeconds` 대신와 함께 사용해야 합니다.

준비 프로브를 사용하여 부분 가용성 감지

Liveness 프로브가 포드를 종료(따라서 앱을 다시 시작)하여 해결된 앱에서 장애를 감지하는 동안 준비 프로브는 앱을 일시적으로 사용할 수 없는 상태를 감지합니다. 이러한 상황에서는 앱이 일시적으로 응답하지 않을 수 있지만이 작업이 완료되면 다시 정상 상태가 될 것으로 예상됩니다.

예를 들어 디스크 I/O 작업이 급증하는 동안 애플리케이션은 일시적으로 요청을 처리하지 못할 수 있습니다. 여기서 애플리케이션의 포드를 종료하는 것은 해결 방법이 아닙니다. 동시에 포드로 전송된 추가 요청이 실패할 수 있습니다.

준비 프로브를 사용하여 앱에서 일시적인 가용성을 감지하고 다시 작동할 때까지 포드에 대한 요청 전송을 중지할 수 있습니다. 실패로 인해 포드가 다시 생성되는 Liveness 프로브와 달리, 실패한 준비 프로브는 포드가 Kubernetes Service에서 트래픽을 수신하지 않음을 의미합니다. 준비 프로브가 성공하면 포드는 서비스에서 트래픽 수신을 재개합니다.

Liveness 프로브와 마찬가지로 포드 외부의 리소스(예: 데이터베이스)에 의존하는 준비 프로브를 구성하지 마세요. 다음은 제대로 구성되지 않은 준비 상태로 인해 애플리케이션이 작동하지 않을 수 있는 시나리오입니다. 앱의 데이터베이스에 연결할 수 없을 때 포드의 준비 프로브가 실패하면 다른 포드 복제본도 동일한 상태 확인 기준을 공유하므로 동시에 실패합니다. 이러한 방식으로 프로브를 설정하면 데이터베이스를 사용할 수 없을 때마다 포드의 준비 프로브가 실패하고 Kubernetes가 모든 포드의 트래픽 전송을 중지합니다.

준비 프로브를 사용하면 배포를 업데이트하는 데 걸리는 시간을 늘릴 수 있다는 부작용이 있습니다. 준비 프로브가 성공하지 않으면 새 복제본은 트래픽을 수신하지 않습니다. 그때까지 이전 복제본은 트래픽을 계속 수신합니다.

중단 처리

포드 수명은 유한합니다. 장기 실행 포드가 있더라도 시간이 되면 포드가 올바르게 종료되도록 하는 것이 좋습니다. 업그레이드 전략에 따라 Kubernetes 클러스터 업그레이드에서는 새 작업자 노드를 생성해야 할 수 있으며, 이를 위해서는 최신 노드에서 모든 포드를 다시 생성해야 합니다. 적절한 종료 처리 및 포드 중단 예산은 포드가 이전 노드에서 제거되고 최신 노드에서 다시 생성되므로 서비스 중단을 방지하는 데 도움이 될 수 있습니다.

작업자 노드를 업그레이드하는 가장 좋은 방법은 새 작업자 노드를 생성하고 이전 노드를 종료하는 것입니다. 작업자 노드를 종료하기 전에 drain 노드를 종료해야 합니다. 작업자 노드가 드레이닝되면 모든 포드가 안전하게 제거됩니다. 여기서는 안전이 키워드입니다. 작업자의 포드가 제거되면 단순히 SIGKILL 신호가 전송되지 않습니다. 대신 제거 중인 포드에 있는 각 컨테이너의 기본 프로세스(PID 1)로 SIGTERM 신호가 전송됩니다. SIGTERM 신호가 전송된 후 Kubernetes는 SIGKILL 신호가 전송되기 전에 어느 정도 시간(유예 기간)을 프로세스에 부여합니다. 이 유예 기간은 기본적으로 30초입니다. kubectl에서 grace-period 플래그를 사용하거나 Podspec에서 선언하여 기본값을 재정의할 수 terminationGracePeriodSeconds 있습니다.

kubectl delete pod <pod name> –grace-period=<seconds>

기본 프로세스에 PID 1이 없는 컨테이너가 있는 것이 일반적입니다. 이 Python 기반 샘플 컨테이너를 고려하세요.

```
$ kubectl exec python-app -it ps
 PID USER TIME COMMAND
 1   root 0:00 {script.sh} /bin/sh ./script.sh
 5   root 0:00 python app.py
```

이 예제에서 쉘 스크립트는 SIGTERM를 수신합니다.이 예제에서 Python 애플리케이션으로 발생하는 기본 프로세스는 SIGTERM 신호를 받지 않습니다. 포드가 종료되면 Python 애플리케이션이 갑자기 종료됩니다. 이는 Python 애플리케이션을 시작하도록 컨테이너ENTRYPOINT의를 변경하여 해결할 수 있습니다. 또는 dumb-init과 같은 도구를 사용하여 애플리케이션이 신호를 처리할 수 있도록 할 수 있습니다.

컨테이너 후크를 사용하여 컨테이너 시작 또는 중지 시 스크립트 또는 HTTP 요청을 실행할 수도 있습니다. PreStop 후크 작업은 컨테이너가 SIGTERM 신호를 수신하기 전에 실행되며이 신호가 전송되기 전에 완료되어야 합니다. terminationGracePeriodSeconds 값은 SIGTERM 신호가 전송될 때가 아니라 PreStop 후크 작업이 실행되기 시작할 때부터 적용됩니다.

권장 사항

포드 중단 예산으로 중요한 워크로드 보호

애플리케이션의 복제본 수가 선언된 임계값 아래로 떨어지면 포드 중단 예산 또는 PDB가 제거 프로세스를 일시적으로 중지할 수 있습니다. 사용 가능한 복제본 수가 임계값을 초과하면 제거 프로세스가 계속됩니다. PDB를 사용하여 `minAvailable` 및 복제본 `maxUnavailable` 수를 선언할 수 있습니다. 예를 들어 앱 사본을 3개 이상 사용하려면 PDB를 생성할 수 있습니다.

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: my-svc-pdb
spec:
  minAvailable: 3
  selector:
    matchLabels:
      app: my-svc
```

위의 PDB 정책은 3개 이상의 복제본을 사용할 수 있을 때까지 Kubernetes에 제거 프로세스를 중지하도록 지시합니다. 노드 드레이닝은를 준수합니다`PodDisruptionBudgets`. EKS 관리형 노드 그룹 업그레이드 중에 노드는 15분 제한 시간으로 드레이닝됩니다. 15분 후 업데이트가 강제되지 않으면(이 옵션을 EKS 콘솔에서 롤링 업데이트라고 함) 업데이트가 실패합니다. 강제로 업데이트하면 포드가 삭제됩니다.

자체 관리형 노드의 경우 AWS 노드 종료 핸들러와 같은 도구를 사용하여 EC2 유지 관리 이벤트 및 EC2 스팟 중단과 같이 EC2 인스턴스를 사용할 수 없게 만들 수 있는 이벤트에 Kubernetes 컨트롤 플레인이 적절하게 응답하도록 할 수도 있습니다. EC2 Kubernetes API를 사용하여 노드를 코딩하여 새 포드가 예약되지 않았는지 확인한 다음 드레이닝하여 실행 중인 포드를 종료합니다.

포드 불리성을 사용하여 여러 노드에서 배포의 포드를 예약하고 노드 업그레이드 중에 PDB 관련 지연을 방지할 수 있습니다.

카오스 엔지니어링 실습

카오스 엔지니어링은 분산 시스템에서 실험하여 프로덕션 환경에서 격렬한 조건을 견딜 수 있는 시스템의 기능에 대한 신뢰를 구축하는 분야입니다.

그의 블로그에서 Dominik Tornow는 Kubernetes가 " 사용자가 시스템의 원하는 상태를 시스템에 표현하는 선언적 시스템이라고 설명합니다. 그런 다음 시스템은 현재 상태와 원하는 상태를 고려하여 현재 상태에서 원하는 상태로 전환할 명령 시퀀스를 결정합니다. " 즉, Kubernetes는 항상 원하는 상태를

저장하며 시스템이 이탈하면 Kubernetes는 상태를 복원하기 위한 조치를 취합니다. 예를 들어 작업자 노드를 사용할 수 없게 되면 Kubernetes는 포드를 다른 작업자 노드로 다시 예약합니다. 마찬가지로 `replica` 충돌이 발생하면 배포 콘솔러가 새를 생성합니다`replica`. 이러한 방식으로 Kubernetes 컨트롤러는 실패를 자동으로 수정합니다.

Gremlin과 같은 카오스 엔지니어링 도구는 Kubernetes 클러스터의 복원력을 테스트하고 단일 장애 지점을 식별하는 데 도움이 됩니다. 클러스터(및 그 이상)에 인공 카오스를 도입하는 도구는 체계적인 약점을 발견하고, 병목 현상과 잘못된 구성을 식별하고, 제어된 환경에서 문제를 해결할 수 있는 기회를 제공합니다. 카오스 엔지니어링 철학은 의도적인 업무 중단과 스트레스 테스트 인프라를 지지하여 예상치 못한 가동 중지 시간을 최소화합니다.

Service Mesh 사용

서비스 메시를 사용하여 애플리케이션의 복원력을 개선할 수 있습니다. 서비스 메시는 service-to-service 통신을 활성화하고 마이크로서비스 네트워크의 관찰성을 높입니다. 대부분의 서비스 메시 제품은 애플리케이션의 네트워크 트래픽을 가로채고 검사하는 각 서비스와 함께 작은 네트워크 프록시를 실행하는 방식으로 작동합니다. 애플리케이션을 수정하지 않고도 애플리케이션을 메시에 배치할 수 있습니다. 서비스 프록시의 내장 기능을 사용하면 네트워크 통계를 생성하고, 액세스 로그를 생성하고, 분산 추적을 위한 아웃바운드 요청에 HTTP 헤더를 추가하도록 할 수 있습니다.

서비스 메시를 사용하면 자동 요청 재시도, 제한 시간, 회로 차단 및 속도 제한과 같은 기능을 통해 마이크로서비스의 복원력을 높일 수 있습니다.

여러 클러스터를 운영하는 경우 서비스 메시를 사용하여 클러스터 간 service-to-service 통신을 활성화할 수 있습니다.

서비스 메시

- 이스티오
- LinkerD
- Consul

관찰성

관찰성은 모니터링, 로깅 및 추적을 포함하는 포괄적인 용어입니다. 마이크로서비스 기반 애플리케이션은 기본적으로 배포됩니다. 단일 시스템 모니터링으로 충분한 모놀리식 애플리케이션과 달리 분산 애플리케이션 아키텍처에서는 각 구성 요소의 성능을 모니터링해야 합니다. 클러스터 수준 모니터링, 로깅 및 분산 추적 시스템을 사용하여 고객을 방해하기 전에 클러스터의 문제를 식별할 수 있습니다.

문제 해결 및 모니터링을 위한 Kubernetes 기본 제공 도구는 제한적입니다. 지표 서버는 리소스 지표를 수집하여 메모리에 저장하지만 유지하지는 않습니다. kubectl을 사용하여 포드의 로그를 볼 수 있지만 Kubernetes는 로그를 자동으로 보존하지 않습니다. 또한 분산 추적 구현은 애플리케이션 코드 수준에서 또는 서비스 메시를 사용하여 수행됩니다.

여기에서 Kubernetes의 확장성이 빛납니다. Kubernetes를 사용하면 선호하는 중앙 집중식 모니터링, 로깅 및 추적 솔루션을 가져올 수 있습니다.

권장 사항

애플리케이션 모니터링

최신 애플리케이션에서 모니터링해야 하는 지표 수는 지속적으로 증가하고 있습니다. 고객의 문제를 해결하는 데 집중할 수 있도록 애플리케이션을 자동으로 추적할 수 있는 방법이 있다면 도움이 됩니다. Prometheus 또는 CloudWatch Container Insights와 같은 클러스터 전체 모니터링 도구는 클러스터와 워크로드를 모니터링하고 문제가 발생하기 전에 신호를 제공할 수 있습니다.

모니터링 도구를 사용하면 운영 팀이 구독할 수 있는 알림을 생성할 수 있습니다. 악화될 경우 중단으로 이어지거나 애플리케이션 성능에 영향을 미칠 수 있는 이벤트에 대한 경보를 활성화하는 규칙을 고려합니다.

모니터링해야 하는 지표가 확실하지 않은 경우 다음 방법에서 영감을 얻을 수 있습니다.

- 빨간색 메서드입니다. 요청, 오류 및 기간을 나타냅니다.
- 사용 방법. 사용률, 포화도 및 오류를 나타냅니다.

Sysdig의 게시물 Kubernetes에 대한 알림 모범 사례에는 애플리케이션 가용성에 영향을 미칠 수 있는 포괄적인 구성 요소 목록이 포함되어 있습니다.

Prometheus 클라이언트 라이브러리를 사용하여 애플리케이션 지표 공개

애플리케이션 상태를 모니터링하고 표준 지표를 집계하는 것 외에도 Prometheus 클라이언트 라이브러리를 사용하여 애플리케이션별 사용자 지정 지표를 노출하여 애플리케이션의 관찰성을 개선할 수도 있습니다.

중앙 집중식 로깅 도구를 사용하여 로그 수집 및 유지

EKS에서의 로깅은 컨트롤 플레인 로그와 애플리케이션 로그의 두 범주로 나뉩니다. EKS 컨트롤 플레인 로깅은 컨트롤 플레인에서 계정의 CloudWatch Logs로 직접 감사 및 진단 로그를 제공합니다. 애플리케이션 로그는 클러스터 내에서 실행되는 포드에서 생성되는 로그입니다. 애플리케이션 로그에는

비즈니스 로직 애플리케이션과 CoreDNS, Cluster Autoscaler, Prometheus 등과 같은 Kubernetes 시스템 구성 요소를 실행하는 포드에서 생성된 로그가 포함됩니다.

EKS는 다섯 가지 유형의 컨트롤 플레인 로그를 제공합니다.

1. Kubernetes API 서버 구성 요소 로그
2. 감사
3. 인증자
4. 컨트롤러 관리자
5. 스케줄러

컨트롤러 관리자 및 스케줄러 로그는 병목 현상 및 오류와 같은 컨트롤 플레인 문제를 진단하는 데 도움이 될 수 있습니다. 기본적으로 EKS 컨트롤 플레인 로그는 CloudWatch Logs로 전송되지 않습니다. 컨트롤 플레인 로깅을 활성화하고 계정의 각 클러스터에 대해 캡처하려는 EKS 컨트롤 플레인 로그 유형을 선택할 수 있습니다.

애플리케이션 로그를 수집하려면 클러스터에 Fluent Bit, Fluentd 또는 CloudWatch Container Insights와 같은 로그 집계 도구를 설치해야 합니다.

Kubernetes 로그 집계 도구는 DaemonSets로 실행되고 노드에서 컨테이너 로그를 스크레이프합니다. 그런 다음 애플리케이션 로그가 중앙 집중식 스토리지 대상으로 전송됩니다. 예를 들어 CloudWatch Container Insights는 Fluent Bit 또는 Fluentd를 사용하여 로그를 수집하고 CloudWatch Logs로 전송하여 저장할 수 있습니다. Fluent Bit 및 Fluentd는 Elasticsearch 및 InfluxDB와 같은 널리 사용되는 많은 로그 분석 시스템을 지원하므로 Fluent 비트 또는 Fluentd의 로그 구성을 수정하여 로그의 스토리지 백엔드를 변경할 수 있습니다.

분산 추적 시스템을 사용하여 병목 현상 식별

일반적인 최신 애플리케이션에는 네트워크를 통해 분산된 구성 요소가 있으며, 그 신뢰성은 애플리케이션을 구성하는 각 구성 요소의 적절한 기능에 따라 달라집니다. 분산 추적 솔루션을 사용하여 요청 흐름 방식과 시스템 통신 방식을 이해할 수 있습니다. 트레이스는 애플리케이션 네트워크에 병목 현상이 있는 위치를 표시하고 계단식 실패를 일으킬 수 있는 문제를 방지할 수 있습니다.

애플리케이션에서 추적을 구현하는 두 가지 옵션이 있습니다. 공유 라이브러리를 사용하여 코드 수준에서 분산 추적을 구현하거나 서비스 메시를 사용할 수 있습니다.

코드 수준에서 추적을 구현하는 것은 불리할 수 있습니다. 이 방법에서는 코드를 변경해야 합니다. 이는 폴리글롯 애플리케이션이 있는 경우 더욱 복잡합니다. 또한 서비스 전체에서 또 다른 라이브러리를 유지 관리할 책임이 있습니다.

LinkerD 및 Istio와 같은 Service Mesh를 사용하여 애플리케이션 코드를 최소한으로 변경하여 애플리케이션에서 분산 추적을 구현할 수 있습니다. 서비스 메시를 사용하여 지표 생성, 로깅 및 추적을 표준화할 수 있습니다.

AWS X-Ray와 같은 추적 도구인 Jaeger는 공유 라이브러리 및 서비스 메시 구현을 모두 지원합니다.

나중에 서비스 메시를 채택하는 경우 도구를 전환할 필요가 없도록 (공유 라이브러리 및 서비스 메시) 구현을 모두 지원하는 AWS X-Ray 또는 Jaeger와 같은 추적 도구를 사용하는 것이 좋습니다.

EKS 제어 플레인

Amazon Elastic Kubernetes Service(EKS)는 자체 Kubernetes 컨트롤 플레인 또는 작업자 노드를 설치, 운영 및 유지 관리할 필요 없이 AWS에서 Kubernetes를 쉽게 실행할 수 있는 관리형 Kubernetes 서비스입니다. 업스트림 Kubernetes를 실행하며 Kubernetes 준수 인증을 받았습니다. 이러한 적합성을 통해 EKS는 EC2 또는 온프레미스에 설치할 수 있는 오픈 소스 커뮤니티 버전과 마찬가지로 Kubernetes APIs를 지원합니다. 업스트림 Kubernetes에서 실행되는 기존 애플리케이션은 Amazon EKS와 호환됩니다.

EKS는 Kubernetes 컨트롤 플레인 노드의 가용성과 확장성을 자동으로 관리하고 비정상 컨트롤 플레인 노드를 자동으로 교체합니다.

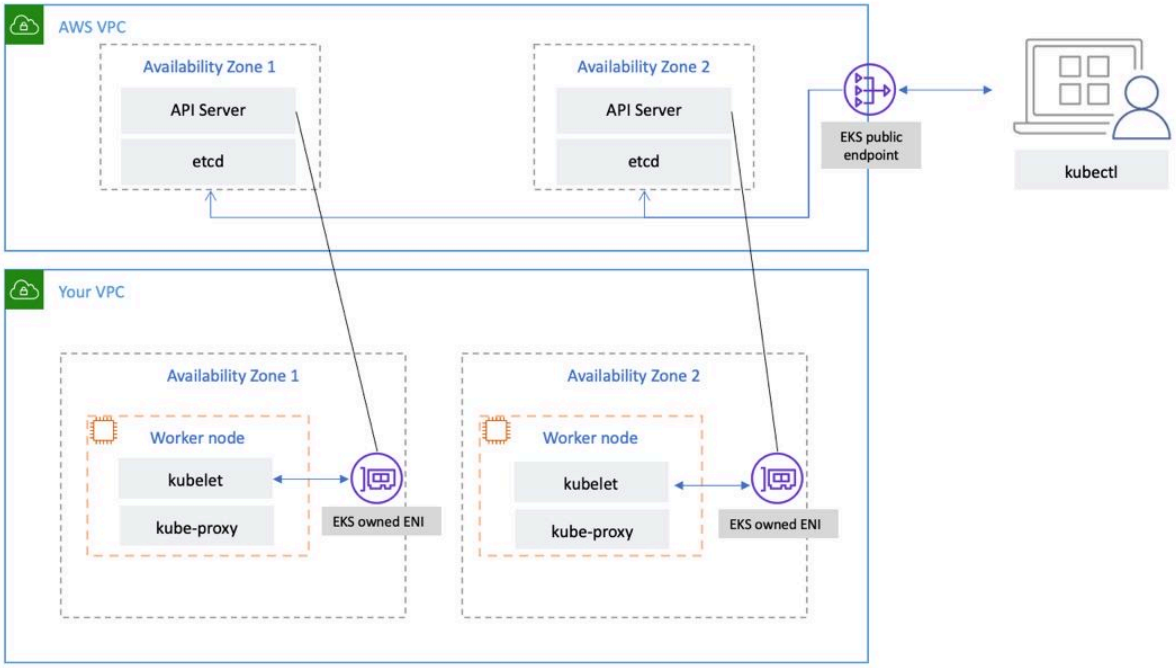
EKS 아키텍처

EKS 아키텍처는 Kubernetes 컨트롤 플레인의 가용성과 내구성을 손상시킬 수 있는 단일 장애 지점을 제거하도록 설계되었습니다.

EKS에서 관리하는 Kubernetes 컨트롤 플레인은 EKS 관리형 VPC 내에서 실행됩니다. EKS 컨트롤 플레인은 Kubernetes API 서버 노드 등 클러스터로 구성됩니다. API 서버, 스케줄러와 같은 구성 요소를 실행하고 Auto Scaling 그룹에서 `kube-controller-manager` 실행되는 Kubernetes API 서버 노드입니다. EKS는 AWS 리전 내 개별 가용 영역(AZs)에서 최소 2개의 API 서버 노드를 실행합니다. 마찬가지로 내구성을 위해 etcd 서버 노드는 세 개의 AZs. EKS는 각 AZ에서 NAT 게이트웨이를 실행하고 API 서버 및 etcd 서버는 프라이빗 서브넷에서 실행됩니다. 이 아키텍처는 단일 AZ의 이벤트가 EKS 클러스터의 가용성에 영향을 미치지 않도록 합니다.

새 클러스터를 생성하면 Amazon EKS는 클러스터와 통신하는 데 사용하는 관리형 Kubernetes API 서버에 대해 가용성이 높은 엔드포인트를 생성합니다(와 같은 도구 사용`kubectl`). 관리형 엔드포인트는 NLB를 사용하여 Kubernetes API 서버를 로드 밸런싱합니다. 또한 EKS는 작업자 노드와의 통신을 용이하게 하기 위해 서로 다른 AZs에 두 개의 ENI를 프로비저닝합니다.

EKS 데이터 영역 네트워크 연결



퍼블릭 인터넷(퍼블릭 엔드포인트 사용) 또는 VPC(EKS 관리형 ENIs 사용) 또는 둘 다에서 Kubernetes 클러스터의 API 서버에 연결할 수 있는지 여부를 구성할 수 있습니다.

사용자 및 작업자 노드가 퍼블릭 엔드포인트를 사용하여 API 서버에 연결하든 EKS 관리형 ENI를 사용하여 API 서버에 연결하든, 연결을 위한 중복 경로가 있습니다.

권장 사항

다음 권장 사항을 검토합니다.

컨트롤 플레인 지표 모니터링

Kubernetes API 지표를 모니터링하면 컨트롤 플레인 성능에 대한 인사이트를 얻고 문제를 식별할 수 있습니다. 비정상 컨트롤 플레인은 클러스터 내에서 실행되는 워크로드의 가용성을 손상시킬 수 있습니다. 예를 들어, 잘못 작성된 컨트롤러는 API 서버를 오버로드하여 애플리케이션의 가용성에 영향을 미칠 수 있습니다.

Kubernetes는 `/metrics` 엔드포인트에서 컨트롤 플레인 지표를 노출합니다.

를 사용하여 노출된 지표를 볼 수 있습니다`kubectl`.

```
kubectl get --raw /metrics
```

이러한 지표는 Prometheus 텍스트 형식으로 표시됩니다.

Prometheus를 사용하여 이러한 지표를 수집하고 저장할 수 있습니다. 2020년 5월에 CloudWatch는 CloudWatch Container Insights에서 Prometheus 지표 모니터링에 대한 지원을 추가했습니다. 따라서 Amazon CloudWatch를 사용하여 EKS 컨트롤 플레인을 모니터링할 수도 있습니다. 자습서를 사용하여 새 Prometheus 스크레이프 대상 추가: Prometheus KPI 서버 지표를 사용하여 지표를 수집하고 CloudWatch 대시보드를 생성하여 클러스터의 컨트롤 플레인을 모니터링할 수 있습니다. CloudWatch

여기에서 Kubernetes API 서버 지표를 찾을 수 있습니다. 예를 들어는 API 요청을 실행하는 데 걸리는 시간을 나타낼 `apiserver_request_duration_seconds` 수 있습니다.

다음 컨트롤 플레인 지표를 모니터링하는 것이 좋습니다.

API 서버

지표	설명
`apiserver_request_total`	각 동사, 드라이 런 값, 그룹, 버전, 리소스, 범위, 구성 요소 및 HTTP 응답 코드에 대해 분류된 apiserver 요청의 카운터.
`apiserver_request_duration_seconds*`	각 동사, 모의 실행 값, 그룹, 버전, 리소스, 하위 리소스, 범위 및 구성 요소에 대한 초 단위의 응답 지연 시간 히스토그램입니다.
`apiserver_admission_controller_admission_duration_seconds*`	이름별로 식별되고 각 작업과 API 리소스 및 유형(검증 또는 승인)에 대해 구분된 초 단위의 허용 컨트롤러 지연 시간 히스토그램입니다.
`apiserver_admission_webhook_rejection_count`	승인 웹후크 거부 수입니다. 이름, 작업, rejection_code, 유형(검증 또는 승인), error_type(calling_webhook_error, apiserver_internal_error, no_error)으로 식별됨
`rest_client_request_duration_seconds*`	지연 시간 히스토그램을 초 단위로 요청합니다. 동사 및 URL로 분류됩니다.

지표	설명
rest_client_requests_total	상태 코드, 메서드 및 호스트로 분할된 HTTP 요청 수.

- 히스토그램 지표에는 _bucket, _sum 및 _count 접미사가 포함됩니다.

etcd

지표	설명
etcd_request_duration_seconds*	각 작업 및 객체 유형에 대한 Etcd 요청 지연 시간 히스토그램.

지표	설명
apiserver_storage_db_total_size_in_bytes 또는 apiserver_storage_size_bytes (EKS v1.28로 시작)	Etcd 데이터베이스 크기입니다.

- 히스토그램 지표에는 _bucket, _sum 및 _count 접미사가 포함됩니다.

Kubernetes 모니터링 개요 대시보드를 사용하여 Kubernetes API 서버 요청, 지연 시간 및 기타 지연 시간 지표를 시각화하고 모니터링하는 것이 좋습니다.

> **Important**
>
> 데이터베이스 크기 제한을 초과하면 etcd는 공간 없음 경보를 내보내고 추가 쓰기 요청 수행을 중지합니다. 즉, 클러스터는 읽기 전용이 되며 새 포드 생성, 배포 조정 등과 같은 객체를 변경하기 위한 모든 요청은 클러스터의 API 서버에서 거부됩니다.

클러스터 인증

EKS는 현재 보유자/서비스 계정 토큰과 웹후크 토큰 인증을 사용하는 IAM 인증이라는 두 가지 유형의 인증을 지원합니다. 사용자가 Kubernetes API를 호출하면 Webhook는 요청에 포함된 인증 토큰을 IAM에 전달합니다. 서명된 기본 64 URL인 토큰은 AWS 명령줄 인터페이스(AWS CLI)에서 생성됩니다.

EKS 클러스터를 생성하는 IAM 사용자 또는 역할은 클러스터에 대한 모든 액세스 권한을 자동으로 얻습니다. aws-auth configmap을 편집하여 EKS 클러스터에 대한 액세스를 관리할 수 있습니다.

aws-auth 구성 맵을 잘못 구성하고 클러스터에 대한 액세스 권한을 잃더라도 클러스터 생성자의 사용자 또는 역할을 사용하여 EKS 클러스터에 액세스할 수 있습니다.

드물지만 AWS 리전에서 IAM 서비스를 사용할 수 없는 경우 Kubernetes 서비스 계정의 보유자 토큰을 사용하여 클러스터를 관리할 수도 있습니다.

클러스터에서 모든 작업을 수행할 수 있는 super-admin 계정을 생성합니다.

```
kubectl -n kube-system create serviceaccount super-admin
```

수퍼 관리자 클러스터 관리자 역할을 제공하는 역할 바인딩을 생성합니다.

```
kubectl create clusterrolebinding super-admin-rb --clusterrole=cluster-admin --
serviceaccount=kube-system:super-admin
```

서비스 계정의 보안 암호 가져오기:

```
SECRET_NAME=`kubectl -n kube-system get serviceaccount/super-admin -o
 jsonpath='{.secrets[0].name}'`
```

보안 암호와 연결된 토큰 가져오기:

```
TOKEN=`kubectl -n kube-system get secret $SECRET_NAME -o jsonpath='{.data.token}'|
 base64 --decode`
```

에 서비스 계정 및 토큰 추가kubeconfig:

```
kubectl config set-credentials super-admin --token=$TOKEN
```

수퍼 관리자 계정을 사용하려면에서 현재 컨텍스트kubeconfig를 설정합니다.

```
kubectl config set-context --current --user=super-admin
```

최종는 다음과 같아kubeconfig야 합니다.

```
apiVersion: v1
clusters:
- cluster:
```

```
    certificate-authority-data:<REDACTED>
    server: https://<CLUSTER>.gr7.us-west-2.eks.amazonaws.com
  name: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
contexts:
- context:
    cluster: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
    user: super-admin
  name: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
current-context: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
kind: Config
preferences: {}
users:
#- name: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
#  user:
#    exec:
#      apiVersion: client.authentication.k8s.io/v1beta1
#      args:
#      - --region
#      - us-west-2
#      - eks
#      - get-token
#      - --cluster-name
#      - <<cluster name>>
#      command: aws
#      env: null
- name: super-admin
  user:
    token: <<super-admin sa's secret>>
```

Admission Webhook

Kubernetes에는 승인 웹후크 검증과 승인 웹후크 변경이라는 두 가지 유형의 승인 웹후크가 있습니다. 이를 통해 사용자는 kubernetes API를 확장하고 API에서 객체를 수락하기 전에 객체를 검증하거나 변경할 수 있습니다. 이러한 웹후크의 구성이 좋지 않으면 클러스터 중요 작업을 차단하여 EKS 컨트롤 플레인이 불안정해질 수 있습니다.

클러스터 중요 작업에 영향을 주지 않으려면 다음과 같이 "catch-all" 웹후크를 설정하지 마십시오.

```
- name: "pod-policy.example.com"
  rules:
  - apiGroups:   ["*"]
    apiVersions: ["*"]
    operations:  ["*"]
```

```
    resources:   ["*"]
    scope: "*"
```

또는 웹후크를 사용할 수 없는 경우 클러스터 중요 워크로드가 손상되지 않도록 웹후크에 제한 시간이 30초 미만인 페일 오픈 정책이 있는지 확인합니다.

안전하지 않은 블록 포드 `sysctls`

`Sysctl`는 사용자가 런타임 중에 커널 파라미터를 수정할 수 있는 Linux 유틸리티입니다. 이러한 커널 파라미터는 네트워크, 파일 시스템, 가상 메모리 및 프로세스 관리와 같은 운영 체제 동작의 다양한 측면을 제어합니다.

Kubernetes를 사용하면 포드에 대한 `sysctl` 프로필을 할당할 수 있습니다. Kubernetes는 안전 및 안전하지 않은 `systcls` 것으로 분류됩니다. 안전`sysctls`은 컨테이너 또는 포드에 네임스페이스가 있으며, 이를 설정해도 노드 또는 노드 자체의 다른 포드에는 영향을 주지 않습니다. 반면 안전하지 않은 sysctl은 다른 포드를 중단하거나 노드를 불안정하게 만들 수 있으므로 기본적으로 비활성화됩니다.

안전하지 않은 `sysctls`는 기본적으로 비활성화되어 있으므로 kubelet은 안전하지 않은 `sysctl` 프로파일이 있는 포드를 생성하지 않습니다. 이러한 포드를 생성하면 스케줄러는 노드를 시작하지 못하는 동안 해당 포드를 노드에 반복적으로 할당합니다. 이 무한 루프는 궁극적으로 클러스터 컨트롤 플레인을 손상시켜 클러스터를 불안정하게 만듭니다.

안전하지 않은 포드를 거부하려면 OPA Gatekeeper 또는 Kyverno를 사용하는 것이 좋습니다`sysctls`.

클러스터 업그레이드 처리

2021년 4월 이후 Kubernetes 릴리스 주기가 연간 릴리스 4개(분기당 1회)에서 연간 릴리스 3개로 변경되었습니다. 약 15주마다 새로운 마이너 버전(예: 1.21 또는 1.22)이 릴리스됩니다. Kubernetes 1.19부터 각 마이너 버전은 처음 릴리스된 후 약 12개월 동안 지원됩니다. Kubernetes v1.28이 출시되면서 컨트롤 플레인과 작업자 노드 간의 호환성 편차가 n-2에서 n-3 마이너 버전으로 확장되었습니다. 자세한 내용은 클러스터 업그레이드 모범 사례를 참조하세요.

클러스터 엔드포인트 연결

Amazon EKS(Elastic Kubernetes Service)로 작업할 때 Kubernetes 컨트롤 플레인 조정 또는 패치 적용과 같은 이벤트 중에 연결 시간 초과 또는 오류가 발생할 수 있습니다. 이러한 이벤트로 인해 kube-apiserver 인스턴스가 교체되어 FQDN을 해결할 때 다른 IP 주소가 반환될 수 있습니다. 이 문서에서는 Kubernetes API 소비자가 안정적인 연결을 유지하기 위한 모범 사례를 간략하게 설명합니다.

> **Note**
>
> 이러한 모범 사례를 구현하려면 새로운 DNS 재해상 및 재시도 전략을 효과적으로 처리하기 위해 클라이언트 구성 또는 스크립트를 업데이트해야 할 수 있습니다.

주요 문제는 DNS 클라이언트 측 캐싱과 퍼블릭 엔드포인트의 퍼블릭 NLB 또는 프라이빗 엔드포인트의 X-ENI 등 EKS 엔드포인트의 오래된 IP 주소 가능성에서 비롯됩니다. kube-apiserver 인스턴스가 교체되면 FQDN(정규화된 도메인 이름)이 새 IP 주소로 확인될 수 있습니다. 그러나 AWS 관리형 Route 53 영역에서 60초로 설정된 DNS TTL(Time to Live) 설정으로 인해 클라이언트는 짧은 시간 동안 오래된 IP 주소를 계속 사용할 수 있습니다.

이러한 문제를 완화하기 위해 Kubernetes API 소비자(예: kubectl, CI/CD 파이프라인 및 사용자 지정 애플리케이션)는 다음 모범 사례를 구현해야 합니다.

- DNS 재분해 구현
- 백오프 및 지터를 사용하여 재시도를 구현합니다. 예를 들어 Failures Happen이라는 제목의이 문서를 참조하세요.
- 클라이언트 제한 시간을 구현합니다. 장기 실행 요청이 애플리케이션을 차단하지 못하도록 적절한 제한 시간을 설정합니다. 일부 Kubernetes 클라이언트 라이브러리, 특히 OpenAPI 생성기에서 생성된 라이브러리는 사용자 지정 제한 시간을 쉽게 설정하지 못할 수 있습니다.
  - kubectl이 있는 예제 1:

```
kubectl get pods --request-timeout 10s # default: no timeout
```

  - Python을 사용하는 예제 2: Kubernetes 클라이언트는 _request_timeout 파라미터를 제공합니다.

이러한 모범 사례를 구현하면 Kubernetes API와 상호 작용할 때 애플리케이션의 안정성과 복원력을 크게 개선할 수 있습니다. 특히 시뮬레이션된 장애 조건에서 이러한 구현을 철저히 테스트하여 실제 조정 또는 패치 이벤트 중에 예상대로 작동하는지 확인해야 합니다.

대규모 클러스터 실행

EKS는 컨트롤 플레인 인스턴스의 로드를 능동적으로 모니터링하고 자동으로 규모를 조정하여 고성능을 보장합니다. 그러나 대규모 클러스터를 실행할 때는 AWS 서비스의 Kubernetes 및 할당량 내에서 발생할 수 있는 성능 문제와 제한을 고려해야 합니다.

- 서비스가 1,000개 이상인 클러스터는 ProjectCalico 팀이 수행한 테스트에 따라 `kube-proxy iptables` 모드에서를 사용할 때 네트워크 지연 시간이 발생할 수 있습니다. 해결 방법은 `ipvs` 모드에서 실행 중으로 전환하는 `kube-proxy` 것입니다.
- CNI가 포드의 IP 주소를 요청해야 하거나 새 EC2 인스턴스를 자주 생성해야 하는 경우에도 EC2 API 요청 제한이 발생할 수 있습니다. EC2 IP 주소를 캐싱하도록 CNI를 구성하여 EC2 API 호출을 줄일 수 있습니다. 더 큰 EC2 인스턴스 유형을 사용하여 EC2 조정 이벤트를 줄일 수 있습니다.

추가 리소스:

- Amazon EKS 워커 노드에 대한 클러스터 네트워킹 설명 해제
- Amazon EKS 클러스터 엔드포인트 액세스 제어
- AWS re:Invent 2019: Amazon EKS under the hood(CON421-R1)

EKS 데이터 영역

가용성과 복원력이 뛰어난 애플리케이션을 운영하려면 가용성과 복원력이 뛰어난 데이터 영역이 필요합니다. 탄력적 데이터 영역은 Kubernetes가 애플리케이션을 자동으로 확장하고 복구할 수 있도록 합니다. 복원력이 뛰어난 데이터 영역은 두 개 이상의 작업자 노드로 구성되며 워크로드에 따라 확장 및 축소되고 장애로부터 자동으로 복구될 수 있습니다.

EKS를 사용하는 작업자 노드에는 EKS Auto Mode 관리형 노드, EC2 인스턴스 및 Fargate 등 여러 가지 옵션이 있습니다.

EKS Auto Mode는 복원력이 뛰어난 데이터 영역으로 가는 가장 쉬운 경로를 제공합니다. Auto Mode는 Kubernetes 클러스터의 AWS 관리를 클러스터 자체 이상으로 확장하여 AWS가 워크로드를 원활하게 운영할 수 있는 인프라를 설정하고 관리할 수 있도록 합니다. Auto Mode는 Kubernetes가 포드를 확장하고 클러스터의 노드가 현재 실행 중인 워크로드에 적합하고 비용 효율적으로 크기가 조정되도록 지속적으로 작동하면서 데이터 영역을 자동으로 확장하거나 축소합니다.

EC2 인스턴스를 선택하는 경우 작업자 노드를 직접 관리하거나 EKS 관리형 노드 그룹을 사용할 수 있습니다. 자동 모드, 관리형 자체 관리형 작업자 노드 및 Fargate가 혼합된 클러스터를 보유할 수 있습니다.

Fargate는 격리된 컴퓨팅 환경에서 각 포드를 실행합니다. Fargate에서 실행되는 각 포드는 자체 작업자 노드를 가져옵니다. Fargate는 Kubernetes가 포드를 조정함에 따라 데이터 영역을 자동으로 조정합니다. 가로 포드 오토스케일러를 사용하여 데이터 영역과 워크로드를 모두 확장할 수 있습니다.

EC2 작업자 노드를 조정하는 기본 방법은(AWS에서 자동으로 수행하는 EKS Auto Mode를 사용하지 않는 경우) Karpenter, Kubernetes Cluster Autoscaler 또는 EC2 Auto Scaling 그룹을 사용하는 것입니다.

권장 사항

여러 AZs에 작업자 노드 및 워크로드 분산

여러 AZ에서 작업자 노드와 포드를 실행하여 개별 AZs. 노드를 생성하는 서브넷을 사용하여에서 작업자 노드가 생성되는 AZ를 제어할 수 있습니다.

AZs 간에 포드를 분산하는 데 권장되는 방법은 포드에 토폴로지 분산 제약 조건을 사용하는 것입니다. EKS Auto Mode 및 Karpenter와 같은 Auto Scaling 기능은 토폴로지 분산 제약 조건을 인식하며 제약 조건을 충족할 수 있도록 올바른 AZs에서 노드를 자동으로 시작합니다.

아래 배포는 가능한 경우 포드를 AZs에 분산시켜 해당 포드가 실행되도록 합니다.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
    spec:
      topologySpreadConstraints:
        - maxSkew: 1
          whenUnsatisfiable: ScheduleAnyway
          topologyKey: topology.kubernetes.io/zone
          labelSelector:
            matchLabels:
              app: web-server
      containers:
      - name: web-app
        image: nginx
        resources:
```

```
          requests:
            cpu: 1
```

> **Note**
>
> kube-scheduler는 해당 레이블과 함께 존재하는 노드를 통해서만 토폴로지 도메인을 인식합니다. 위의 배포가 단일 영역에 노드만 있는 클러스터에 배포되는 경우 모든 포드는 다른 영역을 인식하지 kube-scheduler 못하므로 해당 노드에서 예약됩니다. 이 토폴로지가 스케줄러에서 예상대로 작동하려면 노드가 모든 영역에 이미 있어야 합니다. 토폴로지 분산 제약 조건의 minDomains 속성은이 문제를 방지하기 위해 노드가 실행 중인 경우에도 스케줄러에 적격 도메인 수를 알리는 데 사용됩니다.

> **Warning**
>
> whenUnsatisfiable 로 설정DoNotSchedule하면 토폴로지 분산 제약 조건을 충족할 수 없는 경우 포드를 예약할 수 없습니다. 토폴로지 분산 제약 조건을 위반하는 대신 포드를 실행하지 않는 것이 바람직한 경우에만 설정해야 합니다.

이전 버전의 Kubernetes에서는 포드 안티 친화도 규칙을 사용하여 여러 AZs. 아래 매니페스트는 개별 AZs에서 포드를 예약하는 것을 선호하도록 Kubernetes 스케줄러에 알립니다.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
  labels:
    app: web-server
spec:
  replicas: 4
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
    spec:
      affinity:
```

```
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - web-server
              topologyKey: failure-domain.beta.kubernetes.io/zone
            weight: 100
      containers:
      - name: web-app
        image: nginx
```

> **Warning**
>
> 개별 AZs에서 포드를 예약할 필요가 없습니다. 그렇지 않으면 배포의 포드 수가 AZs 수를 초과하지 않습니다.

EBS 볼륨을 사용할 때 각 AZ에서 노드를 시작할 수 있는지 확인

Amazon EBS를 사용하여 영구 볼륨을 제공하는 경우 포드와 연결된 EBS 볼륨이 동일한 AZ에 있는지 확인해야 합니다. 포드는 다른 AZ에 있는 EBS 지원 영구 볼륨에 액세스할 수 없습니다. Kubernetes 스케줄러는 노드에 있는 레이블에서 작업자 노드가 있는 AZ를 알고 있으며 항상 볼륨과 동일한 AZ에 EBS 볼륨이 필요한 포드를 예약합니다. 그러나 볼륨이 위치한 AZ에 사용 가능한 작업자 노드가 없는 경우 포드를 예약할 수 없습니다.

EKS Auto Mode 또는 Karpenter를 사용하는 경우 NodeClass가 각 AZ에서 서브넷을 선택하는지 확인해야 합니다. 관리형 노드 그룹을 사용하는 경우 각 AZ에 노드 그룹이 있는지 확인해야 합니다.

EBS 스토리지 기능은 EKS Auto Mode에 내장되어 있지만 Karpenter 또는 Managed Node Groups를 사용하는 경우 EBS CSI도 설치해야 합니다.

EKS Auto Mode를 사용하여 작업자 노드 관리

EKS Auto Mode는 운영 오버헤드를 최소화하면서 프로덕션 지원 클러스터를 제공하여 EKS 관리를 간소화합니다. Auto Mode는 클러스터에서 실행 중인 포드에 따라 노드 수를 늘리거나 줄입니다. 노드는 소프트웨어 패치 및 수정 사항을 통해 자동으로 최신 상태로 유지되며, 업데이트는 구성된 NodePool 중단 설정 및 포드 중단 예산에 따라 수행됩니다.

노드 모니터링 에이전트 실행

노드 모니터링 에이전트는 Kubernetes 이벤트를 게시하고 노드에서 상태 조건을 업데이트하여 노드 상태 문제를 모니터링하고 대응합니다. 노드 모니터링 에이전트는 EKS Auto Mode 노드에 포함되어 있으며 Auto Mode에서 관리하지 않는 노드에 대한 EKS 추가 기능으로 설치할 수 있습니다.

EKS Auto Mode, Managed Node Groups 및 Karpenter는 모두 노드 모니터링 에이전트가 보고한 치명적인 노드 조건을 감지하고 이러한 조건이 발생할 때 해당 노드를 자동으로 복구할 수 있습니다.

QoS 구현

중요한 애플리케이션의 경우 포드의 컨테이너에 대해 `requests=limits`를 정의하는 것이 좋습니다. 이렇게 하면 다른 포드가 리소스를 요청하는 경우 컨테이너가 종료되지 않습니다.

컨테이너가 실수로 시스템 리소스를 소비하여 다른 공동 위치 프로세스의 가용성에 영향을 미치는 것을 방지하기 때문에 모든 컨테이너에 CPU 및 메모리 제한을 구현하는 것이 가장 좋습니다.

모든 워크로드에 대한 리소스 요청/제한 구성 및 크기 조정

몇 가지 일반 지침을 리소스 요청 크기 조정 및 워크로드 제한에 적용할 수 있습니다.

- CPU에 리소스 제한을 지정하지 마십시오. 제한이 없는 경우 요청은 컨테이너가 가져오는 상대 CPU 시간 양에 대한 가중치 역할을 합니다. 이를 통해 워크로드는 인공적인 제한이나 결핍 없이 전체 CPU를 사용할 수 있습니다.
- 비 CPU 리소스의 경우 구성 `requests=limits`는 가장 예측 가능한 동작을 제공합니다. `requests!=limits`인 경우 컨테이너의 QOS도 보장에서 버스트 가능으로 감소하여 노드 압력 발생 시 제거될 가능성이 높아집니다.
- CPU가 아닌 리소스의 경우 요청보다 훨씬 큰 제한을 지정하지 마십시오. `limits` 상대적으로 크기가 클`requests`수록 노드가 과도하게 커밋되어 워크로드 중단 가능성이 높아집니다.
- 올바른 크기의 요청은 Karpenter 또는 Cluster AutoScaler와 같은 노드 Auto Scaling 솔루션을 사용할 때 특히 중요합니다. 이러한 도구는 워크로드 요청을 검토하여 프로비저닝할 노드의 수와 크기를 결정합니다. 요청이 너무 작아 제한이 큰 경우 노드에 밀접하게 압축된 워크로드가 제거되거나 OOM이 종료될 수 있습니다.

리소스 요청을 결정하는 것은 어려울 수 있지만 Vertical Pod Autoscaler와 같은 도구는 런타임 시 컨테이너 리소스 사용량을 관찰하여 요청을 '적합하게' 조정하는 데 도움이 될 수 있습니다. 요청 크기를 결정하는 데 유용할 수 있는 다른 도구는 다음과 같습니다.

- 골드록

- Parca
- Prodfiler
- rsg

네임스페이스에 대한 리소스 할당량 구성

네임스페이스는 많은 사용자가 여러 팀 또는 프로젝트에 분산되어 있는 환경에서 사용하기 위한 것입니다. 이름 범위를 제공하며 여러 팀, 프로젝트, 워크로드 간에 클러스터 리소스를 분할하는 방법입니다. 네임스페이스에서 집계 리소스 소비를 제한할 수 있습니다. `ResourceQuota` 객체는 네임스페이스에서 유형별로 생성할 수 있는 객체의 수량과 해당 프로젝트의 리소스에서 사용할 수 있는 컴퓨팅 리소스의 총량을 제한할 수 있습니다. 지정된 네임스페이스에서 요청할 수 있는 스토리지 및/또는 컴퓨팅(CPU 및 메모리) 리소스의 총합을 제한할 수 있습니다.

CPU 및 메모리와 같은 컴퓨팅 리소스의 네임스페이스에 리소스 할당량이 활성화된 경우 사용자는 해당 네임스페이스의 각 컨테이너에 대한 요청 또는 제한을 지정해야 합니다.

각 네임스페이스에 대한 할당량을 구성하는 것이 좋습니다. `LimitRanges`를 사용하여 네임스페이스 내의 컨테이너에 사전 구성된 제한을 자동으로 적용하는 것이 좋습니다.

네임스페이스 내에서 컨테이너 리소스 사용량 제한

Resource Quotas는 네임스페이스가 사용할 수 있는 리소스의 양을 제한하는 데 도움이 됩니다. `LimitRange` 객체는 컨테이너가 요청할 수 있는 최소 및 최대 리소스를 구현하는 데 도움이 될 수 있습니다. 를 사용하면 컨테이너에 대한 기본 요청 및 제한을 설정할 `LimitRange` 수 있습니다. 이는 컴퓨팅 리소스 제한을 설정하는 것이 조직의 표준 관행이 아닌 경우 유용합니다. 이름에서 알 수 있듯이 `LimitRange`는 네임스페이스에서 포드 또는 컨테이너당 최소 및 최대 컴퓨팅 리소스 사용량을 적용할 수 있습니다. 또한 네임스페이스에서 PersistentVolumeClaim당 최소 및 최대 스토리지 요청을 적용합니다.

컨테이너 및 네임스페이스 수준에서 제한을 적용`ResourceQuota`하려면와 `LimitRange` 함께를 사용하는 것이 좋습니다. 이러한 제한을 설정하면 컨테이너 또는 네임스페이스가 클러스터의 다른 테넌트가 사용하는 리소스에 영향을 주지 않습니다.

NodeLocal DNSCache 사용

NodeLocal DNSCache 성능을 개선할 수 있습니다. 이 기능은 클러스터 노드에서 DNS 캐싱 에이전트를 DaemonSet로 실행합니다. 모든 포드는 `kube-dns` 서비스를 사용하는 대신 이름 확인을 위해 노드에서 실행되는 DNS 캐싱 에이전트를 사용합니다. 이 기능은 EKS Auto Mode에 자동으로 포함됩니다.

Auto Scaling CoreDNS 구성

클러스터 DNS 성능을 개선하는 또 다른 방법은 CoreDNS 포드의 기본 자동 크기 조정을 활성화하는 것입니다.

이 기능은 노드 및 CPU 코어 수를 포함하여 클러스터 상태를 지속적으로 모니터링합니다. 해당 정보를 기반으로 컨트롤러는 EKS 클러스터에서 CoreDNS 배포의 복제본 수를 동적으로 조정합니다.

네트워킹 모범 사례

클러스터와 애플리케이션을 효율적으로 운영하려면 Kubernetes 네트워킹을 이해하는 것이 중요합니다. 클러스터 네트워킹이라고도 하는 포드 네트워킹은 Kubernetes 네트워킹의 중심입니다. Kubernetes는 클러스터 네트워킹을 위한 컨테이너 네트워크 인터페이스(CNI) 플러그인을 지원합니다.

Amazon EKS는 Kubernetes 포드 네트워킹을 구현하기 위해 Amazon Virtual Private Cloud(VPC) CNI 플러그인을 공식적으로 지원합니다. VPC CNI는 AWS VPC와의 기본 통합을 제공하며 언더레이 모드에서 작동합니다. 언더레이 모드에서 포드와 호스트는 동일한 네트워크 계층에 위치하고 네트워크 네임스페이스를 공유합니다. 포드의 IP 주소는 클러스터 및 VPC 관점에서 일관됩니다.

이 가이드에서는 Kubernetes 클러스터 네트워킹의 맥락에서 Amazon VPC 컨테이너 네트워크 인터페이스(VPC CNI)를 소개합니다. VPC CNI는 EKS에서 지원하는 기본 네트워킹 플러그인이므로 가이드에서 중점적으로 다룹니다. VPC CNI는 다양한 사용 사례를 지원하도록 고도로 구성할 수 있습니다. 이 가이드에는 다양한 VPC CNI 사용 사례, 운영 모드, 하위 구성 요소에 대한 전용 섹션과 권장 사항이 추가로 포함되어 있습니다.

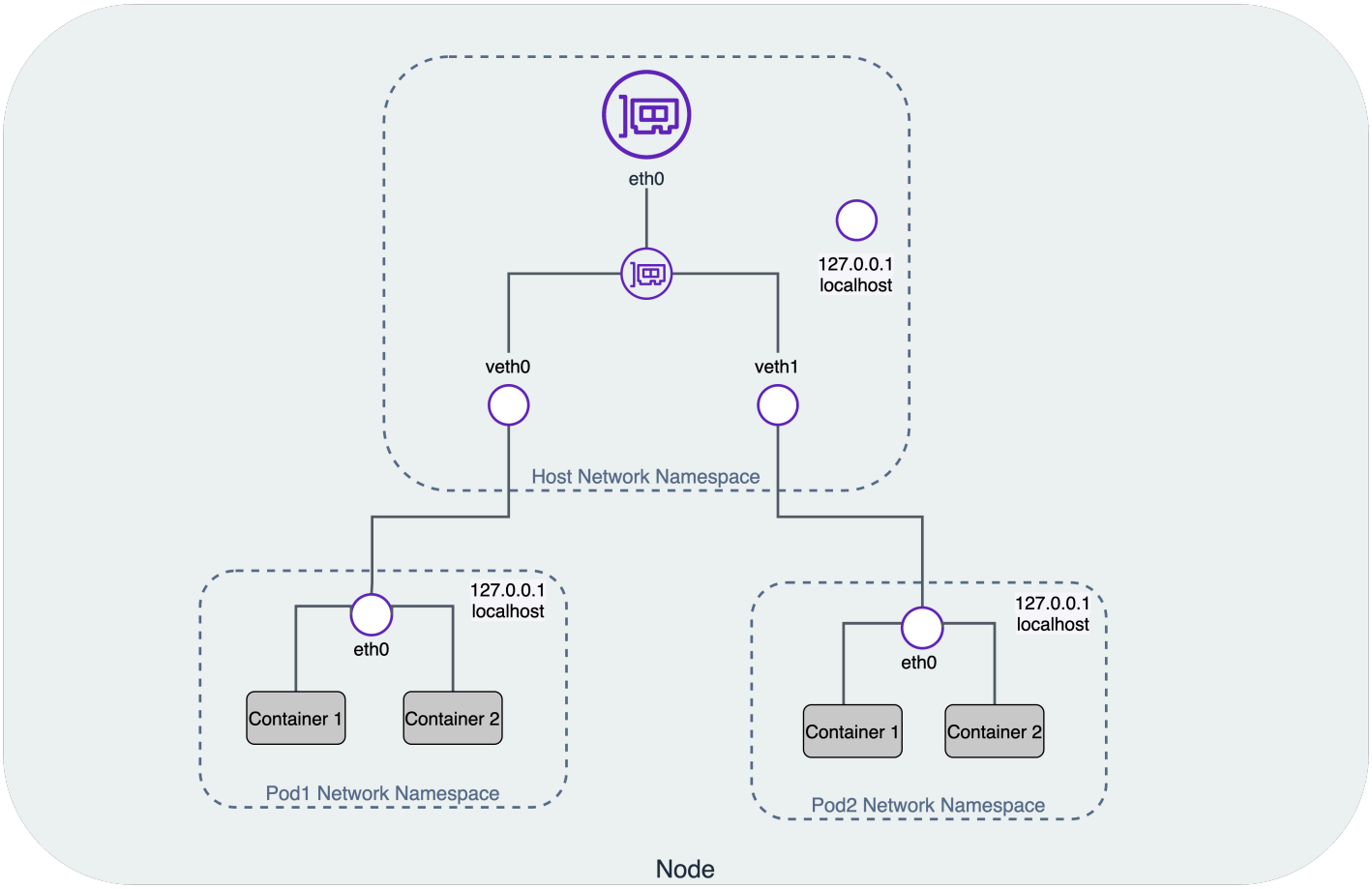
Amazon EKS는 업스트림 Kubernetes를 실행하며 Kubernetes 준수 인증을 받았습니다. 대체 CNI 플러그인을 사용할 수 있지만 이 가이드에서는 대체 CNIs를 관리하기 위한 권장 사항을 제공하지 않습니다. 대체 CNI를 효과적으로 관리하기 위한 파트너 및 리소스 목록은 EKS 대체 CNIs 설명서를 참조하십시오.

Kubernetes 네트워킹 모델

Kubernetes는 클러스터 네트워킹에 대해 다음과 같은 요구 사항을 설정합니다.

- 동일한 노드에 예약된 포드는 NAT(Network Address Translation)를 사용하지 않고 다른 포드와 통신할 수 있어야 합니다.
- 특정 노드에서 실행되는 모든 시스템 데몬(백그라운드 프로세스, 예: kubelet)은 동일한 노드에서 실행되는 포드와 통신할 수 있습니다.
- 호스트 네트워크를 사용하는 포드는 NAT를 사용하지 않고 다른 모든 노드의 다른 모든 포드에 연결할 수 있어야 합니다.

호환되는 네트워킹 구현에서 Kubernetes가 기대하는 사항에 대한 자세한 내용은 Kubernetes 네트워크 모델을 참조하세요. 다음 그림은 포드 네트워크 네임스페이스와 호스트 네트워크 네임스페이스 간의 관계를 보여줍니다.



컨테이너 네트워킹 인터페이스(CNI)

Kubernetes는 Kubernetes 네트워크 모델을 구현하기 위해 CNI 사양 및 플러그인을 지원합니다. CNI는 지원되는 여러 플러그인과 함께 플러그인을 작성하여 컨테이너에서 네트워크 인터페이스를 구성하기 위한 사양(최신 버전 1.0.0) 및 라이브러리로 구성됩니다. CNI는 컨테이너의 네트워크 연결과 컨테이너가 삭제될 때 할당된 리소스 제거에만 관여합니다.

CNI 플러그인은 kubelet에 `--network-plugin=cni` 명령줄 옵션을 전달하여 활성화됩니다. Kubelet은 `--cni-conf-dir` (기본 /etc/cni/net.d)에서 파일을 읽고 해당 파일의 CNI 구성을 사용하여 각 포드의 네트워크를 설정합니다. CNI 구성 파일은 CNI 사양(최소 v0.4.0)과 일치해야 하며 구성에서 참조하는 필수 CNI 플러그인이 `--cni-bin-dir` 디렉터리(기본 /opt/cni/bin)에 있어야 합니다. 디렉터리에 CNI 구성 파일이 여러 개 있는 경우 kubelet은 이름별로 가장 먼저 어휘 순서로 오는 구성 파일을 사용합니다.

Amazon Virtual Private Cloud(VPC) CNI

AWS에서 제공하는 VPC CNI는 EKS 클러스터의 기본 네트워킹 추가 기능입니다. VPC CNI 추가 기능은 EKS 클러스터를 프로비저닝할 때 기본적으로 설치됩니다. VPC CNI는 Kubernetes 작업자 노드에서 실행됩니다. VPC CNI 추가 기능은 CNI 바이너리와 IP 주소 관리(ipamd) 플러그인으로 구성됩니다. CNI는 VPC 네트워크의 IP 주소를 포드에 할당합니다. ipamd는 각 Kubernetes 노드에 대한 AWS 탄력적 네트워킹 인터페이스(ENIs)를 관리하고 IPs의 웜 풀을 유지합니다. VPC CNI는 빠른 포드 시작 시간을 위한 ENIs 및 IP 주소의 사전 할당을 위한 구성 옵션을 제공합니다. 권장 플러그인 관리 모범 사례는 Amazon VPC CNI를 참조하세요.

Amazon EKS에서는 클러스터를 생성할 때 두 개 이상의 가용 영역에 서브넷을 지정하는 것이 좋습니다. Amazon VPC CNI는 노드 서브넷의 포드에 IP 주소를 할당합니다. 서브넷에서 사용 가능한 IP 주소를 확인하는 것이 좋습니다. EKS 클러스터를 배포하기 전에 VPC 및 서브넷 권장 사항을 고려하세요.

Amazon VPC CNI는 노드의 기본 ENIs에 연결된 서브넷에서 ENI 및 보조 IP 주소의 웜 풀을 할당합니다. VPC CNI의이 모드를 보조 IP 모드라고 합니다. IP 주소 수와 따라서 포드 수(포드 밀도)는 인스턴스 유형에 의해 정의된 ENIs 수와 ENI당 IP 주소(제한)로 정의됩니다. 보조 모드는 기본값이며 인스턴스 유형이 작은 소규모 클러스터에 적합합니다. 포드 밀도 문제가 발생하는 경우 접두사 모드를 사용하는 것이 좋습니다. ENIs.

Amazon VPC CNI는 기본적으로 AWS VPC와 통합되며 사용자는 Kubernetes 클러스터 구축을 위한 기존 AWS VPC 네트워킹 및 보안 모범 사례를 적용할 수 있습니다. 여기에는 네트워크 트래픽 격리를 위해 VPC 흐름 로그, VPC 라우팅 정책 및 보안 그룹을 사용하는 기능이 포함됩니다. 기본적으로 Amazon VPC CNI는 노드의 기본 ENI와 연결된 보안 그룹을 포드에 적용합니다. 포드에 대해 다른 네트워크 규칙을 할당하려는 경우 포드에 대한 보안 그룹을 활성화하는 것이 좋습니다.

기본적으로 VPC CNI는 노드의 기본 ENI에 할당된 서브넷의 포드에 IP 주소를 할당합니다. 수천 개의 워크로드가 있는 대규모 클러스터를 실행할 때 IPv4 주소가 부족해지는 것이 일반적입니다. AWS VPC를 사용하면 IPv4 CIDRs 블록의 소진을 해결할 보조 CIDR을 할당하여 사용 가능한 IPs를 확장할 수 있습니다. AWS VPC CNI를 사용하면 포드에 대해 다른 서브넷 CIDR 범위를 사용할 수 있습니다. VPC CNI의이 기능을 사용자 지정 네트워킹이라고 합니다. 사용자 지정 네트워킹을 사용하여 EKS에서 `100.64.0.0/10` 및 `198.19.0.0/16` CIDRs(CG-NAT)을 사용하는 것이 좋습니다. 이렇게 하면 포드가 더 이상 VPC의 RFC1918 IP 주소를 사용하지 않는 환경을 효과적으로 생성할 수 있습니다.

사용자 지정 네트워킹은 IPv4 주소 소진 문제를 해결하기 위한 한 가지 옵션이지만 운영 오버헤드가 필요합니다. 이 문제를 해결하려면 사용자 지정 네트워킹보다 IPv6 클러스터를 사용하는 것이 좋습니다. 특히 VPC에 사용 가능한 IPv4 주소 공간이 모두 소진된 경우 IPvIPv6 클러스터로 마이그레이션하는 것이 좋습니다. 조직의 IPv6 지원 계획을 평가하고 IPv6에 대한 투자가 장기적인 가치가 더 높을 수 있는지 고려합니다.

EKS의 IPv6 지원은 제한된 IPv4 주소 공간으로 인한 IP 소진 문제를 해결하는 데 중점을 둡니다. IPv4 소진과 관련된 고객 문제에 대응하여 EKS는 듀얼 스택 포드보다 IPv6-only 포드를 우선시했습니다. 즉, 포드는 IPv4 리소스에 액세스할 수 있지만 VPC CIDR 범위에서 IPv4 주소가 할당되지 않습니다. VPC CNI는 AWS 관리형 VPC IPv6 CIDR 블록의 포드에 IPv6 주소를 할당합니다.

서브넷 계산기

이 프로젝트에는 서브넷 계산기 Excel 문서가 포함되어 있습니다. 이 계산기 문서는 `WARM_IP_TARGET` 및와 같은 다양한 ENI 구성 옵션에서 지정된 워크로드의 IP 주소 소비를 시뮬레이션합니다`WARM_ENI_TARGET`. 문서에는 두 개의 시트가 포함되어 있습니다. 하나는 웜 ENI 모드용이고 다른 하나는 웜 IP 모드용입니다. 이러한 모드에 대한 자세한 내용은 VPC CNI 지침을 검토하세요.

입력:

- 서브넷 CIDR 크기
- 웜 ENI 대상 또는 웜 IP 대상
- 인스턴스 목록
  - 인스턴스당 예약된 워크로드 포드의 유형, 수 및 수

출력:

- 호스팅된 총 포드 수
- 사용된 서브넷 IPs 수
- 남은 서브넷 IPs 수
- 인스턴스 수준 세부 정보
  - 인스턴스당 웜 IPs/ENIs 수
  - 인스턴스당 활성 IPs/ENIs 수

VPC 및 서브넷 고려 사항

EKS 클러스터를 운영하려면 Kubernetes 네트워킹 외에도 AWS VPC 네트워킹에 대한 지식이 필요합니다.

VPC를 설계하거나 기존 VPC에 클러스터를 배포하기 전에 EKS 컨트롤 플레VPCs 통신 메커니즘을 이해하는 것이 좋습니다.

EKS와 함께 사용할 VPC 및 서브넷을 설계할 때는 클러스터 VPC 고려 사항 및 Amazon EKS 보안 그룹 고려 사항을 참조하세요. https://docs.aws.amazon.com/eks/latest/userguide/sec-group-reqs.html

개요

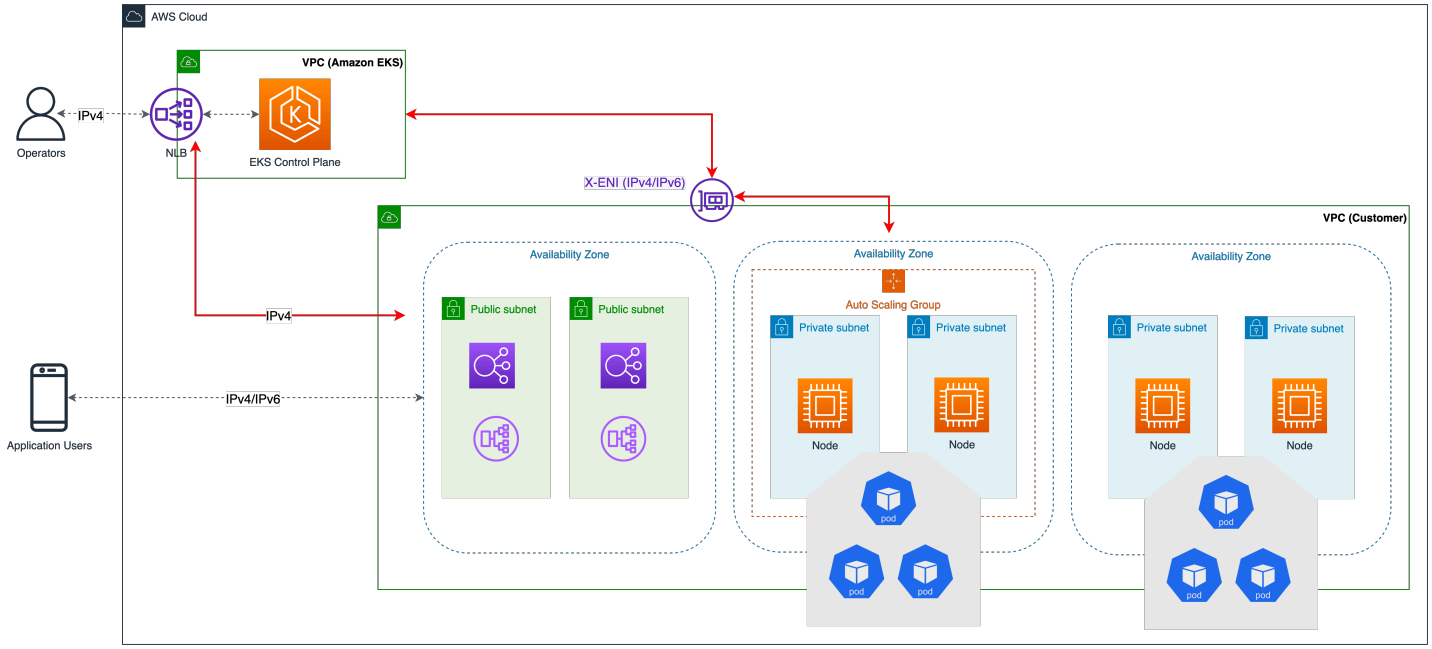
EKS 클러스터 아키텍처

EKS 클러스터는 두 개의 VPCs로 구성됩니다.

- Kubernetes 컨트롤 플레인을 호스팅하는 AWS 관리형 VPC입니다. 이 VPC는 고객 계정에 표시되지 않습니다.
- Kubernetes 노드를 호스팅하는 고객 관리형 VPC입니다. 여기서 컨테이너는 물론 클러스터에서 사용하는 로드 밸런서와 같은 기타 고객 관리형 AWS 인프라도 실행됩니다. 이 VPC는 고객 계정에 표시됩니다. 클러스터를 생성하기 전에 고객 관리형 VPC를 생성해야 합니다. eksctl은 VPC를 제공하지 않으면 VPC를 생성합니다.

고객 VPC의 노드에는 AWS VPC의 관리형 API 서버 엔드포인트에 연결할 수 있는 기능이 필요합니다. 이렇게 하면 노드가 Kubernetes 컨트롤 플레인에 등록하고 애플리케이션 포드 실행 요청을 수신할 수 있습니다.

노드는 (a) EKS 퍼블릭 엔드포인트 또는 (b) EKS에서 관리하는 교차 계정 탄력적 네트워크 인터페이스(X-ENI)를 통해 EKS 컨트롤 플레인에 연결됩니다. 클러스터를 생성할 때 VPC 서브넷을 두 개 이상 지정해야 합니다. EKS는 클러스터 생성 중에 지정된 각 서브넷(클러스터 서브넷이라고도 함)에 X-ENI를 배치합니다. Kubernetes API 서버는 이러한 교차 계정 ENIs 사용하여 고객 관리형 클러스터 VPC 서브넷에 배포된 노드와 통신합니다.



노드가 시작되면 EKS 부트스트랩 스크립트가 실행되고 Kubernetes 노드 구성 파일이 설치됩니다. 각 인스턴스의 부팅 프로세스의 일부로 컨테이너 런타임 에이전트, kubelet 및 Kubernetes 노드 에이전트가 시작됩니다.

노드를 등록하기 위해 Kubelet은 Kubernetes 클러스터 엔드포인트에 문의합니다. VPC 외부의 퍼블릭 엔드포인트 또는 VPC 내의 프라이빗 엔드포인트와 연결을 설정합니다. Kubelet은 API 지침을 수신하고 정기적으로 엔드포인트에 상태 업데이트 및 하트비트를 제공합니다.

EKS 제어 플레인 통신

EKS에는 클러스터 엔드포인트에 대한 액세스를 제어하는 두 가지 방법이 있습니다. 엔드포인트 액세스 제어를 사용하면 퍼블릭 인터넷에서 엔드포인트에 연결할 수 있는지 아니면 VPC를 통해서만 엔드포인트에 연결할 수 있는지 선택할 수 있습니다. 퍼블릭 엔드포인트(기본값), 프라이빗 엔드포인트 또는 둘 다를 한 번에 켤 수 있습니다.

클러스터 API 엔드포인트의 구성에 따라 노드가 컨트롤 플레인과 통신하는 데 걸리는 경로가 결정됩니다. 이러한 엔드포인트 설정은 EKS 콘솔 또는 API를 통해 언제든지 변경할 수 있습니다.

퍼블릭 엔드포인트

새 Amazon EKS 클러스터의 기본 동작입니다. 클러스터의 퍼블릭 엔드포인트만 활성화되면 클러스터의 VPC 내에서 시작되는 Kubernetes API 요청(예: 컨트롤 플레인 통신을 위한 작업자 노드)은 VPC를 벗어나지만 Amazon의 네트워크는 나가지 않습니다. 노드가 컨트롤 플레인에 연결되려면 퍼블릭 IP 주

소와 인터넷 게이트웨이에 대한 경로 또는 NAT 게이트웨이의 퍼블릭 IP 주소를 사용할 수 있는 NAT 게이트웨이에 대한 경로가 있어야 합니다.

퍼블릭 및 프라이빗 엔드포인트

퍼블릭 엔드포인트와 프라이빗 엔드포인트가 모두 활성화되면 VPC 내 Kubernetes API 요청은 VPC 내 X-ENIs를 통해 컨트롤 플레인과 통신합니다. 인터넷에서 클러스터 API 서버에 액세스할 수 있습니다.

프라이빗 엔드포인트

프라이빗 엔드포인트만 활성화된 경우 인터넷에서 API 서버에 대한 퍼블릭 액세스 권한이 없습니다. 클러스터 API 서버에 대한 모든 트래픽은 클러스터의 VPC 또는 연결된 네트워크 내에서 비롯되어야 합니다. 노드는 VPC 내의 X-ENIs 통해 API 서버와 통신합니다. 클러스터 관리 도구는 프라이빗 엔드포인트에 액세스할 수 있어야 합니다. Amazon VPC 외부에서 프라이빗 Amazon EKS 클러스터 엔드포인트에 연결하는 방법에 대해 자세히 알아봅니다.

클러스터의 API 서버 엔드포인트는 퍼블릭 DNS 서버에서 VPC의 프라이빗 IP 주소로 확인됩니다. 이전에는 VPC 내에서만 엔드포인트를 확인할 수 있었습니다.

VPC 구성

Amazon VPC는 IPv4 및 IPv6 주소 지정을 지원합니다. Amazon EKS는 기본적으로 IPv4를 지원합니다. VPC에는 이와 연결된 IPv4 CIDR 블록이 있어야 합니다. 선택적으로 여러 IPv4 Classless Inter-Domain Routing(CIDR) 블록과 여러 IPv6 CIDR 블록을 VPC에 연결할 수 있습니다. VPC를 생성할 때 RFC 1918에 지정된 대로 프라이빗 IPv4 주소 범위에서 VPC에 대한 IPv4 CIDR 블록을 지정해야 합니다. http://www.faqs.org/rfcs/rfc1918.html 허용되는 블록 크기는 /16 접두사(65,536 IP 주소)와 /28 접두사(16 IP 주소) 사이입니다.

새 VPC를 생성할 때 단일 IPv6 CIDR 블록을 연결하고 기존 VPC를 변경할 때 최대 5개까지 연결할 수 있습니다. IPv6 CIDR 블록 크기의 접두사 길이는 /44에서 /60 사이일 수 있으며 IPv6 서브넷의 경우 /44/에서 /64 사이일 수 있습니다. Amazon에서 유지 관리하는 IPv6 주소 풀에서 IPv6 CIDR 블록을 요청할 수 있습니다. 자세한 내용은 VPC 사용 설명서의 VPC CIDR 블록 섹션을 참조하세요.

Amazon EKS 클러스터는 IPv4와 IPv6를 모두 지원합니다. 기본적으로 EKS 클러스터는 IPv4 IP를 사용합니다. 클러스터 생성 시 IPv6를 지정하면에서 IPv6 클러스터를 사용할 수 있습니다. IPv6 클러스터에는 듀얼 스택 VPCs 및 서브넷이 필요합니다.

Amazon EKS는 클러스터 생성 중에 서로 다른 가용 영역에 있는 서브넷을 두 개 이상 사용할 것을 권장합니다. 클러스터 생성 중에 전달하는 서브넷을 클러스터 서브넷이라고 합니다. 클러스터를 생성할

때 Amazon EKS는 지정한 서브넷에 최대 4개의 크로스 계정(x-계정 또는 x-ENIs) ENIs를 생성합니다. x-ENIs는 항상 배포되며 로그 전송, exec 및 프록시와 같은 클러스터 관리 트래픽에 사용됩니다. 전체 VPC 및 서브넷 요구 사항 세부 정보는 EKS 사용 설명서를 참조하세요.

Kubernetes 작업자 노드는 클러스터 서브넷에서 실행할 수 있지만 권장되지 않습니다. 클러스터 업그레이드 중에 Amazon EKS는 클러스터 서브넷에 추가 ENIs를 프로비저닝합니다. 클러스터가 확장되면 작업자 노드와 포드가 클러스터 서브넷에서 사용 가능한 IPs를 사용할 수 있습니다. 따라서 사용 가능한 IPs지 확인하기 위해 /28 넷마스크가 있는 전용 클러스터 서브넷 사용을 고려할 수 있습니다.

Kubernetes 작업자 노드는 퍼블릭 또는 프라이빗 서브넷에서 실행할 수 있습니다. 서브넷이 퍼블릭인지 프라이빗인지는 서브넷 내의 트래픽이 인터넷 게이트웨이를 통해 라우팅되는지 여부를 나타냅니다. 퍼블릭 서브넷에는 인터넷 게이트웨이를 통해 인터넷에 대한 라우팅 테이블 항목이 있지만 프라이빗 서브넷에는 없습니다.

다른 곳에서 시작되어 노드에 도달하는 트래픽을 수신이라고 합니다. 노드에서 시작되어 네트워크에서 나가는 트래픽을 송신이라고 합니다. 인터넷 게이트웨이로 구성된 서브넷 내에 퍼블릭 또는 탄력적 IP 주소(EIPs)가 있는 노드는 VPC 외부에서 수신할 수 있습니다. 프라이빗 서브넷에는 일반적으로 NAT 게이트웨이로의 라우팅이 있습니다.이 라우팅은 VPC 외부의 서브넷에 있는 노드로의 수신 트래픽을 허용하지 않으면서 노드로부터의 트래픽이 VPC(송신)를 벗어나도록 허용합니다.

IPv6 세계에서 모든 주소는 인터넷 라우팅이 가능합니다. 노드 및 포드와 연결된 IPv6 주소는 퍼블릭입니다. 프라이빗 서브넷은 VPC에서 외부 전용 인터넷 게이트웨이(EIGW)를 구현하여 아웃바운드 트래픽을 허용하는 동시에 모든 수신 트래픽을 차단함으로써 지원됩니다. IPv6 서브넷 구현 모범 사례는 VPC 사용 설명서에서 확인할 수 있습니다.

다음과 같은 세 가지 방법으로 VPC와 서브넷을 구성할 수 있습니다.

퍼블릭 서브넷만 사용

동일한 퍼블릭 서브넷에서 노드와 수신 리소스(예: 로드 밸런서)가 모두 생성됩니다. 퍼블릭 서브넷에 태그를 지정`kubernetes.io/role/elb`하여 인터넷을 마주하는 로드 밸런서를 구성합니다. 이 구성에서 클러스터 엔드포인트는 퍼블릭, 프라이빗 또는 둘 다(퍼블릭 및 프라이빗)로 구성할 수 있습니다.

프라이빗 및 퍼블릭 서브넷 사용

노드는 프라이빗 서브넷에서 생성되는 반면 수신 리소스는 퍼블릭 서브넷에서 인스턴스화됩니다. 클러스터 엔드포인트에 대한 퍼블릭, 프라이빗 또는 둘 다(퍼블릭 및 프라이빗) 액세스를 활성화할 수 있습니다. 클러스터 엔드포인트의 구성에 따라 노드 트래픽은 NAT 게이트웨이 또는 ENI를 통해 들어갑니다.

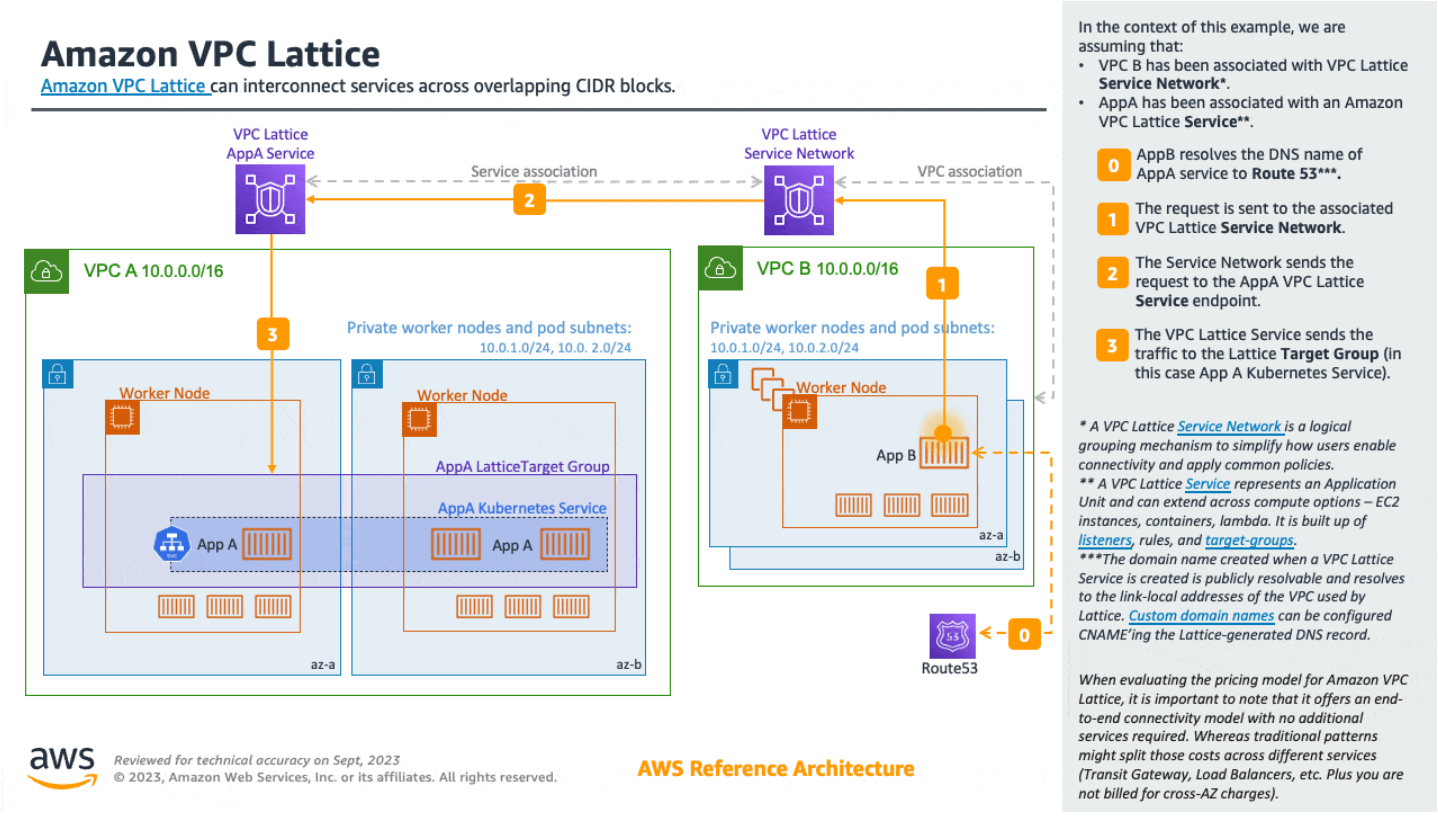
프라이빗 서브넷만 사용

노드와 수신 모두 프라이빗 서브넷에 생성됩니다. `kubernetes.io/role/internal-elb` 서브넷 태그를 사용하여 내부 로드 밸런서를 구성합니다. 클러스터의 엔드포인트에 액세스하려면 VPN 연결이 필요합니다. EC2 및 모든 Amazon ECR 및 S3 리포지토리에 대해 AWS PrivateLink를 활성화해야 합니다. 클러스터의 프라이빗 엔드포인트만 활성화해야 합니다. 프라이빗 클러스터를 프로비저닝하기 전에 EKS 프라이빗 클러스터 요구 사항을 살펴보는 것이 좋습니다.

VPCs 간 통신

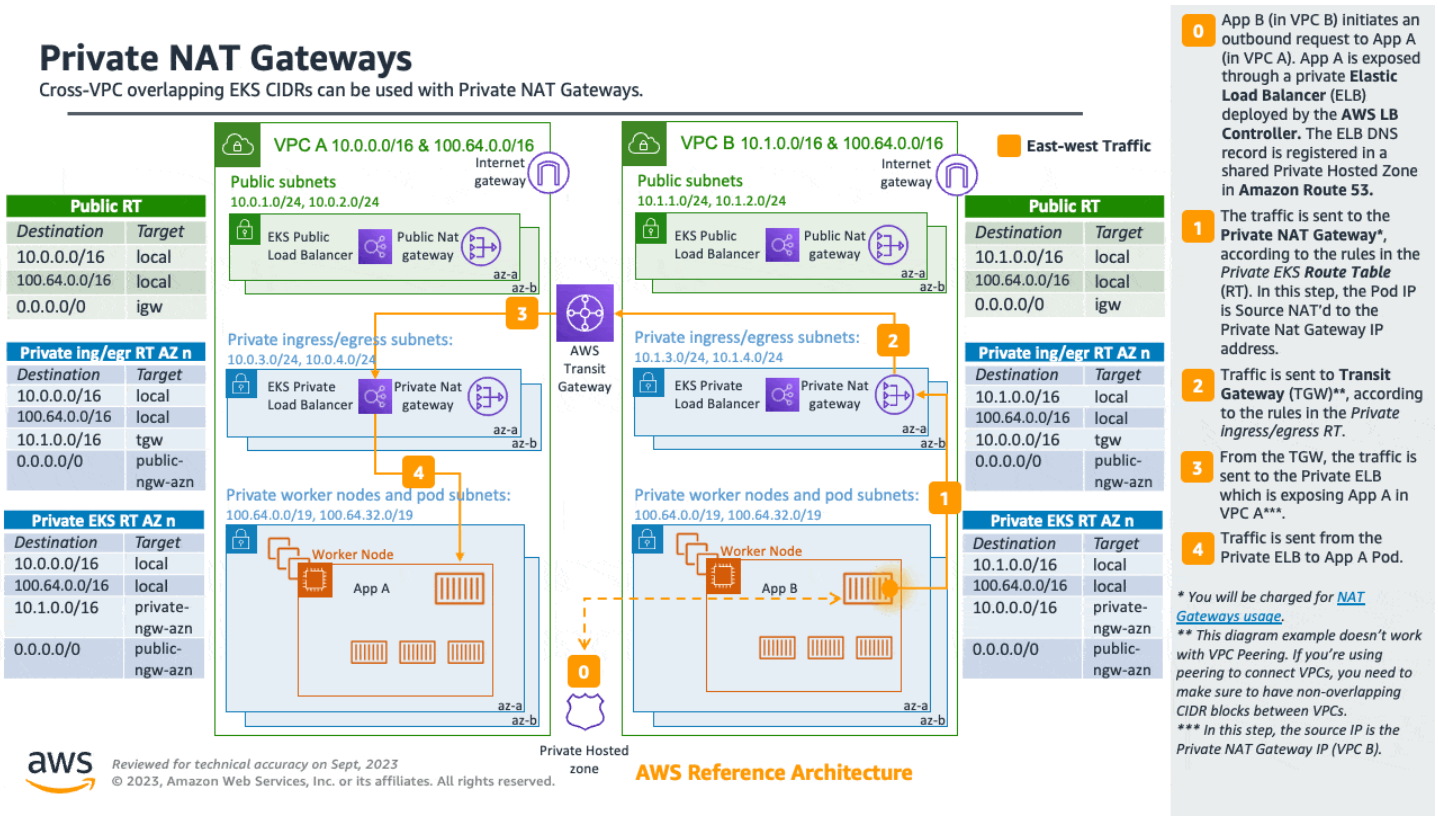
여러 VPCs에 배포된 별도의 EKS 클러스터가 필요한 경우 많은 시나리오VPCs.

Amazon VPC Lattice를 사용하여 여러 VPCs 및 계정에서 서비스를 일관되고 안전하게 연결할 수 있습니다(VPC 피어링, AWS PrivateLink 또는 AWS Transit Gateway와 같은 서비스에서 추가 연결을 제공할 필요 없음). 여기에서 자세히 알아보세요.



Amazon VPC Lattice는 IPv4 및 IPv6의 링크-로컬 주소 공간에서 작동하여 IPv4 주소가 겹칠 수 있는 서비스 간에 연결을 제공합니다. 운영 효율성을 위해 겹치지 않는 IP 범위에 EKS 클러스터 및 노드를 배포하는 것이 좋습니다. 인프라에 중복 IP 범위가 있는 VPCs 포함된 경우 그에 따라 네트워크를 설계해야 합니다. 전송 게이트웨이와 함께 프라이빗 NAT 게이트웨이 또는 사용자 지정 네트워킹 모드의

VPC CNI를 사용하여 EKS의 워크로드를 통합하여 라우팅 가능한 RFC1918 IP 주소를 유지하면서 중복되는 CIDR 문제를 해결하는 것이 좋습니다.

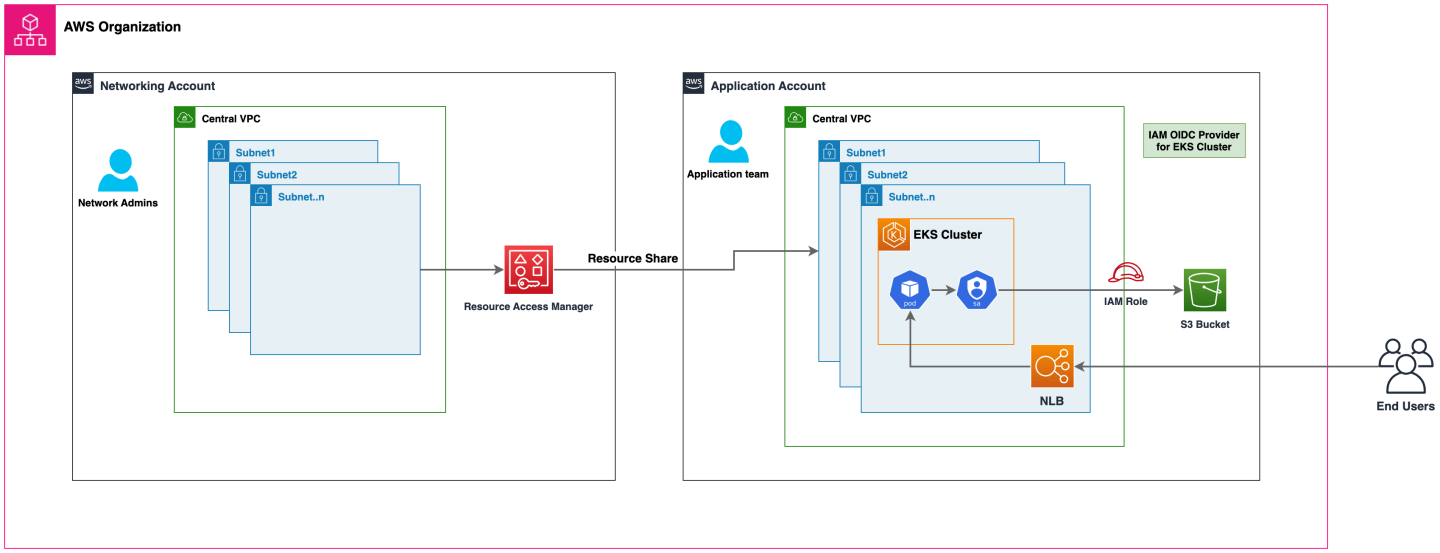


서비스 공급자이고 Kubernetes 서비스 및 수신(ALB 또는 NLB)을 별도의 계정의 고객 VPC와 공유하려는 경우 엔드포인트 서비스라고도 하는 AWS PrivateLink를 활용하는 것이 좋습니다.

여러 계정에서 VPC 공유

많은 기업이 네트워크 관리를 간소화하고, 비용을 절감하고, AWS Organization의 여러 AWS 계정에서 보안을 개선하기 위한 수단으로 공유 Amazon VPCs를 채택했습니다. AWS Resource Access Manager(RAM)를 활용하여 지원되는 AWS 리소스를 개별 AWS 계정, 조직 단위(OUs) 또는 전체 AWS Organization과 안전하게 공유합니다.

AWS RAM을 사용하여 다른 AWS 계정의 공유 VPC 서브넷에 Amazon EKS 클러스터, 관리형 노드 그룹 및 기타 지원 AWS 리소스(예: LoadBalancers, 보안 그룹, 엔드포인트 등)를 배포할 수 있습니다. 아래 그림은 상위 수준 아키텍처의 예를 보여줍니다. 이를 통해 중앙 네트워킹 팀은 VPCs, 서브넷 등과 같은 네트워킹 구성을 제어하고 애플리케이션 또는 플랫폼 팀은 해당 AWS 계정에 Amazon EKS 클러스터를 배포할 수 있습니다. 이 시나리오에 대한 전체 연습은이 github 리포지토리에서 확인할 수 있습니다.



공유 서브넷 사용 시 고려 사항

- Amazon EKS 클러스터와 작업자 노드는 모두 동일한 VPC의 일부인 공유 서브넷 내에서 생성할 수 있습니다. Amazon EKS는 여러 VPCs에서 클러스터 생성을 지원하지 않습니다.
- Amazon EKS는 AWS VPC 보안 그룹(SGs)을 사용하여 Kubernetes 컨트롤 플레인과 클러스터의 작업자 노드 간의 트래픽을 제어합니다. 보안 그룹은 작업자 노드, 기타 VPC 리소스 및 외부 IP 주소 간의 트래픽을 제어하는 데에도 사용됩니다. 애플리케이션/참가자 계정에서 이러한 보안 그룹을 생성해야 합니다. 포드에 사용하려는 보안 그룹도 참가자 계정에 있어야 합니다. 중앙 VPC 계정에 있는 보안 그룹과의 필수 트래픽을 허용하도록 보안 그룹 내의 인바운드 및 아웃바운드 규칙을 구성할 수 있습니다.
- Amazon EKS 클러스터가 있는 참가자 계정 내에서 IAM 역할 및 관련 정책을 생성합니다. 이러한 IAM 역할 및 정책은 Amazon EKS에서 관리하는 Kubernetes 클러스터와 Fargate에서 실행되는 노드 및 포드에 필요한 권한을 부여하는 데 필수적입니다. 권한을 통해 Amazon EKS는 사용자를 대신하여 다른 AWS 서비스를 호출할 수 있습니다.
- 다음 접근 방식을 따라 k8s 포드에서 Amazon S3 버킷, Dynamodb 테이블 등과 같은 AWS 리소스에 대한 교차 계정 액세스를 허용할 수 있습니다.
  - 리소스 기반 정책 접근 방식: AWS 서비스가 리소스 정책을 지원하는 경우 적절한 리소스 기반 정책을 추가하여 kubernetes 포드에 할당된 IAM 역할에 대한 교차 계정 액세스를 허용할 수 있습니다. 이 시나리오에서는 OIDC 공급자, IAM 역할 및 권한 정책이 애플리케이션 계정에 존재합니다. 리소스 기반 정책을 지원하는 AWS 서비스를 찾으려면 IAM으로 작업하는 AWS 서비스를 참조하고 리소스 기반 열에서 예인 서비스를 찾습니다.
  - OIDC 공급자 접근 방식: 리소스가 있는 다른 참가자 AWS 계정에서 OIDC 공급자, IAM 역할, 권한 및 신뢰 정책과 같은 IAM 리소스가 생성됩니다. 이러한 역할은 애플리케이션 계정의 Kubernetes

포드에 할당되므로 교차 계정 리소스에 액세스할 수 있습니다. 이 접근 방식에 대한 전체 연습은 Kubernetes 서비스 계정의 교차 계정 IAM 역할 블로그를 참조하세요.

- Amazon Elastic Loadbalancer(ELB) 리소스(ALB 또는 NLB)를 배포하여 애플리케이션 또는 중앙 네트워킹 계정의 k8s 포드로 트래픽을 라우팅할 수 있습니다. 중앙 네트워킹 계정에 ELB 리소스를 배포하는 방법에 대한 자세한 지침은 교차 계정 Load Balancer 통해 Amazon EKS 포드 노출 연습을 참조하세요. 이 옵션은 중앙 네트워킹 계정에 Load Balancer 리소스의 보안 구성을 완전히 제어할 수 있는 권한을 부여하므로 유연성이 향상됩니다.
- Amazon VPC CNIcustom networking feature를 사용하는 경우 중앙 네트워킹 계정에 나열된 가용 영역(AZ) ID 매핑을 사용하여 각를 생성해야 합니다ENIConfig. 이는 물리적 AZs 각 AWS 계정의 AZ 이름에 무작위로 매핑하기 때문입니다.

보안 그룹

보안 그룹은 연결된 리소스에 도달하고 나갈 수 있는 트래픽을 제어합니다. Amazon EKS는 보안 그룹을 사용하여 컨트롤 플레인과 노드 간의 통신을 관리합니다. 클러스터를 생성할 때 Amazon EKS에서 eks-cluster-sg-my-cluster-uniqueID라는 보안 그룹을 만듭니다. EKS는 이러한 보안 그룹을 관리형 ENIs 및 노드에 연결합니다. 기본 규칙을 사용하면 클러스터와 노드 간에 모든 트래픽이 자유롭게 흐를 수 있으며 모든 아웃바운드 트래픽은 모든 대상에 대해 허용됩니다.

클러스터를 생성할 때 자체 보안 그룹을 지정할 수 있습니다. 자체 보안 그룹을 지정할 때는 보안 그룹에 대한 권장 사항을 참조하세요.

권장 사항

다중 AZ 배포 고려

AWS 리전은 지연 시간이 짧고 처리량이 많으며 중복성이 높은 네트워킹과 연결된 물리적으로 분리되고 격리된 여러 가용 영역(AZ)을 제공합니다. 가용 영역을 사용하면 중단 없이 가용 영역 간에 자동으로 장애 조치를 수행하는 애플리케이션을 설계하고 운영할 수 있습니다. Amazon EKS는 EKS 클러스터를 여러 가용 영역에 배포할 것을 적극 권장합니다. 클러스터를 생성할 때 두 개 이상의 가용 영역에 서브넷을 지정하는 것이 좋습니다.

노드에서 실행되는 Kubelet은와 같은 노드 객체에 레이블을 자동으로 추가합니다topology.kubernetes.io/region=us-west-2. 포드 토폴로지 분산 제약 조건과 함께 노드 레이블을 사용하여 포드가 영역 간에 분산되는 방식을 제어하는 것이 좋습니다. 이러한 힌트를 통해 Kubernetes 스케줄러는 예상 가용성을 높이기 위해 포드를 배치하여 상관관계가 있는 장애가 전체 워크로드에 영향을 미칠 위험을 줄일 수 있습니다. 노드 선택기 및 AZ 분산 제약의 예를 보려면 포드에 노드 할당을 참조하세요.

노드를 생성할 때 서브넷 또는 가용 영역을 정의할 수 있습니다. 구성된 서브넷이 없는 경우 노드가 클러스터 서브넷에 배치됩니다. 관리형 노드 그룹에 대한 EKS 지원은 사용 가능한 용량의 여러 가용 영역에 노드를 자동으로 분산합니다. Karpenter는 워크로드가 토폴로지 분산 한도를 정의하는 경우 노드를 지정된 AZ로 조정하여 AZs 분산 배치를 준수합니다.

AWS Elastic Load Balancer는 Kubernetes 클러스터의 AWS Load Balancer Controller에서 관리합니다. Kubernetes 수신 리소스용 Application Load Balancer(ALB)와 Load Balancer 유형의 Kubernetes 서비스용 Network Load Balancer(NLB)를 프로비저닝합니다. Elastic Load Balancer 컨트롤러는 태그를 사용하여 서브넷을 검색합니다. ELB 컨트롤러가 수신 리소스를 성공적으로 프로비저닝하려면 최소 2개의 가용 영역(AZs)이 필요합니다. 지리적 중복성의 안전성 및 신뢰성을 활용하려면 두 개 이상의 AZs에서 서브넷을 설정하는 것이 좋습니다.

프라이빗 서브넷에 노드 배포

프라이빗 서브넷과 퍼블릭 서브넷을 모두 포함하는 VPC는 EKS에 Kubernetes 워크로드를 배포하는 데 이상적인 방법입니다. 두 개의 개별 가용 영역에서 최소 두 개의 퍼블릭 서브넷과 두 개의 프라이빗 서브넷을 설정하는 것이 좋습니다. 퍼블릭 서브넷의 관련 라우팅 테이블에는 인터넷 게이트웨이에 대한 경로가 포함되어 있습니다. 포드는 NAT 게이트웨이를 통해 인터넷과 상호 작용할 수 있습니다. 프라이빗 서브넷은 IPv6 환경(EIGW)의 외부 전용 인터넷 게이트웨이에서 지원됩니다.

프라이빗 서브넷의 노드를 인스턴스화하면 노드에 대한 트래픽을 최대한 제어할 수 있으며 대부분의 Kubernetes 애플리케이션에 효과적입니다. 수신 리소스(예: 로드 밸런서)는 퍼블릭 서브넷에서 인스턴스화되고 프라이빗 서브넷에서 작동하는 포드로 트래픽을 라우팅합니다.

엄격한 보안 및 네트워크 격리가 필요한 경우 프라이빗 전용 모드를 고려하세요. 이 구성에서는 AWS 리전의 VPC 내의 개별 가용 영역에 3개의 프라이빗 서브넷이 배포됩니다. 서브넷에 배포된 리소스는 인터넷에 액세스할 수 없으며 인터넷도 서브넷의 리소스에 액세스할 수 없습니다. Kubernetes 애플리케이션이 다른 AWS 서비스에 액세스하려면 PrivateLink 인터페이스 및/또는 게이트웨이 엔드포인트를 구성해야 합니다. AWS Load Balancer Controller를 사용하여 트래픽을 포드로 리디렉션하도록 내부 Load Balancer 설정할 수 있습니다. 컨트롤러가 로드 밸런서를 프로비저닝하려면 프라이빗 서브넷에 태그(`kubernetes.io/role/internal-elb: 1`)를 지정해야 합니다. 노드를 클러스터에 등록하려면 클러스터 엔드포인트를 프라이빗 모드로 설정해야 합니다. 전체 요구 사항 및 고려 사항은 프라이빗 클러스터 가이드를 참조하세요.

클러스터 엔드포인트에 대한 퍼블릭 및 프라이빗 모드 고려

Amazon EKS는 퍼블릭 전용, public-and-private 클러스터 엔드포인트 모드를 제공합니다. 기본 모드는 퍼블릭 전용이지만 퍼블릭 및 프라이빗 모드에서 클러스터 엔드포인트를 구성하는 것이 좋습니다. 이 옵션을 사용하면 클러스터의 VPC 내에서 Kubernetes API 호출(예: node-to-control-plane인 통신)이 프

라이빗 VPC 엔드포인트 및 트래픽을 활용하여 클러스터의 VPC 내에 유지할 수 있습니다. 반면 클러스터 API 서버는 인터넷에서 연결할 수 있습니다. 그러나 퍼블릭 엔드포인트를 사용할 수 있는 CIDR 블록을 제한하는 것이 좋습니다. CIDR 블록 제한을 포함하여 퍼블릭 및 프라이빗 엔드포인트 액세스를 구성하는 방법을 알아봅니다.

보안 및 네트워크 격리가 필요한 경우 프라이빗 전용 엔드포인트를 사용하는 것이 좋습니다. API 서버에 비공개로 연결하려면 EKS 사용 설명서에 나열된 옵션 중 하나를 사용하는 것이 좋습니다.

보안 그룹을 신중하게 구성

Amazon EKS는 사용자 지정 보안 그룹 사용을 지원합니다. 모든 사용자 지정 보안 그룹은 노드와 Kubernetes 컨트롤 플레인 간의 통신을 허용해야 합니다. 조직에서 열린 통신을 허용하지 않는 경우 포트 요구 사항을 확인하고 규칙을 수동으로 구성하십시오.

EKS는 클러스터 생성 중에 제공하는 사용자 지정 보안 그룹을 관리형 인터페이스(X-ENIs)에 적용합니다. 그러나 노드와 즉시 연결되지는 않습니다. 노드 그룹을 생성할 때는 사용자 지정 보안 그룹을 수동으로 연결하는 것이 좋습니다. 노드 자동 크기 조정 중에 사용자 지정 보안 그룹의 Karpenter 노드 템플릿 검색을 활성화하려면 securityGroupSelectorTerms를 활성화하는 것이 좋습니다.

모든 노드 간 통신 트래픽을 허용하는 보안 그룹을 생성하는 것이 좋습니다. 부트스트랩 프로세스 중에 클러스터 엔드포인트에 액세스하려면 노드에 아웃바운드 인터넷 연결이 필요합니다. 온프레미스 연결 및 컨테이너 레지스트리 액세스와 같은 외부 액세스 요구 사항을 평가하고 규칙을 적절하게 설정합니다. 프로덕션 환경에 변경 사항을 적용하기 전에 개발 환경에서 연결을 주의 깊게 확인하는 것이 좋습니다.

각 가용 영역에 NAT 게이트웨이 배포

프라이빗 서브넷(IPv4 및 IPv6)에 노드를 배포하는 경우 각 가용 영역(AZ)에 NAT 게이트웨이를 생성하여 영역 독립적인 아키텍처를 보장하고 교차 AZ 지출을 줄이는 것이 좋습니다. AZ의 각 NAT 게이트웨이는 중복성을 통해 구현됩니다.

Amazon VPC CNI

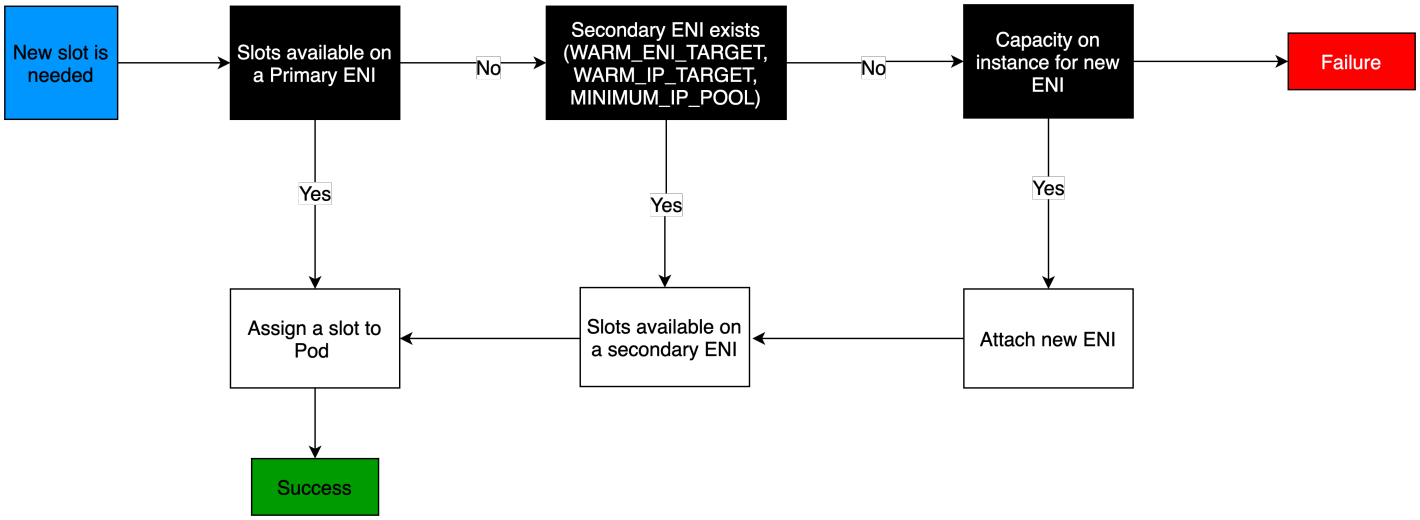
Amazon EKS는 VPC CNI라고도 하는 Amazon VPC 컨테이너 네트워크 인터페이스 플러그인을 통해 클러스터 네트워킹을 구현합니다. CNI 플러그인을 사용하면 Kubernetes 포드가 VPC 네트워크에서와 동일한 IP 주소를 가질 수 있습니다. 특히 포드 내의 모든 컨테이너는 네트워크 네임스페이스를 공유하며 로컬 포트를 사용하여 서로 통신할 수 있습니다.

Amazon VPC CNI에는 두 가지 구성 요소가 있습니다.

- 포드 간 통신을 활성화Pod-to-Pod 네트워크를 설정하는 CNI 바이너리입니다. CNI 바이너리는 노드 루트 파일 시스템에서 실행되며 새 포드가에 추가되거나 노드에서 기존 포드가 제거될 때 kubelet에 의해 호출됩니다.
- 장기 실행 노드-로컬 IP 주소 관리(IPAM) 데몬인 ipamd는 다음을 담당합니다.
  - 노드에서 ENIs 관리
  - 사용 가능한 IP 주소 또는 접두사의 웜 풀 유지 관리

인스턴스가 생성되면 EC2는 기본 서브넷과 연결된 기본 ENI를 생성하고 연결합니다. 기본 서브넷은 퍼블릭 또는 프라이빗일 수 있습니다. hostNetwork 모드에서 실행되는 포드는 노드 기본 ENI에 할당된 기본 IP 주소를 사용하고 호스트와 동일한 네트워크 네임스페이스를 공유합니다.

CNI 플러그인은 노드에서 탄력적 네트워크 인터페이스(ENI)를 관리합니다. 노드가 프로비저닝되면 CNI 플러그인은 노드의 서브넷에서 기본 ENI로 슬롯 풀(IPs 또는 접두사)을 자동으로 할당합니다. 이 풀을 웜 풀이라고 하며, 크기는 노드의 인스턴스 유형에 따라 결정됩니다. CNI 설정에 따라 슬롯은 IP 주소 또는 접두사일 수 있습니다. ENI의 슬롯이 할당되면 CNI는 슬롯의 웜 풀이 있는 추가 ENIs를 노드에 연결할 수 있습니다. 이러한 추가 ENIs 보조 ENIs. 각 ENI는 인스턴스 유형에 따라 특정 수의 슬롯만 지원할 수 있습니다. CNI는 일반적으로 포드 수에 해당하는 필요한 슬롯 수를 기반으로 더 많은 ENIs를 인스턴스에 연결합니다. 이 프로세스는 노드가 더 이상 추가 ENI를 지원할 수 없을 때까지 계속됩니다. 또한 CNI는 더 빠른 포드 시작을 위해 "웜" ENIs 및 슬롯을 사전 할당합니다. 각 인스턴스 유형에는 연결할 수 있는 최대 ENIs 수가 있습니다. 이는 컴퓨팅 리소스 외에도 포드 밀도(노드당 포드 수)에 대한 제약 조건 중 하나입니다.



최대 네트워크 인터페이스 수와 사용할 수 있는 최대 슬롯 수는 EC2 인스턴스 유형에 따라 다릅니다. 각 포드는 슬롯의 IP 주소를 사용하기 때문에 특정 EC2 인스턴스에서 실행할 수 있는 포드 수는 연결할 수 있는 ENIs 수와 각 ENI가 지원하는 슬롯 수에 따라 달라집니다. 인스턴스의 CPU 및 메모리

리소스가 고갈되지 않도록 EKS당 최대 포드 사용 설명서를 설정하는 것이 좋습니다. 를 사용하는 포드hostNetwork는이 계산에서 제외됩니다. max-pod-calculator.sh 스크립트를 사용하여 지정된 인스턴스 유형에 대한 EKS의 권장 최대 포드를 계산하는 것이 좋습니다.

개요

보조 IP 모드는 VPC CNI의 기본 모드입니다. 이 가이드에서는 보조 IP 모드가 활성화된 경우 VPC CNI 동작에 대한 일반적인 개요를 제공합니다. ipamd(IP 주소 할당)의 기능은 , the section called “Linux용 접두사 모드” the section called “포드당 보안 그룹”및와 같은 VPC CNI의 구성 설정에 따라 달라질 수 있습니다the section called “사용자 지정 네트워킹”.

Amazon VPC CNI는 작업자 노드에 aws-node라는 Kubernetes Daemonset으로 배포됩니다. 작업자 노드가 프로비저닝되면 기본 ENI라고 하는 기본 ENI가 노드에 연결됩니다. CNI는 노드의 기본 ENIs에 연결된 서브넷에서 ENI 및 보조 IP 주소의 웜 풀을 할당합니다. 기본적으로 ipamd는 노드에 추가 ENI를 할당하려고 시도합니다. IPAMD는 단일 포드가 예약되고 기본 ENI의 보조 IP 주소가 할당될 때 추가 ENI를 할당합니다. 이 "웜" ENI를 사용하면 더 빠른 포드 네트워킹이 가능합니다. 보조 IP 주소 풀이 부족해지면 CNI는 다른 ENI를 추가하여 더 할당합니다.

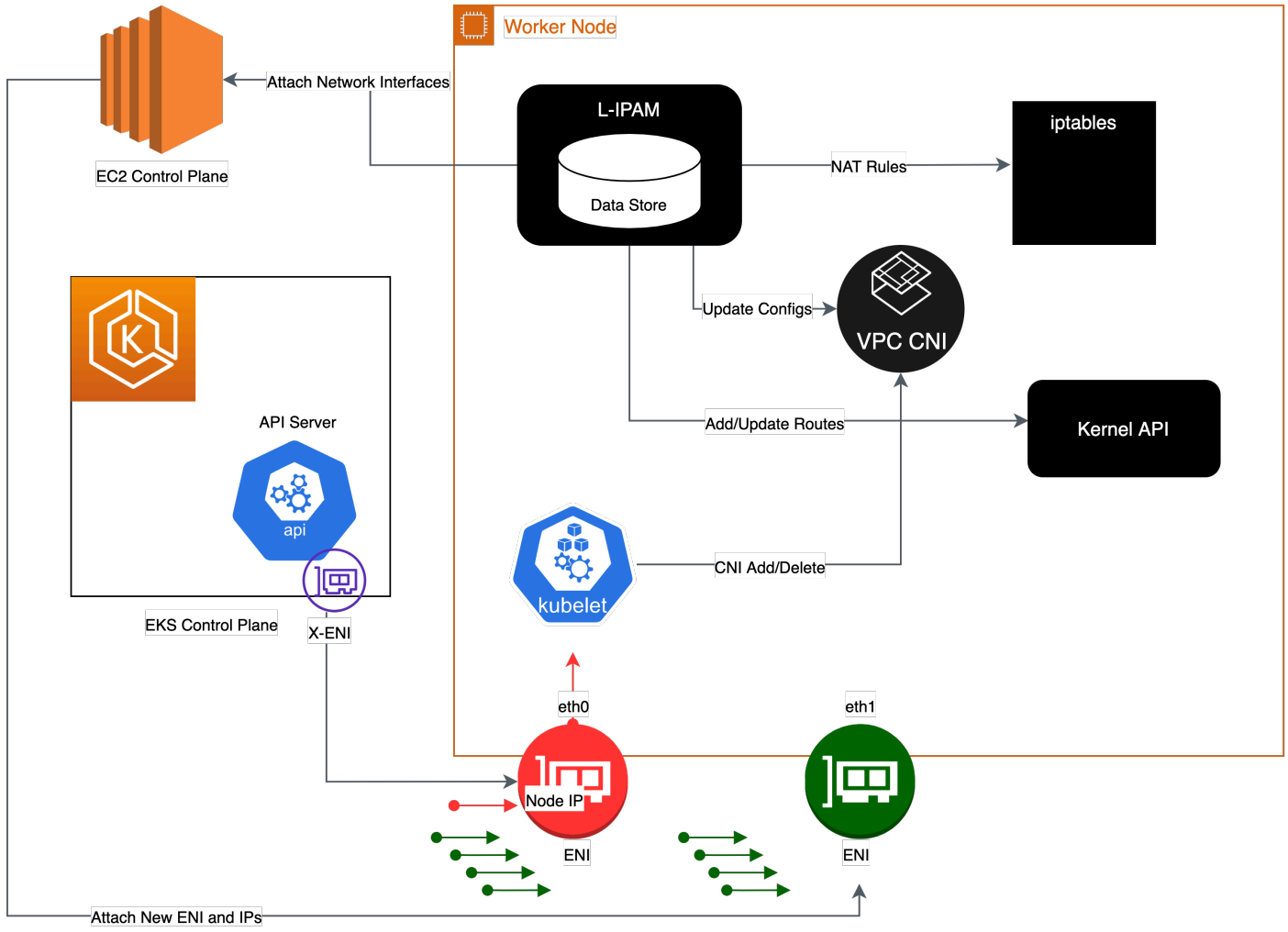
풀의 ENIs 및 IP 주소 수는 WARM_ENI_TARGET, WARM_IP_TARGET, MINIMUM_IP_TARGET이라는 환경 변수를 통해 구성됩니다. aws-node Daemonset은 충분한 수의 ENIs가 연결되어 있는지 주기적으로 확인합니다. WARM_ENI_TARGET, 또는 WARM_IP_TARGET 및 MINIMUM_IP_TARGET 조건이 모두 충족되면 충분한 수의 ENIs가 연결됩니다. 연결된 ENIs가 충분하지 않은 경우 CNI는 MAX_ENI 한도에 도달할 때까지 EC2에 API를 호출하여 더 많이 연결합니다.

- WARM_ENI_TARGET - 정수, 값이 0보다 크면 요구 사항이 활성화되었음을 나타냄
  - 유지할 웜 ENIs. ENI는 노드에 보조 ENI로 연결될 때 "웜"이지만 포드에서는 사용되지 않습니다. 보다 구체적으로 말하자면, ENI의 IP 주소는 포드와 연결되지 않았습니다.
  - 예: 각 ENIs가 5개의 IP 주소를 지원하는 2개의 ENI가 있는 인스턴스를 가정합니다. WARM_ENI_TARGET은 1로 설정됩니다. 정확히 5개의 IP 주소가 인스턴스와 연결된 경우 CNI는 인스턴스에 연결된 2ENIs를 유지합니다. 첫 번째 ENI가 사용 중이며이 ENI의 가능한 IP 주소 5개가 모두 사용됩니다. 두 번째 ENI는 풀에 5개의 IP 주소가 모두 있는 "웜"입니다. 인스턴스에서 다른 포드가 시작되면 6번째 IP 주소가 필요합니다. CNI는이 6번째 포드에 두 번째 ENI의 IP 주소와 풀의 5개의 IPs 주소를 할당합니다. 이제 두 번째 ENI가 사용 중이며 더 이상 "웜" 상태가 아닙니다. CNI는 3번째 ENI를 할당하여 최소 1개의 웜 ENI를 유지합니다.

> **Note**
>
> 웜 ENIs는 여전히 VPC의 CIDR에서 IP 주소를 사용합니다. IP 주소는 포드와 같은 워크로드와 연결될 때까지 "미사용" 또는 "웜"입니다.

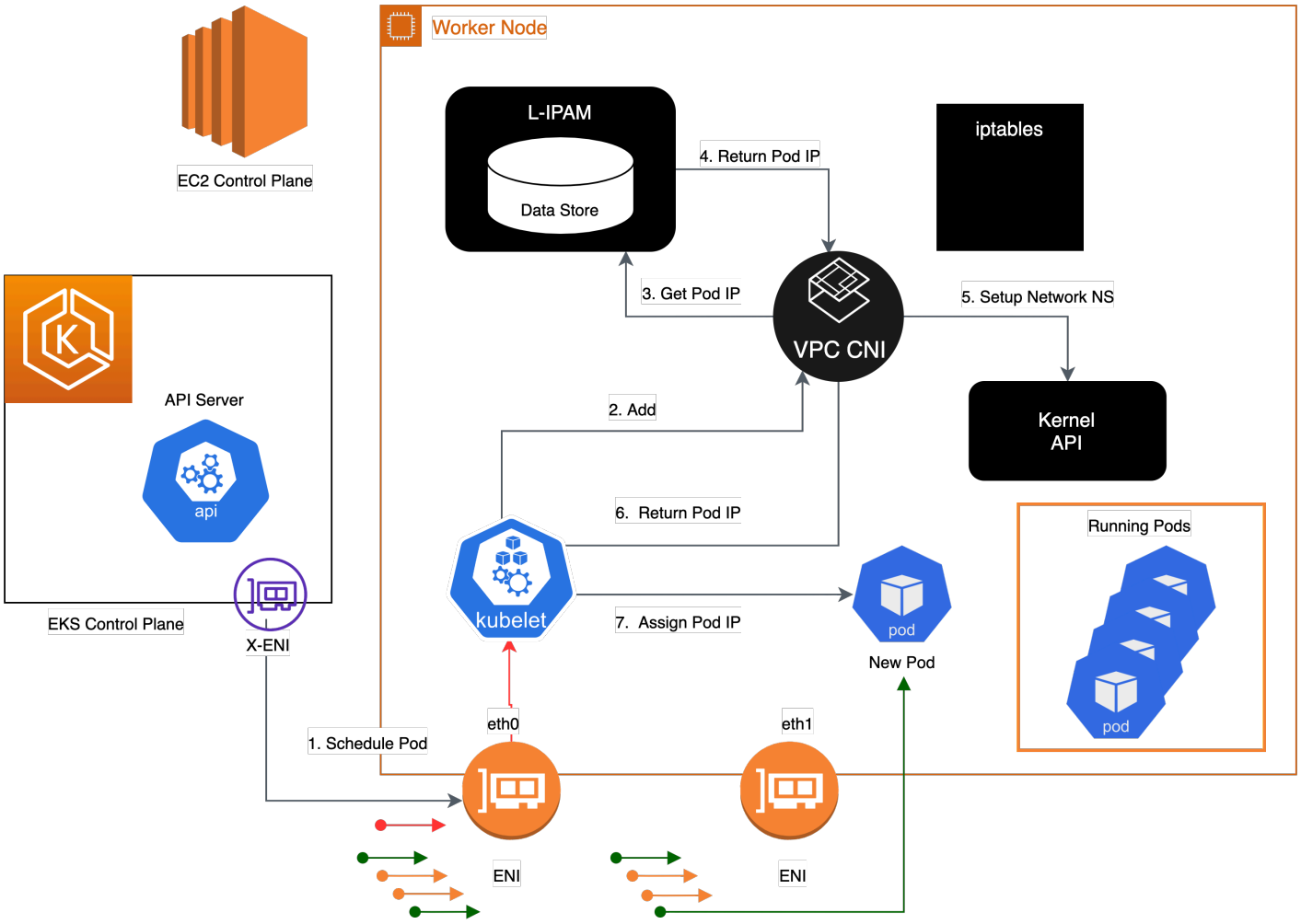
- `WARM_IP_TARGET`, 정수, 값이 0보다 크면 요구 사항이 활성화되었음을 나타냅니다.
  - 유지할 웜 IP 주소 수입니다. 웜 IP는 활발하게 연결된 ENI에서 사용할 수 있지만 포드에 할당되지 않았습니다. 즉, 사용 가능한 웜 IPs 수는 추가 ENI 없이 포드에 할당할 수 있는 IPs 수입니다.
- 예: 각 ENI가 20개의 IP 주소를 지원하는 1개의 ENI가 있는 인스턴스를 가정합니다. WARM_IP_TARGET은 5로 설정됩니다. WARM_ENI_TARGET은 0으로 설정됩니다. 16번째 IP 주소가 필요할 때까지 1개의 ENI만 연결됩니다. 그런 다음 CNI는 서브넷 CIDR에서 가능한 주소 20개를 사용하여 두 번째 ENI를 연결합니다.
- `MINIMUM_IP_TARGET`, 정수, 값이 0보다 크면 요구 사항이 활성화되었음을 나타냅니다.
  - 언제든지 할당할 최소 IP 주소 수입니다. 이는 일반적으로 인스턴스 시작 시 여러 ENIs의 할당을 미리 로드하는 데 사용됩니다.
  - 예: 새로 시작된 인스턴스를 고려합니다. 여기에는 1개의 ENI가 있으며 각 ENI는 10개의 IP 주소를 지원합니다. MINIMUM_IP_TARGET은 100으로 설정됩니다. ENI는 총 100개의 주소에 대해 9개의 추가 ENIs를 즉시 연결합니다. 이는 WARM_IP_TARGET 또는 WARM_ENI_TARGET 값에 관계없이 발생합니다.

이 프로젝트에는 서브넷 계산기 Excel 문서가 포함되어 있습니다. 이 계산기 문서는 `WARM_IP_TARGET` 및와 같은 다양한 ENI 구성 옵션에서 지정된 워크로드의 IP 주소 소비를 시뮬레이션합니다`WARM_ENI_TARGET`.

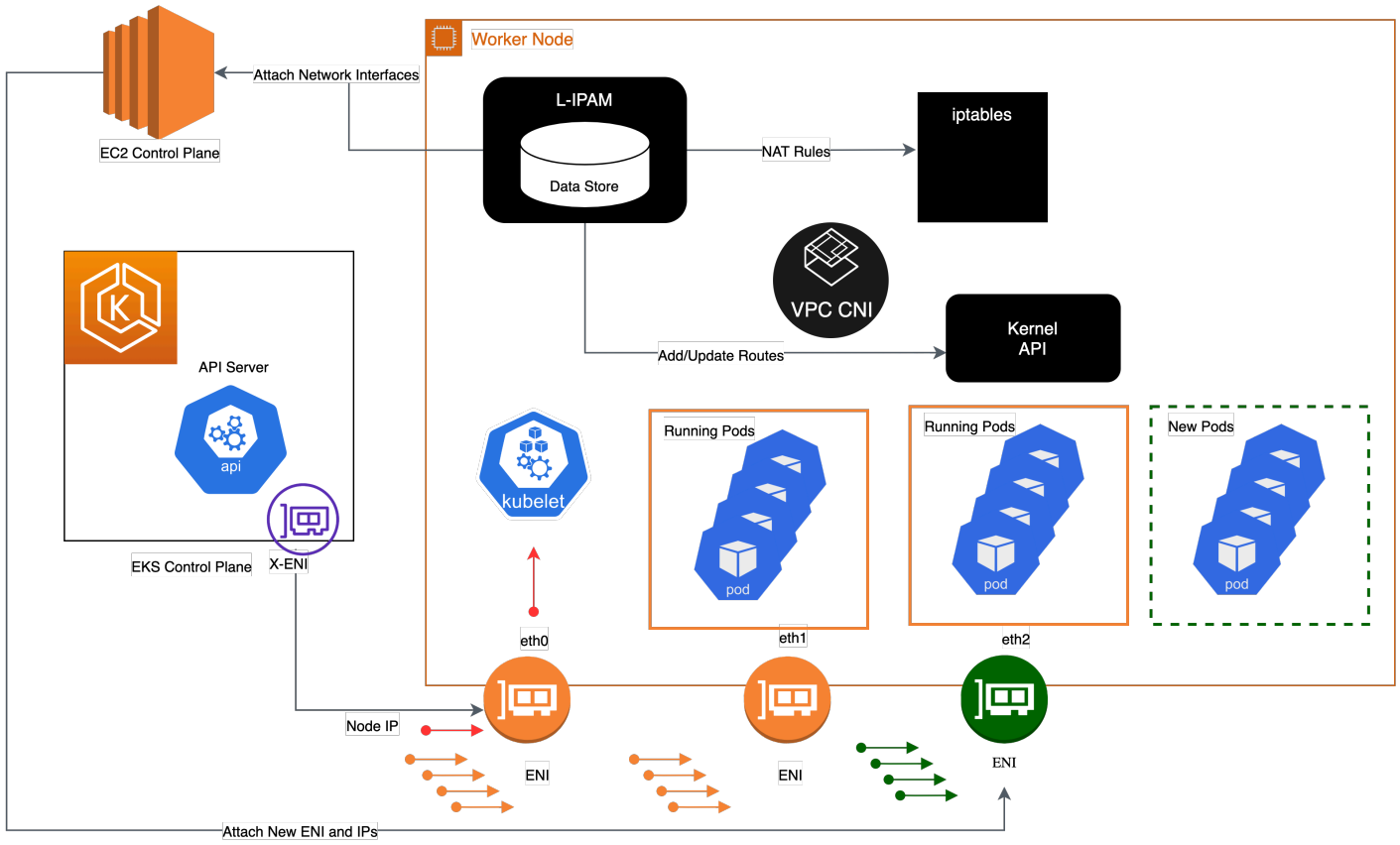


Kubelet이 포드 추가 요청을 수신하면 CNI 바이너리가 ipamd에 사용 가능한 IP 주소를 쿼리한 다음 ipamd가 포드에 제공합니다. CNI 바이너리가 호스트 및 포드 네트워크를 연결합니다.

노드에 배포된 포드는 기본적으로 기본 ENI와 동일한 보안 그룹에 할당됩니다. 또는 포드를 다른 보안 그룹으로 구성할 수 있습니다.



IP 주소 풀이 고갈되면 플러그 인은 다른 탄력적 네트워크 인터페이스를 인스턴스에 자동으로 연결하고 이 인터페이스에 다른 보조 IP 주소 집합을 할당합니다. 이 프로세스는 노드가 탄력적 네트워크 인터페이스를 추가 지원할 수 없을 때까지 계속됩니다.



포드가 삭제되면 VPC CNI는 포드의 IP 주소를 30초 휴지 캐시에 배치합니다. 휴지 캐시IPs는 새 포드에 할당되지 않습니다. 쿨링 오프 기간이 끝나면 VPC CNI는 포드 IP를 웜 풀로 다시 이동합니다. 쿨링 오프 기간은 포드 IP 주소가 조기에 재활용되는 것을 방지하고 모든 클러스터 노드에서 kube-proxy가 iptables 규칙 업데이트를 완료하도록 허용합니다. IPs 또는 ENIs 수가 웜 풀 설정 수를 초과하면 ipamd 플러그인은 IPs 및 ENIs에 반환합니다.

위에서 보조 IP 모드에서 설명한 대로 각 포드는 인스턴스에 연결된 ENIs 중 하나에서 하나의 보조 프라이빗 IP 주소를 수신합니다. 각 포드는 IP 주소를 사용하므로 특정 EC2 인스턴스에서 실행할 수 있는 포드 수는 연결할 수 있는 ENIs 수와 지원하는 IP 주소 수에 따라 달라집니다. VPC CNI는 제한 파일을 확인하여 각 인스턴스 유형에 허용되는 ENIs 및 IP 주소 수를 확인합니다.

다음 공식을 사용하여 노드에 배포할 수 있는 최대 포드 수를 결정할 수 있습니다.

```
(Number of network interfaces for the instance type * (the number of IP addresses per
 network interface - 1)) + 2
```

+2는 kube-proxy 및 VPC CNI와 같은 호스트 네트워킹이 필요한 포드를 나타냅니다. Amazon EKS에서는 kube-proxy 및 VPC CNI가 각 노드에서 작동해야 하며 이러한 요구 사항은 max-pods 값에 반영됩

니다. 추가 호스트 네트워킹 포드를 실행하려면 최대 포드 값을 업데이트하는 것이 좋습니다. 시작 템플릿에서를 사용자 데이터`--kubelet-extra-args "–max-pods=110"`로 지정할 수 있습니다.

예를 들어 c5.large 노드가 3개인 클러스터(ENIs당 최대 10IPs)에서 클러스터가 시작되고 CoreDNS 포드가 2개 있는 경우 CNI는 IP 주소 50개를 사용하고 웜 풀에 IPs 43개를 유지합니다. 웜 풀을 사용하면 애플리케이션이 배포될 때 포드를 더 빠르게 시작할 수 있습니다.

노드 1(CoreDNS 포드 포함): ENIs, IPs 할당

노드 2(CoreDNS 포드 포함): ENIs, IPs 할당

노드 3(포드 없음): ENI 1개. 10개의 IPs 할당되었습니다.

노드 1 및 노드 2(동일한 구성)의 경우:

- ENIs당 IPs 10개 = 총 IPs 20개
- 기본 IPs(ENI당 1개)를 빼기 = IPs
- CoreDNS 포드의 IP 1을 빼기 = 사용 가능한 IPs 17개
- 따라서 이러한 각 노드에는 웜 풀에 17IPs가 있습니다.

노드 3의 경우:

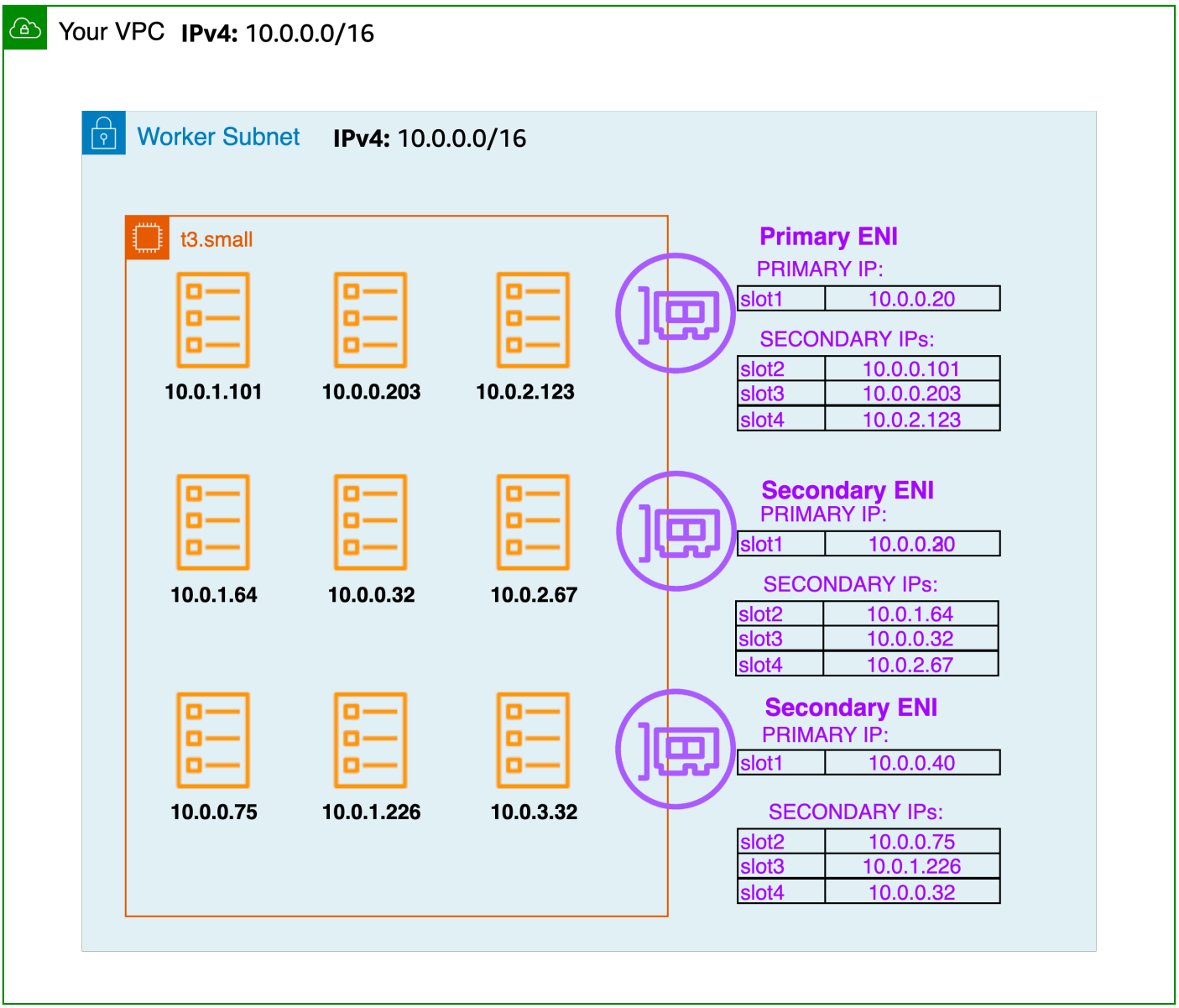
- ENI 1개 × IPs = 총 IPs 10개
- 웜 풀에서 사용 가능한 기본 IP 1개 = IPs

총 웜 풀 계산: - 17(노드 1) + 17(노드 2) + 9(노드 3) = 43 IPs

종종 데몬 세트로 실행되는 인프라 포드는 각각 최대 포드 수에 기여한다는 점에 유의하세요. 여기에는 다음이 포함될 수 있습니다.

- CoreDNS
- Amazon Elastic LoadBalancer
- 지표 서버의 운영 포드

이러한 포드의 용량을 결합하여 인프라를 계획하는 것이 좋습니다. 각 인스턴스 유형에서 지원하는 최대 포드 수 목록은 GitHub의 eni-max-Pods.txt를 참조하세요.



권장 사항

Auto Mode를 사용하여 EKS 클러스터 배포

EKS Auto Mode를 사용하여 클러스터를 생성하면 AWS가 클러스터의 VPC 컨테이너 네트워크 인터페이스(CNI) 구성을 관리합니다. Amazon EKS Auto Mode를 사용하면 네트워킹 추가 기능을 설치하거나 업그레이드할 필요가 없습니다. 그러나 워크로드가 관리형 VPC CNI 구성과 호환되는지 확인합니다.

VPC CNI 관리형 추가 기능 배포

클러스터를 프로비저닝하면 Amazon EKS는 VPC CNI를 자동으로 설치합니다. 그럼에도 불구하고 Amazon EKS는 클러스터가 컴퓨팅, 스토리지 및 네트워킹과 같은 기본 AWS 리소스와 상호 작용할 수 있도록 하는 관리형 추가 기능을 지원합니다. VPC CNI를 포함한 관리형 추가 기능을 사용하여 클러스터를 배포하는 것이 좋습니다.

Amazon EKS 관리형 추가 기능은 Amazon EKS 클러스터에 VPC CNI 설치 및 관리를 제공합니다. Amazon EKS 추가 기능에는 최신 보안 패치, 버그 수정이 포함되며 Amazon EKS에서 작동하도록 AWS에서 검증했습니다. VPC CNI 추가 기능을 사용하면 Amazon EKS 클러스터의 보안 및 안정성을 지속적으로 보장하고 추가 기능을 설치, 구성 및 업데이트하는 데 필요한 노력을 줄일 수 있습니다. 또한 Amazon EKS API, AWS Management Console, AWS CLI 및 eksctl을 통해 관리형 추가 기능을 추가, 업데이트 또는 삭제할 수 있습니다.

`kubectl get` 명령과 함께 `--show-managed-fields` 플래그를 사용하여 VPC CNI의 관리형 필드를 찾을 수 있습니다.

```
kubectl get daemonset aws-node --show-managed-fields -n kube-system -o yaml
```

관리형 추가 기능은 15분마다 구성을 자동으로 덮어써 구성 드리프트를 방지합니다. 즉, 추가 기능 생성 후 Kubernetes API를 통해 관리형 추가 기능을 변경하면 자동 드리프트 방지 프로세스가 덮어쓰고 추가 기능 업데이트 프로세스 중에 기본값으로 설정됩니다.

EKS에서 관리하는 필드는 관리자가 EKS인 managedFields 아래에 나열됩니다. EKS에서 관리하는 필드에는 서비스 계정, 이미지, 이미지 URL, 라이브니스 프로브, 준비 프로브, 레이블, 볼륨 및 볼륨 마운트가 포함됩니다.

> **Note**
>
> WARM_ENI_TARGET, WARM_IP_TARGET, MINIMUM_IP_TARGET과 같이 가장 자주 사용되는 필드는 관리되지 않으며 조정되지 않습니다. 이러한 필드의 변경 사항은 추가 기능 업데이트 시 유지됩니다.

프로덕션 클러스터를 업데이트하기 전에 특정 구성에 대해 비프로덕션 클러스터의 추가 기능 동작을 테스트하는 것이 좋습니다. 또한 추가 기능 구성에 대한 EKS 사용 설명서의 단계를 따릅니다.

관리형 추가 기능으로 마이그레이션

버전 호환성을 관리하고 자체 관리형 VPC CNI의 보안 패치를 업데이트합니다. 자체 관리형 추가 기능을 업데이트하려면 EKS 사용 설명서에 설명된 Kubernetes APIs 및 지침을 사용해야 합니다. 기존 EKS 클러스터의 관리형 추가 기능으로 마이그레이션하는 것이 좋습니다. 마이그레이션 전에 현재 CNI 설정의 백업을 생성하는 것이 좋습니다. 관리형 추가 기능을 구성하려면 Amazon EKS API, AWS Management Console 또는 AWS 명령줄 인터페이스를 활용할 수 있습니다.

```
kubectl apply view-last-applied daemonset aws-node -n kube-system  aws-k8s-cni-old.yaml
```

필드가 기본 설정으로 관리형으로 나열된 경우 Amazon EKS는 CNI 구성 설정을 대체합니다. 관리형 필드를 수정하지 않도록 주의합니다. 추가 기능은 웜 환경 변수 및 CNI 모드와 같은 구성 필드를 조정하지 않습니다. 관리형 CNI로 마이그레이션하는 동안 포드와 애플리케이션은 계속 실행됩니다.

업데이트 전 CNI 설정 백업

VPC CNI는 고객 데이터 영역(노드)에서 실행되므로 Amazon EKS는 새 버전이 릴리스되거나 클러스터를 새 Kubernetes 마이너 버전으로 업데이트한 후 추가 기능(관리형 및 자체 관리형)을 자동으로 업데이트하지 않습니다. 기존 클러스터의 추가 기능을 업데이트하려면 update-addon API를 통해 업데이트를 트리거하거나 추가 기능을 위해 EKS 콘솔에서 지금 업데이트 링크를 클릭해야 합니다. 자체 관리형 추가 기능을 배포한 경우 자체 관리형 VPC CNI 추가 기능 업데이트에 언급된 단계를 따릅니다.

한 번에 하나의 마이너 버전을 업데이트하는 것이 좋습니다. 예를 들어 현재 버전이 `1.9`이고 `1.11`으로 업데이트하려는 경우 먼저 최신 패치 버전 `1.10`로 업데이트한 다음 최신 패치 버전 `1.11`으로 업데이트해야 합니다.

Amazon VPC CNI를 업데이트하기 전에 aws-node Daemonset 검사를 수행합니다. 기존 설정을 백업합니다. 관리형 추가 기능을 사용하는 경우 Amazon EKS가 재정의할 수 있는 설정을 업데이트하지 않았는지 확인합니다. 자동화 워크플로에서 사후 업데이트 후크를 사용하거나 추가 기능 업데이트 후 수동 적용 단계를 사용하는 것이 좋습니다.

```
kubectl apply view-last-applied daemonset aws-node -n kube-system  aws-k8s-cni-old.yaml
```

자체 관리형 추가 기능의 경우 `releases` GitHub의와 백업을 비교하여 사용 가능한 버전을 확인하고 업데이트하려는 버전의 변경 사항을 숙지합니다. Helm을 사용하여 자체 관리형 추가 기능을 관리하고 값 파일을 활용하여 설정을 적용하는 것이 좋습니다. Daemonset 삭제와 관련된 모든 업데이트 작업은 애플리케이션 가동 중지를 초래하므로 피해야 합니다.

보안 컨텍스트 이해

VPC CNI를 효율적으로 관리하기 위해 구성된 보안 컨텍스트를 이해하는 것이 좋습니다. Amazon VPC CNI에는 두 가지 구성 요소인 CNI 바이너리와 ipamd(aws-node) Daemonset이 있습니다. CNI는 노드에서 바이너리로 실행되고 노드 루트 파일 시스템에 액세스할 수 있으며 노드 수준에서 iptable을 처리할 때 권한이 있는 액세스 권한도 가집니다. 포드가 추가되거나 제거되면 kubelet에서 CNI 바이너리를 호출합니다.

aws-node Daemonset은 노드 수준에서 IP 주소 관리를 담당하는 장기 실행 프로세스입니다. aws-node는 `hostNetwork` 모드에서 실행되며 루프백 디바이스에 대한 액세스와 동일한 노드에 있는 다른 포드의 네트워크 활동을 허용합니다. aws-node init-container는 권한 있는 모드에서 실행되고 CRI 소켓을 탑재하여 Daemonset이 노드에서 실행되는 포드의 IP 사용량을 모니터링할 수 있도록 합니다. Amazon EKS는 aws-node init 컨테이너의 권한 있는 요구 사항을 제거하기 위해 노력하고 있습니다. 또한 aws-node는 NAT 항목을 업데이트하고 iptables 모듈을 로드해야 하므로 NET_ADMIN 권한으로 실행됩니다.

Amazon EKS는 포드 및 네트워킹 설정에 대한 IP 관리를 위해 aws-node 매니페스트에 정의된 보안 정책을 배포할 것을 권장합니다. 최신 버전의 VPC CNI로 업데이트하는 것이 좋습니다. 또한 특정 보안 요구 사항이 있는 경우 GitHub 문제를 여는 것이 좋습니다.

CNI에 별도의 IAM 역할 사용

AWS VPC CNI에는 AWS Identity and Access Management(IAM) 권한이 필요합니다. IAM 역할을 사용하려면 먼저 CNI 정책을 설정해야 합니다. IPv4 클러스터에 대한 AWS 관리형 정책`AmazonEKS_CNI_Policy`인를 사용할 수 있습니다. AmazonEKS CNI 관리형 정책에는 IPv4 클러스터에 대한 권한만 있습니다. 여기에 나열된 권한을 사용하여 IPv6 클러스터에 대한 별도의 IAM 정책을 생성해야 합니다.

기본적으로 VPC CNI는 Amazon EKS 노드 IAM 역할(관리형 노드 그룹과 자체 관리형 노드 그룹 모두)을 상속합니다.

Amazon VPC CNI에 대한 관련 정책을 사용하여 별도의 IAM 역할을 구성하는 것이 좋습니다. 그렇지 않으면 Amazon VPC CNI의 포드는 노드 IAM 역할에 할당된 권한을 얻고 노드에 할당된 인스턴스 프로파일에 액세스할 수 있습니다.

VPC CNI 플러그인은 aws-node라는 서비스 계정을 생성하고 구성합니다. 기본적으로 서비스 계정은 Amazon EKS CNI 정책이 연결된 Amazon EKS 노드 IAM 역할에 바인딩됩니다. 별도의 IAM 역할을 사용하려면 Amazon EKS CNI 정책이 연결된 새 서비스 계정을 생성하는 것이 좋습니다. 새 서비스 계정을 사용하려면 CNI 포드를 재배포해야 합니다. 새 클러스터를 생성할 때 VPC CNI 관리형 추가 기능에

`--service-account-role-arn`를 지정하는 것이 좋습니다. Amazon EKS 노드 역할에서 IPv4 및 IPv6 모두에 대한 Amazon EKS CNI 정책을 제거해야 합니다.

보안 위반의 폭발 반경을 최소화하려면 액세스 인스턴스 메타데이터를 차단하는 것이 좋습니다.

Liveness/Readiness 프로브 실패 처리

프로브 실패로 인해 애플리케이션의 포드가 containerCreating 상태에서 멈추지 않도록 EKS 1.20 이상 클러스터의 라이브니스 및 준비 프로브 제한 시간 값(기본값 `timeoutSeconds: 10`)을 늘리는 것이 좋습니다. 이 문제는 데이터 집약적 및 배치 처리 클러스터에서 발견되었습니다. CPU 사용량이 높으면 aws-node 프로브 상태가 실패하여 채워지지 않은 Pod CPU 요청이 발생합니다. 프로브 제한 시간을 수정하는 것 외에도 aws-node에 대한 CPU 리소스 요청(기본값 `CPU: 25m`)이 올바르게 구성되었는지 확인합니다. 노드에 문제가 없는 한 설정을 업데이트하지 않는 것이 좋습니다.

Amazon EKS 지원에 참여하는 동안 노드`bash /opt/cni/bin/aws-cni-support.sh`에서 sudo를 실행하는 것이 좋습니다. 스크립트는 노드에서 kubelet 로그 및 메모리 사용률을 평가하는 데 도움이 됩니다. 스크립트를 실행하려면 Amazon EKS 작업자 노드에 SSM 에이전트를 설치하는 것이 좋습니다.

비EKS 최적화 AMI 인스턴스에서 IPTables 전달 정책 구성

사용자 지정 AMI를 사용하는 경우 kubelet.service에서 iptables 전달 정책을 ACCEPT로 설정해야 합니다. 많은 시스템에서 iptables 전달 정책을 DROP로 설정합니다. HashiCorp Packer와 AWS GitHub의 Amazon EKS AMI 리포지토리에서 리소스 및 구성 스크립트를 사용하여 빌드 사양을 사용하여 사용자 지정 AMI를 빌드할 수 있습니다. kubelet.service를 업데이트하고 여기에 지정된 지침에 따라 사용자 지정 AMI를 생성할 수 있습니다.

정기적으로 CNI 버전 업그레이드

VPC CNI는 이전 버전과 호환됩니다. 최신 버전은 Amazon EKS가 지원하는 모든 Kubernetes 버전에서 작동합니다. 또한 VPC CNI는 EKS 추가 기능으로 제공됩니다(위의 "VPC CNI 관리형 추가 기능 배포" 참조). EKS 추가 기능은 추가 기능의 업그레이드를 오케스트레이션하지만 데이터 영역에서 실행되므로 CNI와 같은 추가 기능을 자동으로 업그레이드하지 않습니다. 관리형 및 자체 관리형 작업자 노드 업그레이드 후 VPC CNI 추가 기능을 업그레이드하는 것은 사용자의 책임입니다.

IP 주소 사용률 최적화

컨테이너화된 환경은 애플리케이션 현대화 덕분에 빠른 속도로 확장되고 있습니다. 즉, 점점 더 많은 작업자 노드와 포드가 배포되고 있습니다.

Amazon VPC CNI 플러그인은 각 포드에 VPC의 CIDR(들)의 IP 주소를 할당합니다. 이 접근 방식은 VPC 흐름 로그 및 기타 모니터링 솔루션과 같은 도구를 사용하여 포드 주소에 대한 전체 가시성을 제공합니다. 워크로드 유형에 따라 포드에서 많은 수의 IP 주소를 사용할 수 있습니다.

AWS 네트워킹 아키텍처를 설계할 때는 VPC 및 노드 수준에서 Amazon EKS IP 소비를 최적화하는 것이 중요합니다. 이렇게 하면 IP 소진 문제를 완화하고 노드당 포드 밀도를 높일 수 있습니다.

이 섹션에서는 이러한 목표를 달성하는 데 도움이 되는 기법에 대해 설명합니다.

노드 수준 IP 소비 최적화

접두사 위임은 Amazon Virtual Private Cloud(Amazon VPC)의 기능으로, Amazon Elastic Compute Cloud(Amazon EC2) 인스턴스에 IPv4 또는 IPv6 접두사를 할당할 수 있습니다.Amazon EC2 네트워크 인터페이스(ENI)당 IP 주소를 늘려 노드당 포드 밀도를 높이고 컴퓨팅 효율성을 개선합니다. 접두사 위임은 사용자 지정 네트워킹에서도 지원됩니다.

자세한 내용은 Linux 노드를 사용한 접두사 위임 및 Windows 노드를 사용한 접두사 위임 섹션을 참조하세요.

IP 소모 완화

클러스터가 사용 가능한 모든 IP 주소를 사용하지 않도록 하려면 확장을 염두에 두고 VPCs 및 서브넷의 크기를 조정하는 것이 좋습니다.

IPv6를 채택하는 것은 처음부터 이러한 문제를 방지하는 좋은 방법입니다. 그러나 확장성 요구 사항이 초기 계획을 초과하고 IPv6를 채택할 수 없는 조직의 경우 IP 주소 소진에 대한 권장 대응은 VPC 설계를 개선하는 것입니다. Amazon EKS 고객에게 가장 일반적으로 사용되는 기법은 라우팅할 수 없는 보조 CIDRs VPC에 추가하고 IP 주소를 포드에 할당할 때이 추가 IP 공간을 사용하도록 VPC CNI를 구성하는 것입니다. 이를 일반적으로 사용자 지정 네트워킹이라고 합니다.

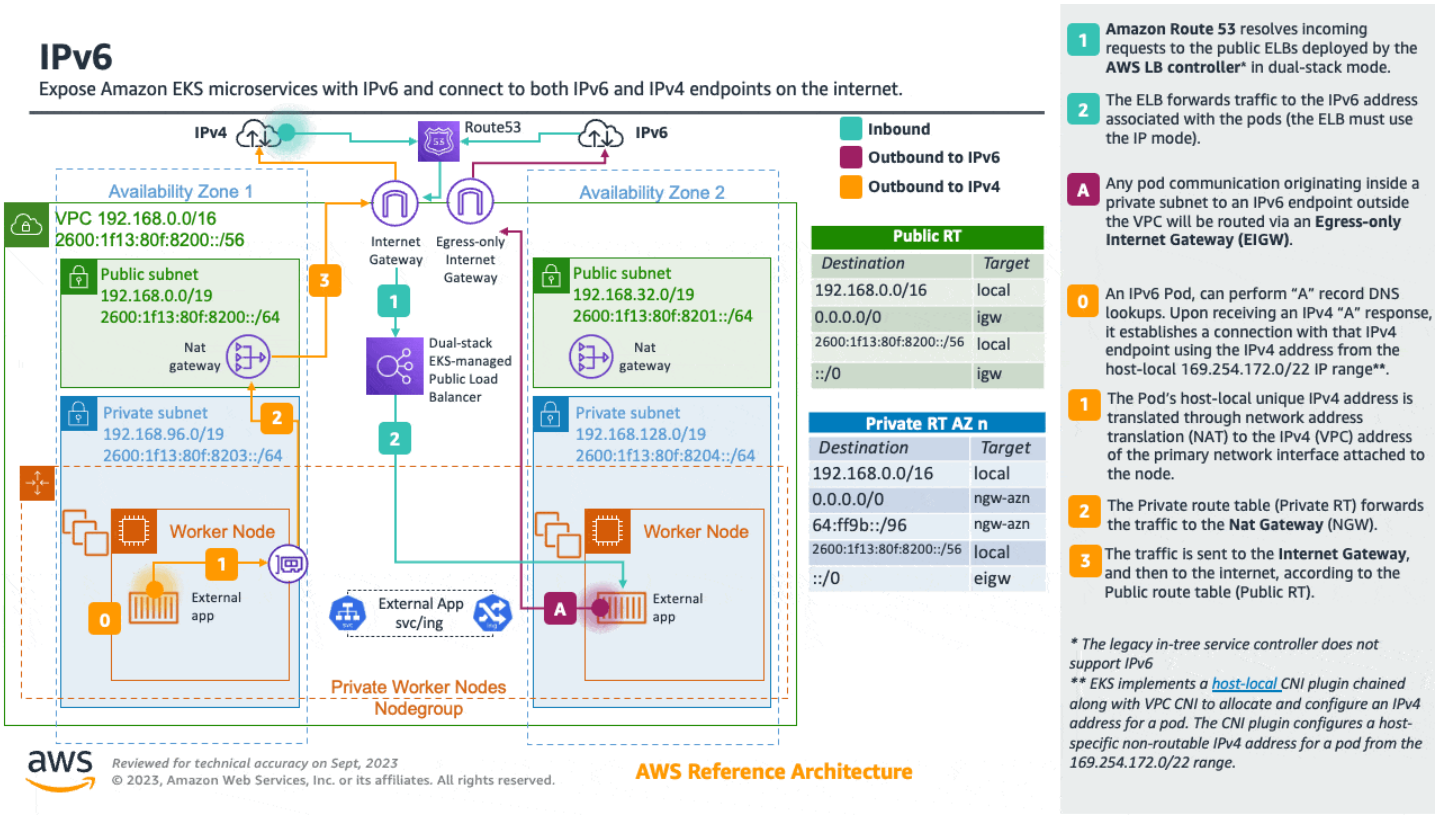
노드에 할당된 IPs 웜 IP 풀을 최적화하는 데 사용할 수 있는 Amazon VPC CNI의 변수를 다룹니다. Amazon EKS에는 내장되어 있지 않지만 IP 소진을 완화하는 데 도움이 될 수 있는 다른 아키텍처 패턴으로이 섹션을 닫습니다.

IPv6 사용(권장)

IPv6 채택은 RFC1918 제한 사항을 해결하는 가장 쉬운 방법입니다. 네트워크 아키텍처를 선택할 때 IPv6를 첫 번째 옵션으로 채택하는 것이 좋습니다. IPv6는 훨씬 더 큰 총 IP 주소 공간을 제공하며 클러스터 관리자는 IPv4 제한을 해결하기 위한 노력 없이 애플리케이션을 마이그레이션하고 확장하는 데 집중할 수 있습니다.

Amazon EKS 클러스터는 IPv4와 IPv6를 모두 지원합니다. 기본적으로 EKS 클러스터는 IPv4 주소 공간을 사용합니다. 클러스터 생성 시 IPv6 기반 주소 공간을 지정하면 IPv6를 사용할 수 있습니다. IPv6 EKS 클러스터에서 포드 및 서비스는 레거시 IPv4 엔드포인트가 IPv6 클러스터에서 실행되는 서비스에 연결하는 기능을 유지하면서 IPv6 주소를 수신하고 그 반대의 경우도 마찬가지입니다IPv4. 클러스터 내의 모든 pod-to-pod 통신은 항상 IPv6를 통해 이루어집니다. VPC(/56) 내에서 IPv6 서브넷의 IPv6 CIDR 블록 크기는 /64로 고정됩니다. 이렇게 하면가 EKS에서 배포를 확장할 수 있는 2^64(약 180억) IPv6 주소가 제공됩니다.

자세한 내용은 IPv6 EKS 클러스터 실행 섹션을 참조하고 실습 경험은 IPv6로 실습 가져오기 워크숍의 Amazon EKS에서 IPv6 이해 섹션을 참조하세요. IPv6



IPv4 클러스터에서 IP 소비 최적화

이 섹션은 레거시 애플리케이션을 실행 중이거나 IPv6로 마이그레이션할 준비가 되지 않은 고객을 대상으로 합니다. 모든 조직이 최대한 빨리 IPv6로 마이그레이션하는 것이 좋지만, 일부 조직에서는 IPv4로 컨테이너 워크로드를 확장하기 위해 대체 접근 방식을 찾아야 할 수도 있습니다. 이러한 이유로 Amazon EKS 클러스터를 사용하여 IPv4(RFC1918) 주소 공간 소비를 최적화하는 아키텍처 패턴도 안내합니다.

성장 계획

IP 소진에 대한 첫 번째 방어선으로, 클러스터가 사용 가능한 모든 IP 주소를 사용하지 않도록 IPv4 VPCs 및 서브넷의 크기를 늘리는 것이 좋습니다. 서브넷에 사용 가능한 IP 주소가 충분하지 않으면 새 포드 또는 노드를 생성할 수 없습니다.

VPC와 서브넷을 빌드하기 전에 필요한 워크로드 규모에서 역방향으로 작업하는 것이 좋습니다. 예를 들어, eksctl(EKS에서 클러스터를 생성하고 관리하기 위한 간단한 CLI 도구)/19 서브넷을 사용하여 클러스터를 빌드하는 경우 기본적으로 생성됩니다. /19의 넷마스크는 대부분의 워크로드 유형에 적합하여 8,000개 이상의 주소를 할당할 수 있습니다.

> **Important**
>
> VPCs 및 서브넷의 크기를 조정할 때 로드 밸런서, RDS 데이터베이스 및 기타 인 VPC 서비스와 같이 IP 주소를 사용할 수 있는 여러 요소(포드 및 노드 제외)가 있을 수 있습니다.

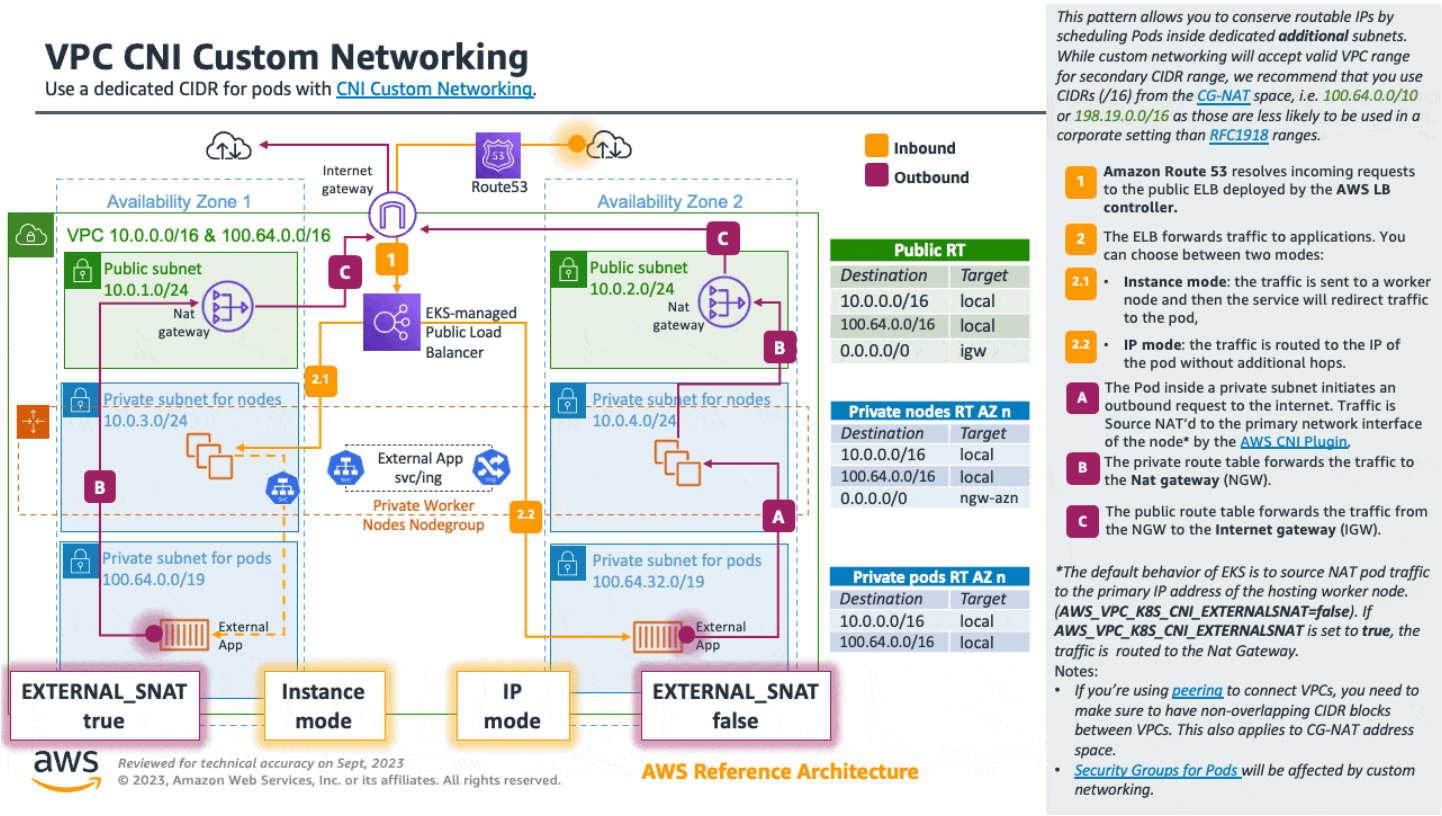
또한 Amazon EKS는 컨트롤 플레인과의 통신을 허용하는 데 필요한 탄력적 네트워크 인터페이스(X-ENI)를 최대 4개까지 생성할 수 있습니다(자세한 내용은 여기 참조). 클러스터 업그레이드 중에 Amazon EKS는 새 X-ENIs를 생성하고 업그레이드가 성공하면 이전 X-ENI를 삭제합니다. 따라서 EKS 클러스터와 연결된 서브넷의 경우 최소 /28(16 IP 주소)의 넷마스크를 사용하는 것이 좋습니다.

샘플 EKS 서브넷 계산기 스프레드시트를 사용하여 네트워크를 계획할 수 있습니다. 스프레드시트는 워크로드 및 VPC ENI 구성을 기반으로 IP 사용량을 계산합니다. IP 사용량을 IPv4 서브넷과 비교하여 구성 및 서브넷 크기가 워크로드에 충분한지 확인합니다. VPC의 서브넷에 사용 가능한 IP 주소가 부족한 경우 VPC의 원래 CIDR 블록을 사용하여 새 서브넷을 생성하는 것이 좋습니다. 이제 Amazon EKS에서 클러스터 서브넷 및 보안 그룹의 수정을 허용합니다.

사용자 지정 네트워킹

RFC1918 IP 공간을 소진하려는 경우 사용자 지정 네트워킹 패턴을 사용하여 전용 추가 서브넷 내에서 포드를 예약하여 라우팅 가능한 IPs를 절약할 수 있습니다. 사용자 지정 네트워킹은 보조 CIDR 범위에 대해 유효한 VPC 범위를 허용하지만 CG-NAT 공간의 CIDRs(/16)`100.64.0.0/10`을 사용하는 것이 좋습니다. 즉, RFC1918 범위보다 기업 환경에서 사용할 가능성이 낮기 `198.19.0.0/16` 때문입니다.

자세한 내용은 사용자 지정 네트워킹 전용 섹션을 참조하세요.



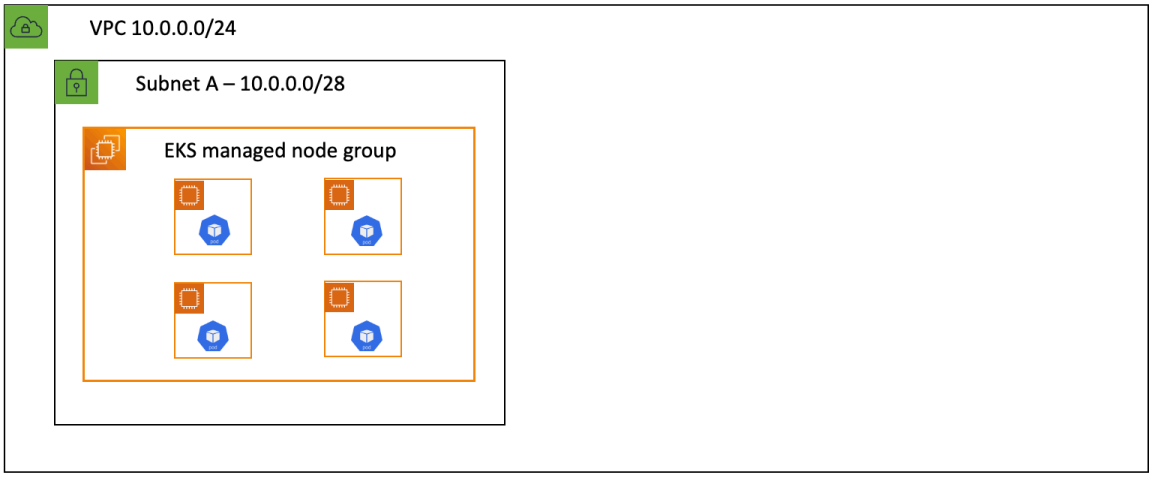
향상된 서브넷 검색

향상된 서브넷 검색은 Amazon VPC CNI에서 검색할 수 있도록 새 서브넷에 태그를 지정하여 IP 소진을 위한 간소화된 네트워크 구성 대안을 제공합니다. 향상된 서브넷 검색을 사용하면 현재 워크로드가 동일한 서브넷에서 계속 실행될 수 있으며 Amazon Elastic Kubernetes Service(Amazon EKS)는 이제 새로운 "사용 가능한 서브넷(들)"에서 추가 포드를 예약할 수 있습니다.

클러스터의 현재 서브넷에 IP 주소가 부족한 경우 다음과 같이 Amazon EKS 클러스터에 서브넷을 추가할 수 있습니다.

1. 새 CIDR 블록을 VPC에 연결합니다.
2. 새 CIDR 블록에서 새 서브넷을 생성하고 "kubernetes.io/role/cni" = "1"로 태그를 지정합니다.
3. "true"에 대한 Amazon VPC CNI 추가 기능의 ENABLE_SUBNET_DISCOVERY 구성을 활성화합니다(버전 1.18.0 이후 기본값).

VPC 및 Amazon EKS 클러스터에서 향상된 서브넷 검색이 활성화되면 다음 다이어그램에 설명된 대로 새 탄력적 네트워크 인터페이스(ENIs)가 Amazon EKS 노드에 연결됩니다.



자세한 내용은 AWS 컨테이너 블로그에서 Amazon VPC CNI의 향상된 서브넷 검색 도입을 참조하세요.

IPs웜 풀 최적화

기본 구성을 사용하면 VPC CNI가 전체 ENI(및 연결된 IPs)를 웜 풀에 유지합니다. 이는 특히 더 큰 인스턴스 유형에서 많은 수의 IPs 소비할 수 있습니다.

클러스터 서브넷에 사용 가능한 IP 주소 수가 제한된 경우 다음 VPC CNI 구성 환경 변수를 검사합니다.

- `WARM_IP_TARGET`
- `MINIMUM_IP_TARGET`
- `WARM_ENI_TARGET`

노드에서 실행될 것으로 예상되는 포드 수와 `MINIMUM_IP_TARGET` 거의 일치하도록 값을 구성할 수 있습니다. 이렇게 하면 포드가 생성될 때 CNI가 EC2 API를 호출하지 않고 웜 풀에서 IP 주소를 할당할 수 있습니다.

값을 WARM_IP_TARGET 너무 낮게 설정하면 EC2 API에 대한 추가 호출이 발생하고 요청 제한이 발생할 수 있습니다. 대규모 클러스터의 경우 요청 제한을 방지하기 MINIMUM_IP_TARGET 위해와 함께를 사용합니다.

이러한 옵션을 구성하려면 aws-k8s-cni.yaml 매니페스트를 다운로드하고 환경 변수를 설정할 수 있습니다. 작성 시 최신 릴리스가 여기에 있습니다. 구성 값의 버전이 설치된 VPC CNI 버전과 일치하는지 확인합니다.

> **Warning**
>
> 이러한 설정은 CNI를 업데이트할 때 기본값으로 재설정됩니다. 업데이트하기 전에 CNI를 백업하십시오. 구성 설정을 검토하여 업데이트에 성공한 후 다시 적용해야 하는지 확인합니다.

기존 애플리케이션의 가동 중지 시간 없이 CNI 파라미터를 즉시 조정할 수 있지만 확장성 요구 사항을 지원하는 값을 선택해야 합니다. 예를 들어 배치 워크로드를 사용하는 경우 포드 규모 요구 사항에 맞게 기본값을 업데이트WARM_ENI_TARGET하는 것이 좋습니다. 를 높은 값으로 설정WARM_ENI_TARGET하면 항상 대규모 배치 워크로드를 실행하는 데 필요한 웜 IP 풀이 유지되므로 데이터 처리 지연을 방지할 수 있습니다.

> **Warning**
>
> IP 주소 소진에 대한 권장 응답은 VPC 설계를 개선하는 것입니다. IPv6 및 보조 CIDRs 같은 솔루션을 고려합니다. 이러한 값을 조정하여 웜 IPs 수를 최소화하는 것은 다른 옵션이 제외된 후의 임시 해결책이어야 합니다. 이러한 값을 잘못 구성하면 클러스터 작업에 방해가 될 수 있습니다. 프로덕션 시스템을 변경하기 전에 이 페이지의 고려 사항을 검토해야 합니다.

IP 주소 인벤토리 모니터링

위에서 설명한 솔루션 외에도 IP 사용률을 파악하는 것도 중요합니다. CNI 지표 헬퍼를 사용하여 서브넷의 IP 주소 인벤토리를 모니터링할 수 있습니다. 사용 가능한 지표는 다음과 같습니다.

- 클러스터가 지원할 수 있는 최대 ENIs 수
- 이미 할당된 ENIs 수
- 포드에 현재 할당된 IP 주소 수
- 사용 가능한 총 및 최대 IP 주소 수

서브넷에 IP 주소가 부족할 경우 알림을 받도록 CloudWatch 경보를 설정할 수도 있습니다.

> **Warning**
>
> VPC CNI의 `DISABLE_METRICS` 변수가 false로 설정되어 있는지 확인합니다.

추가 고려 사항

IP 소진에 도움이 될 수 있는 Amazon EKS에 내장되지 않은 다른 아키텍처 패턴이 있습니다. 예를 들어 VPCs 간 통신을 최적화하거나 여러 계정에서 VPC를 공유하여 IPv4 주소 할당을 제한할 수 있습니다.

다음 패턴에 대해 자세히 알아보세요.

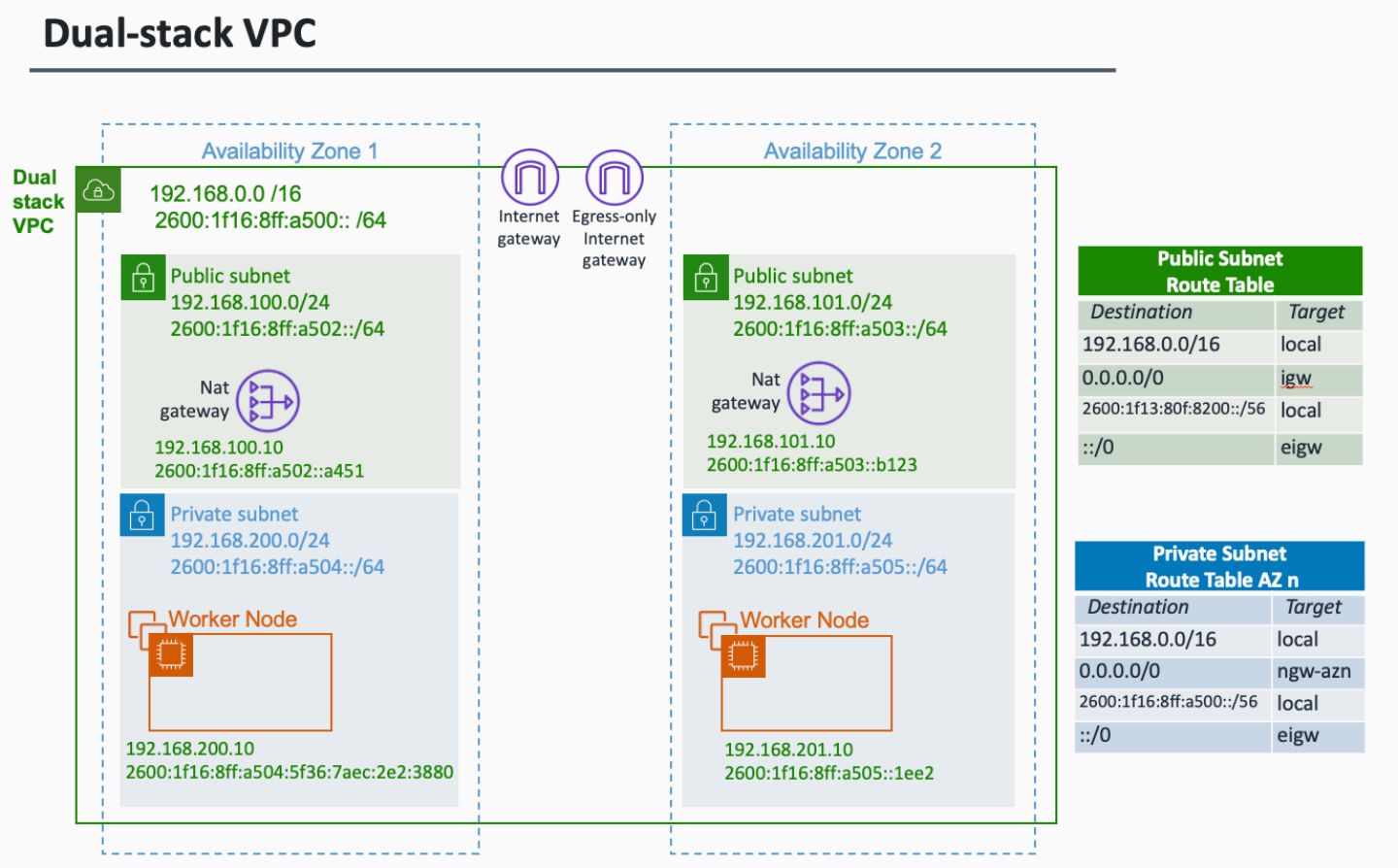
- 하이퍼스케일 Amazon VPC 네트워크 설계
- Amazon VPC Lattice를 사용하여 안전한 다중 계정 다중 VPC 연결을 구축합니다.

IPv6 EKS 클러스터 실행

IPv6 모드의 EKS는 대규모 EKS 클러스터에서 흔히 나타나는 IPv4 소진 문제를 해결합니다. IPv6에 대한 EKS의 지원은 IPv4 주소 공간의 제한된 크기에서 발생하는 IPv4 소진 문제를 해결하는 데 중점을 둡니다. 이는 많은 고객이 제기한 중요한 문제이며 Kubernetes IPv4/IPv6 듀얼 스택 기능과는 다릅니다. 또한 EKS/IPv6는 IPv6 CIDRs을 사용하여 네트워크 경계를 상호 연결할 수 있는 유연성을 제공하므로 CIDR 중복으로 인해 발생할 가능성을 최소화하여 2-Fold 문제(클러스터 내, 교차 클러스터)를 해결할 수 있습니다. IPv6 모드(--ip-family ipv6)로 EKS 클러스터를 배포할 때 작업은 되돌릴 수 없습니다. 간단히 말해서 EKS IPv6 지원은 클러스터의 전체 수명 동안 활성화됩니다.

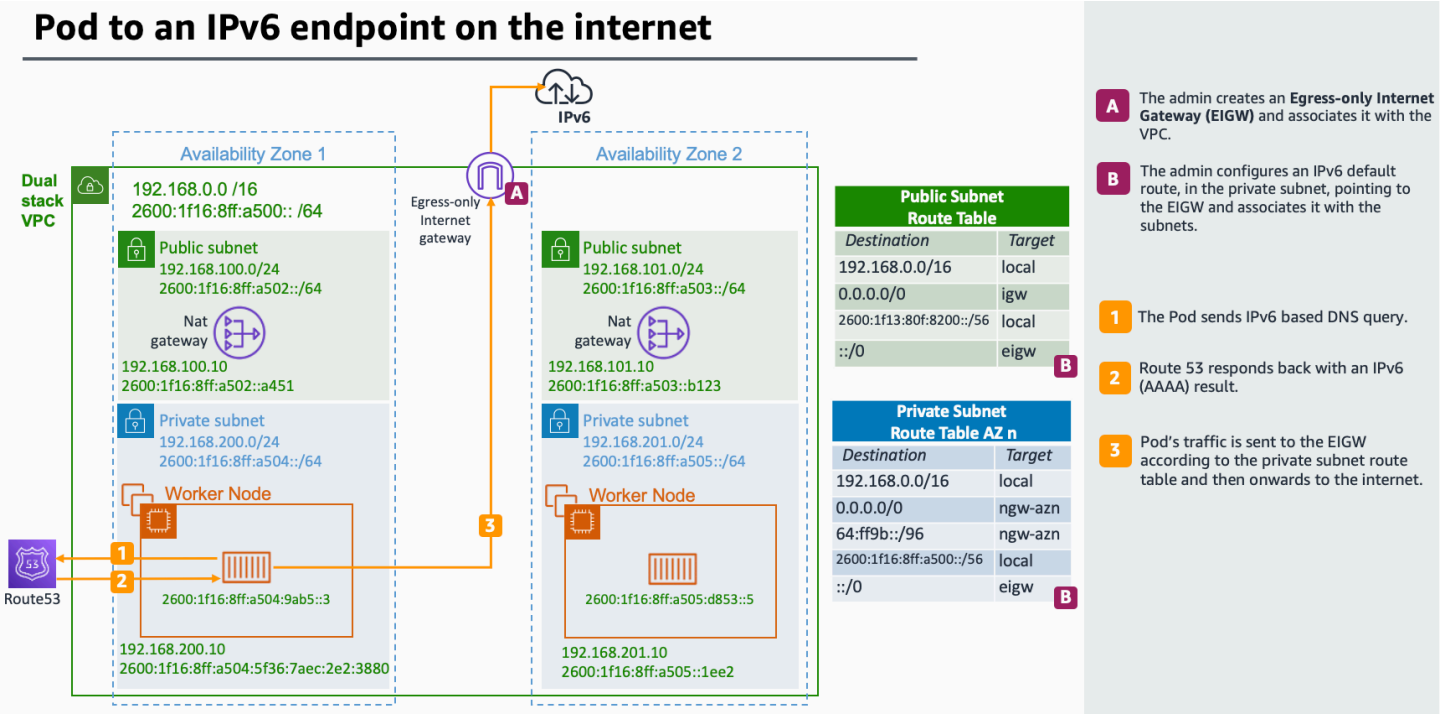
IPv6 EKS 클러스터에서 포드 및 서비스는 레거시 IPv4 엔드포인트와의 호환성을 유지하면서 IPv6 주소를 수신합니다. IPv4 여기에는 외부 IPv4 엔드포인트가 클러스터 내 서비스에 액세스하는 기능과 포드가 외부 IPv4 엔드포인트에 액세스하는 기능이 포함됩니다.

Amazon EKS IPv6 지원은 네이티브 VPC IPv6 기능을 활용합니다. 각 VPC에는 IPv4 주소 접두사(CIDR 블록 크기는 /16~/28일 수 있음)와 Amazon GUA(글로벌 유니캐스트 주소) 내의 고유한 /56 IPv6 주소 접두사(고정)가 할당됩니다. VPC의 각 서브넷에 /64 주소 접두사를 할당할 수 있습니다. 라우팅 테이블, 네트워크 액세스 제어 목록, 피어링 및 DNS 확인과 같은 IPv4 기능은 IPv6 지원 VPC에서 동일한 방식으로 작동합니다. 그런 다음 VPC를 듀얼 스택 VPC라고 하며, 듀얼 스택 서브넷에 이어 다음 다이어그램은 EKS/IPV4IPv6IPv6 VPC 파운데이션 패턴을 보여줍니다.



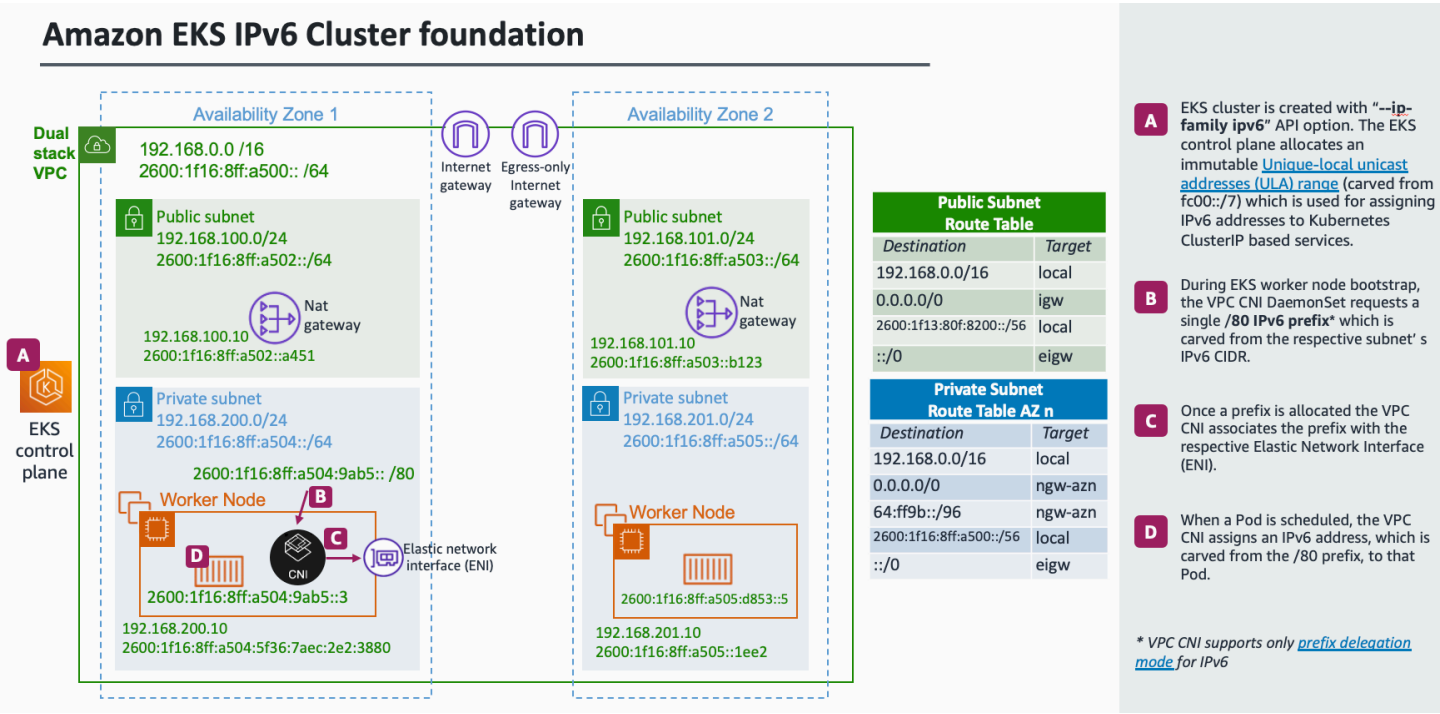
IPv6 세계에서 모든 주소는 인터넷 라우팅이 가능합니다. 기본적으로 VPC는 퍼블릭 GUA 범위에서 IPv6 CIDR을 할당합니다. 그러나 2024년 8월부터 Amazon VPCs IP 주소 관리자(IPAM)를 사용하여 VPC 및 서브넷에 프라이빗 IPv6 주소 지정을 사용할 수도 있습니다. 자세한 내용은 이 AWS 네트워킹 블로그 게시물 및 VPC 설명서를 참조하십시오.

다음 다이어그램은 EKS/IPv6 클러스터 내의 포드 IPv6 인터넷 송신 흐름을 보여줍니다.IPv6



IPv6 서브넷 구현 모범 사례는 VPC 사용 설명서에서 확인할 수 있습니다.

IPv6 EKS 클러스터에서 노드와 포드는 퍼블릭 IPv6 주소를 수신합니다. EKS는 고유 로컬 IPv6 유니캐스트 주소(ULA)를 기반으로 서비스에 IPv6 주소를 할당합니다. IPv6 클러스터의 ULA 서비스 CIDR은 클러스터 생성 단계에서 자동으로 할당되며 IPv4와 달리 지정할 수 없습니다. 다음 다이어그램은 EKS/IPv6 기반 클러스터 컨트롤 플레인 데이터 계획 파운데이션 패턴을 보여줍니다.



개요

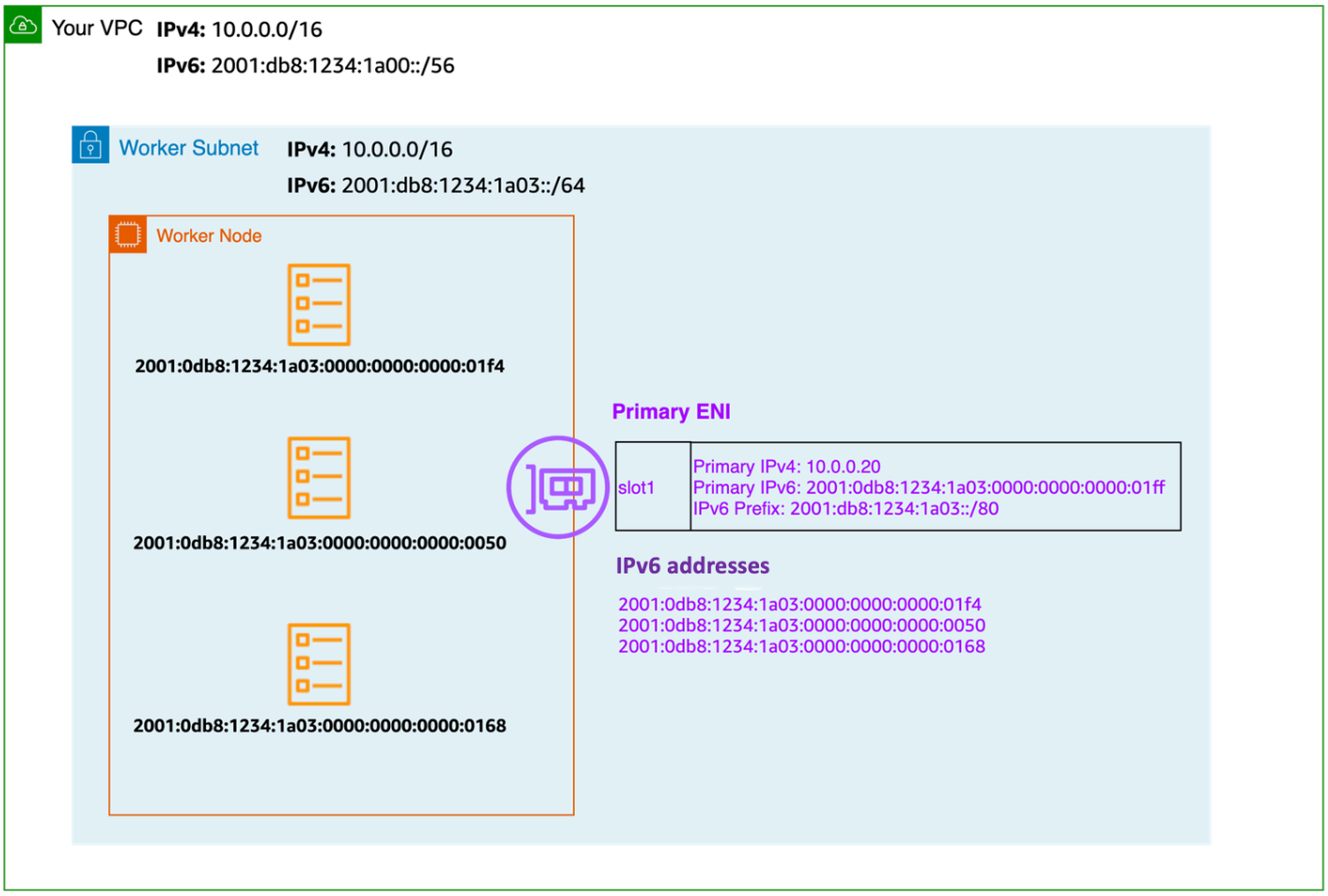
EKS/IPv6는 접두사 모드(VPC-CNI 플러그인 ENI IP 할당 모드)에서만 지원됩니다. 접두사 모드에 대해 자세히 알아봅니다.

접두사 할당은 Nitro 기반 EC2 인스턴스에서만 작동하므로 EKS/IPv6는 클러스터 데이터 플레인이 EC2 Nitro 기반 인스턴스를 사용하는 경우에만 지원됩니다.

간단히 말해서 IPv6 접두사 /80(작업자 노드당)은 ~10^14개의 IPv6 주소를 생성하며, 제한 계수는 더 이상 IPs가 아니라 포드 밀도(리소스별)입니다.

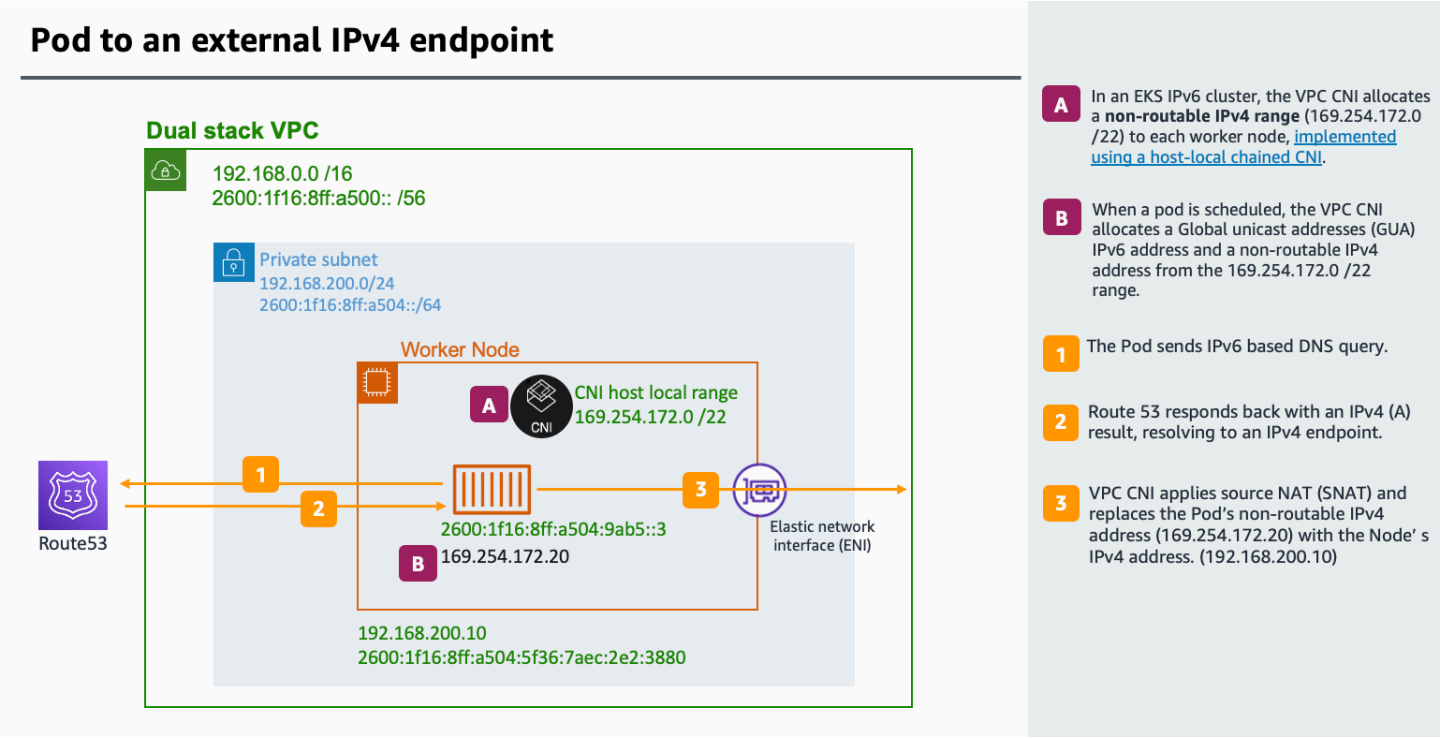
IPv6 접두사 할당은 EKS 작업자 노드 부트스트랩 시간에만 발생합니다. 이 동작은 프라이빗 IPv4 주소를 적시에 할당하려는 VPC CNI 플러그인(ipamd)에 의해 생성되는 제한된 API 호출로 인해 포드 일정에서 높은 포드 이탈 EKS/IPv4 클러스터가 지연되는 시나리오를 완화하는 것으로 알려져 있습니다. IPv4 또한 VPC-CNI 플러그인 고급 노브가 WARM_IP/ENI, MINIMUM_IP를 불필요하게 튜닝하도록 하는 것도 알려져 있습니다.

다음 다이어그램은 IPv6 작업자 노드 탄력적 네트워크 인터페이스(ENI)를 확대합니다.



모든 EKS 작업자 노드에는 해당 DNS 항목과 함께 IPv4 및 IPv6 주소가 할당됩니다. 지정된 작업자 노드의 경우 듀얼 스택 서브넷의 단일 IPv4 주소만 사용됩니다. IPv6에 대한 EKS 지원을 사용하면 의견이 많은 외부 전용 IPv4 모델을 통해 IPv4 엔드포인트(AWS, 온프레미스, 인터넷)와 통신할 수 있습니다. EKS는 VPC CNI 플러그인의 보조인 호스트-로컬 CNI 플러그인을 구현합니다.이 플러그인은 포드에 대한 IPv4 주소를 할당하고 구성합니다. CNI 플러그인은 169.254.172.0/22 범위의 포드에 대해 라우팅할 수 없는 호스트별 IPv4 주소를 구성합니다. 포드에 할당된 IPv4 주소는 작업자 노드에 고유하며 작업자 노드 외에는 알리지 않습니다. 169.254.172.0/22는 대규모 인스턴스 유형을 지원할 수 있는 최대 1,024개의 고유한 IPv4 주소를 제공합니다.

다음 다이어그램은 클러스터 경계(비인터넷) 외부의 IPv4 엔드포인트에 연결하는 IPvIPv6 포드의 흐름을 보여줍니다.

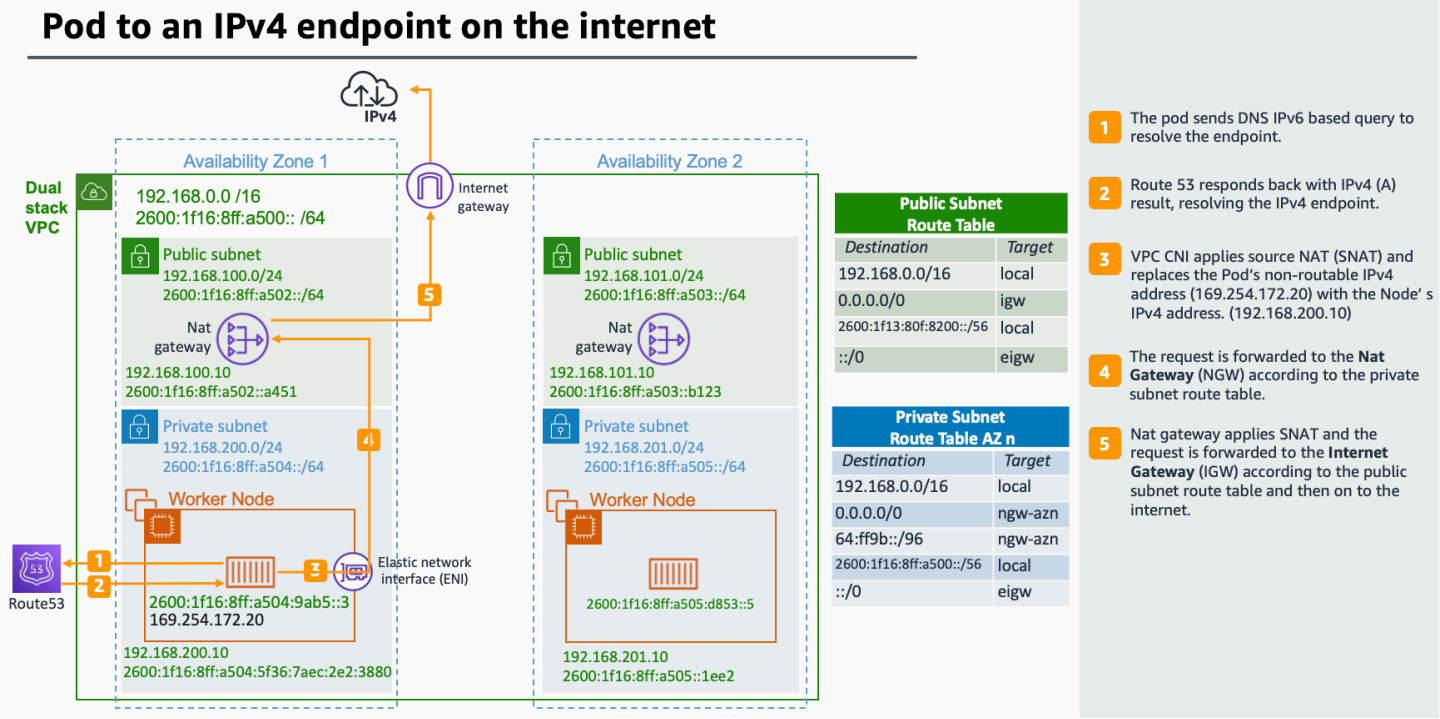


위의 다이어그램에서 포드는 엔드포인트에 대한 DNS 조회를 수행하고 IPv4 "A" 응답을 수신하면 포드의 노드 전용 고유 IPv4 주소가 소스 네트워크 주소 변환(SNAT)을 통해 EC2 작업자 노드에 연결된 기본 네트워크 인터페이스의 프라이빗 IPv4(VPC) 주소로 변환됩니다.

> **Note**
>
> 위의 패턴을 사용하려면 EKS/IPv6 포드가 실행 중인 서브넷에서 DNS64를 비활성화해야 합니다. DNS64가 활성화되면 DNS 해석기는 IPv4 주소와 함께 IPv4-only 엔드포인트에 대해 합성된 IPv4 IPv6 주소를 반환합니다. 따라서 트래픽은 위 패턴과 같이 VPC 내에 머무르지 않고

NAT 게이트웨이(아키텍처에 포함된 경우)의 NAT64 기능을 통해 라우팅됩니다. 이로 인해 예기치 않은 NAT 게이트웨이 사용량 및 관련 비용이 발생할 수 있습니다.

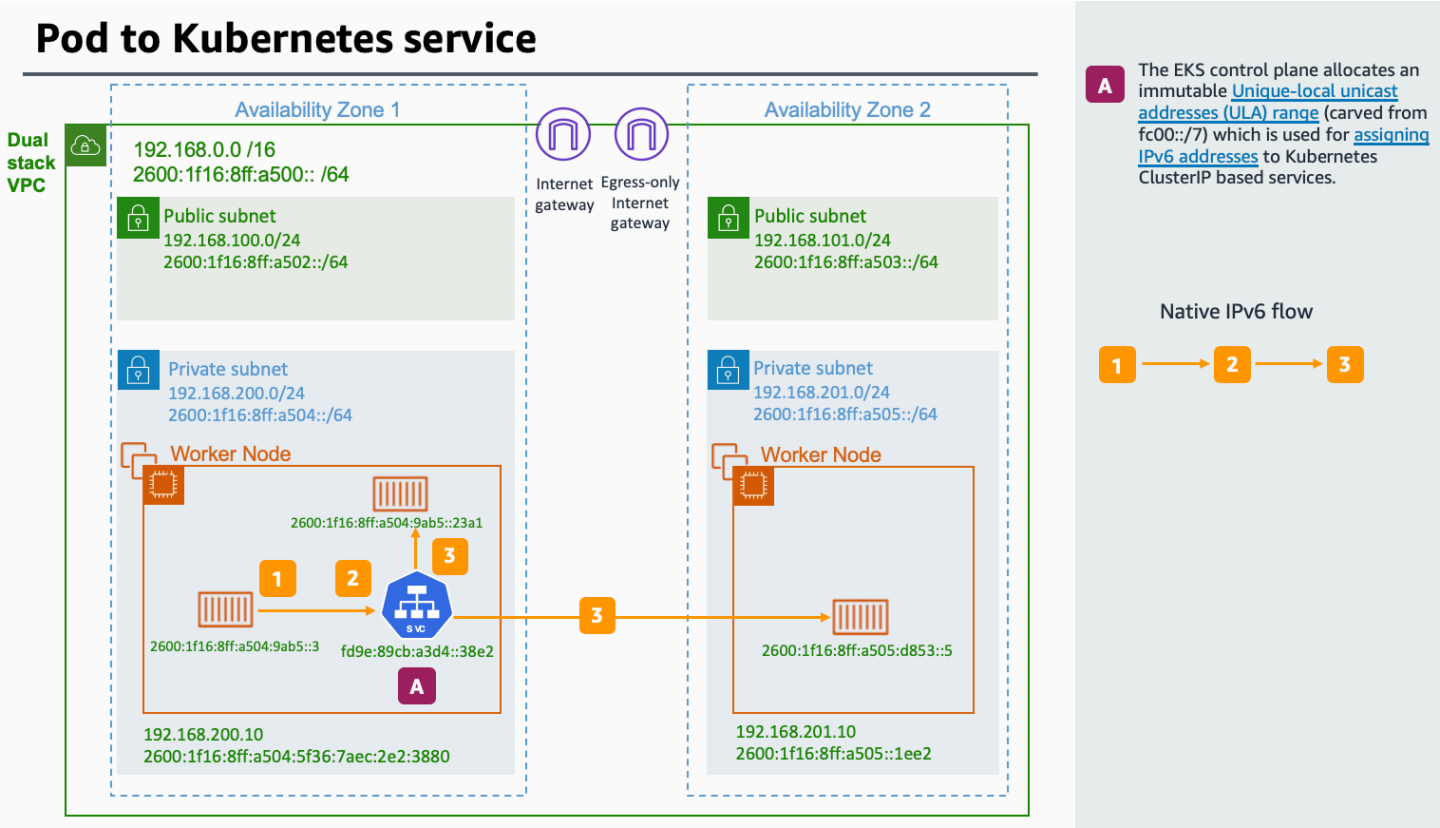
또한 EKS/IPv6 포드는 유사한 흐름이 존재하도록 하려면 퍼블릭 IPv4 주소를 사용하여 인터넷을 통해 IPv4 엔드포인트에 연결해야 합니다. 다음 다이어그램은 클러스터 경계 외부의 IPv4 엔드포인트에 연결하는 IPv6 포드의 흐름을 보여줍니다(인터넷 라우팅 가능). IPv4



위의 다이어그램에서 포드는 엔드포인트에 대한 DNS 조회를 수행하고 IPv4 "A" 응답을 수신하면 포드의 노드 전용 고유 IPv4 주소가 소스 네트워크 주소 변환(SNAT)을 통해 EC2 작업자 노드에 연결된 기본 네트워크 인터페이스의 프라이빗 IPv4(VPC) 주소로 변환됩니다. 그런 다음 포드 IPv4 주소(소스 IPv4: EC2 기본 IP)가 IPv4 NAT 게이트웨이로 라우팅되어 EC2 기본 IP가 유효한 인터넷 라우팅 가능 IPv4 퍼블릭 IP 주소(NAT 게이트웨이 할당 퍼블릭 IP)로 변환됩니다.

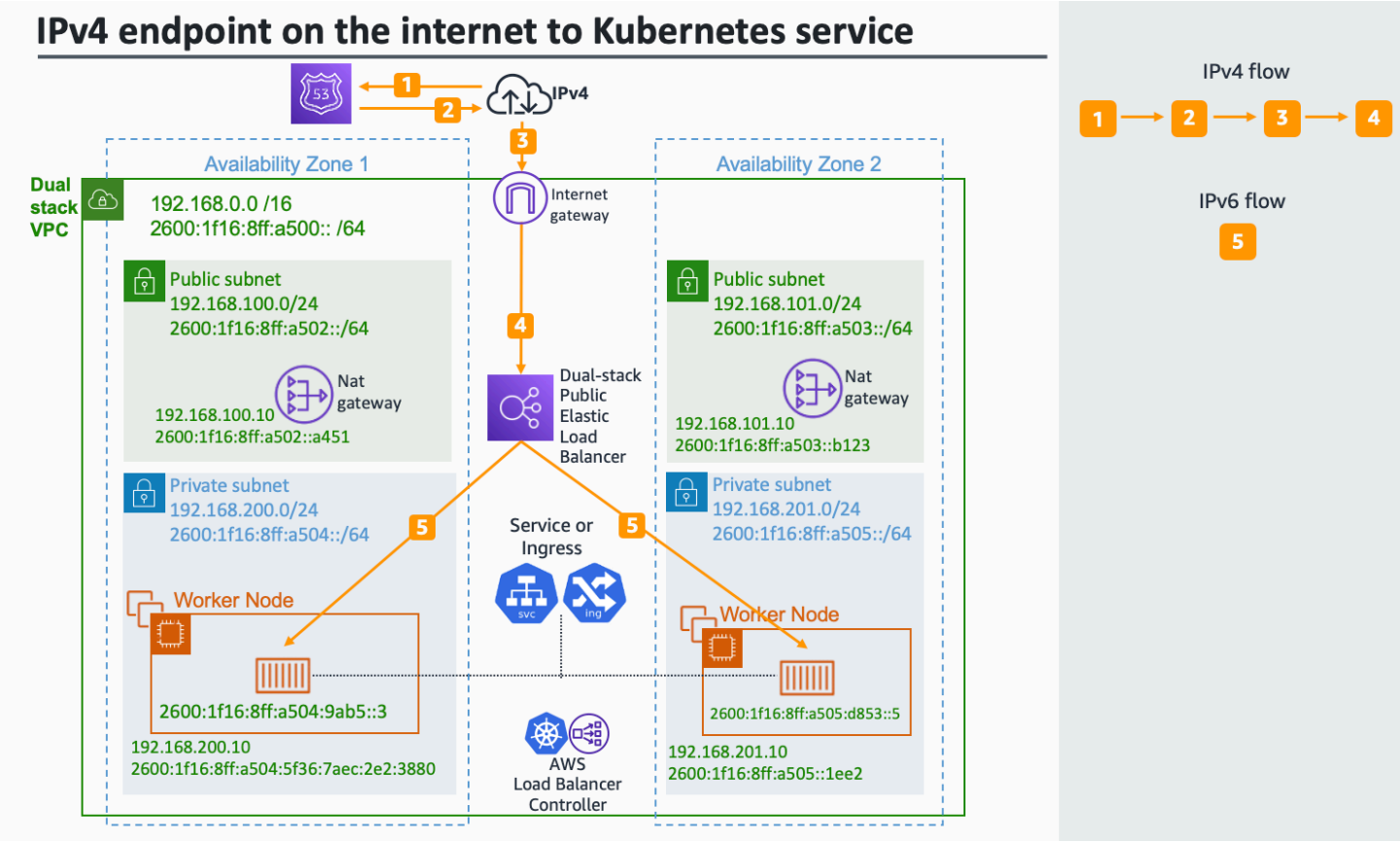
노드 간 Pod-to-Pod 통신은 항상 IPv6 주소를 사용합니다. VPC CNI는 IPv4 연결을 차단하면서 IPv6를 처리하도록 iptable을 구성합니다. IPv4

Kubernetes 서비스는 고유 로컬 IPv6 유니캐스트 주소(ULA)로부터 IPv6 주소(ClusterIP)만 수신합니다. IPv6 IPv6 클러스터의 ULA 서비스 CIDR은 EKS 클러스터 생성 단계에서 자동으로 할당되며 수정할 수 없습니다. 다음 다이어그램은 Kubernetes Service에 대한 포드 흐름을 보여줍니다.



서비스는 AWS 로드 밸런서를 사용하여 인터넷에 노출됩니다. 로드 밸런서는 퍼블릭 IPv4 및 IPv6 주소, 즉 듀얼 스택 로드 밸런서를 수신합니다. IPv6 클러스터 kubernetes 서비스에 액세스하는 IPv4 클라이언트의 경우 로드 밸런서는 IPv4에서 IPv6로의 변환을 수행합니다. IPv6

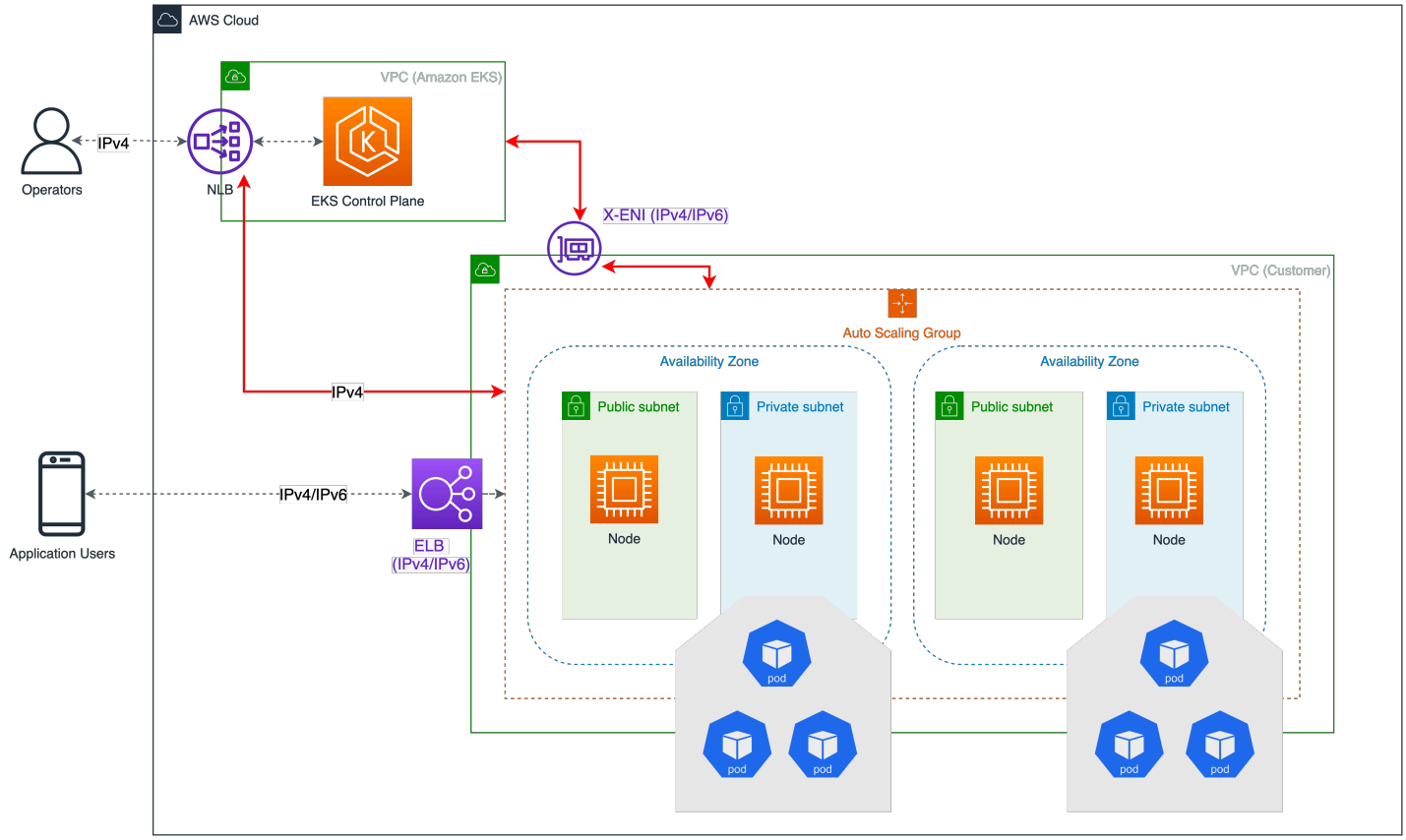
Amazon EKS는 프라이빗 서브넷에서 작업자 노드와 포드를 실행할 것을 권장합니다. 프라이빗 서브넷에 있는 노드에서 실행되는 포드로 트래픽을 로드 밸런싱하는 퍼블릭 서브넷에서 퍼블릭 로드 밸런서를 생성할 수 있습니다. 다음 다이어그램은 EKS/IPv6 수신 기반 서비스에 액세스하는 인터넷 IPv4 사용자를 보여줍니다.



> **Note**
>
> 위의 패턴은가 AWS 로드 밸런서 컨트롤러의 최신 버전을 배포해야 합니다.

EKS 제어 플레인 데이터 플레인 통신

EKS는 크로스 계정 ENIs(X-ENIs)를 듀얼 스택 모드(IPv4/IPv6)로 프로비저닝합니다. kubelet 및 kube-proxy와 같은 Kubernetes 노드 구성 요소는 듀얼 스택을 지원하도록 구성됩니다. Kubelet 및 kube-proxy는 hostNetwork 모드에서 실행되며 노드의 기본 네트워크 인터페이스에 연결된 IPv4 및 IPv6 주소 모두에 바인딩됩니다. Kubernetes API 서버는 IPv6 기반 X-ENIs를 통해 포드 및 노드 구성 요소와 통신합니다. 포드는 X-ENIs를 통해 API-서버와 통신하고 포드와 API-서버 간 통신은 항상 IPv6 모드를 사용합니다.



권장 사항

컴퓨팅 리소스 기반 일정

단일 IPv6 접두사는 단일 노드에서 많은 포드를 실행하기에 충분합니다. 또한 노드의 최대 포드 수에 대한 ENI 및 IP 제한을 효과적으로 제거합니다. IPv6는 max-Pods에 대한 직접적인 종속성을 제거하지만 m5.large와 같은 작은 인스턴스 유형의 접두사 연결을 사용하는 경우 IP 주소를 소진하기 훨씬 전에 인스턴스의 CPU 및 메모리 리소스를 소진할 가능성이 높습니다. 자체 관리형 노드 그룹 또는 사용자 지정 AMI ID가 있는 관리형 노드 그룹을 사용하는 경우 EKS 권장 최대 포드 값을 수동으로 설정해야 합니다.

다음 공식을 사용하여 IPv6 EKS 클러스터의 노드에 배포할 수 있는 최대 포드 수를 결정할 수 있습니다.

```
((Number of network interfaces for instance type (number of prefixes per network
 interface-1)* 16) + 2
```

```
((3 ENIs)_((10 secondary IPs per ENI-1)_ 16)) + 2 = 460 (real)
```

관리형 노드 그룹은 자동으로 최대 포드 수를 계산합니다. 리소스 제한으로 인한 포드 예약 실패를 방지하려면 최대 포드 수에 대한 EKS의 권장 값을 변경하지 마세요.

기존 사용자 지정 네트워킹의 목적 평가

사용자 지정 네트워킹이 현재 활성화된 경우 Amazon EKS는 IPv6를 사용하여 필요에 대한 재평가를 권장합니다. 사용자 지정 네트워킹을 사용하여 IPv4 소진 문제를 해결하도록 선택한 경우 IPv6에서는 더 이상 필요하지 않습니다. 사용자 지정 네트워킹을 활용하여 노드 및 포드에 대한 별도의 네트워크와 같은 보안 요구 사항을 충족하는 경우 EKS 로드맵 요청을 제출하는 것이 좋습니다.

EKS/IPv6 클러스터의 Fargate 포드

EKS는 Fargate에서 실행되는 포드에 대해 IPv6를 지원합니다. Fargate에서 실행되는 포드는 VPC CIDR 범위(IPv4IPv6)에서 조각된 IPv6 및 VPC 라우팅 가능 프라이빗 IPv4 주소를 사용합니다. IPv4 IPv4IPv6 간단히 말해서 EKS/Fargate 포드 클러스터의 밀도는 사용 가능한 IPv4 및 IPv6 주소로 제한됩니다. 향후 성장을 위해 듀얼 스택 서브넷/VPC CIDRs 것이 좋습니다. 기본 서브넷에 사용 가능한 IPv6 주소와 관계없이 사용 가능한 IPv4 주소가 없는 경우 새 Fargate 포드를 예약할 IPv6 수 없습니다.

AWS Load Balancer Controller(LBC) 배포

업스트림 인트리 Kubernetes 서비스 컨트롤러는 IPv6를 지원하지 않습니다. 최신 버전의 AWS Load Balancer Controller 추가 기능을 사용하는 것이 좋습니다. LBC는 및 주석이 달린 해당 kubernetes 서비스/수신 정의를 사용할 때만 듀얼 스택 NLB 또는 듀얼 스택 ALB를 배포`"alb.ingress.kubernetes.io/ip-address-type: dualstack"`합니다. `"alb.ingress.kubernetes.io/target-type: ip"`

사용자 지정 네트워킹

기본적으로 Amazon VPC CNI는 기본 서브넷에서 선택한 IP 주소를 포드에 할당합니다. 기본 서브넷은 기본 ENI가 연결된 서브넷 CIDR로, 일반적으로 노드/호스트의 서브넷입니다.

서브넷 CIDR이 너무 작으면 CNI가 포드에 할당할 충분한 보조 IP 주소를 획득하지 못할 수 있습니다. 이는 EKS IPv4 클러스터의 일반적인 문제입니다.

사용자 지정 네트워킹은이 문제의 한 가지 해결 방법입니다.

사용자 지정 네트워킹은 보조 VPC 주소 공간(CIDR)에서 노드 및 포드 IPs. 사용자 지정 네트워킹 지원은 ENIConfig 사용자 지정 리소스를 지원합니다. ENIConfig에는 포드가 속할 보안 그룹(들)과 함께 대체 서브넷 CIDR 범위(보조 VPC CIDR에서 조각됨)가 포함됩니다. 사용자 지정 네트워킹이 활성화되면

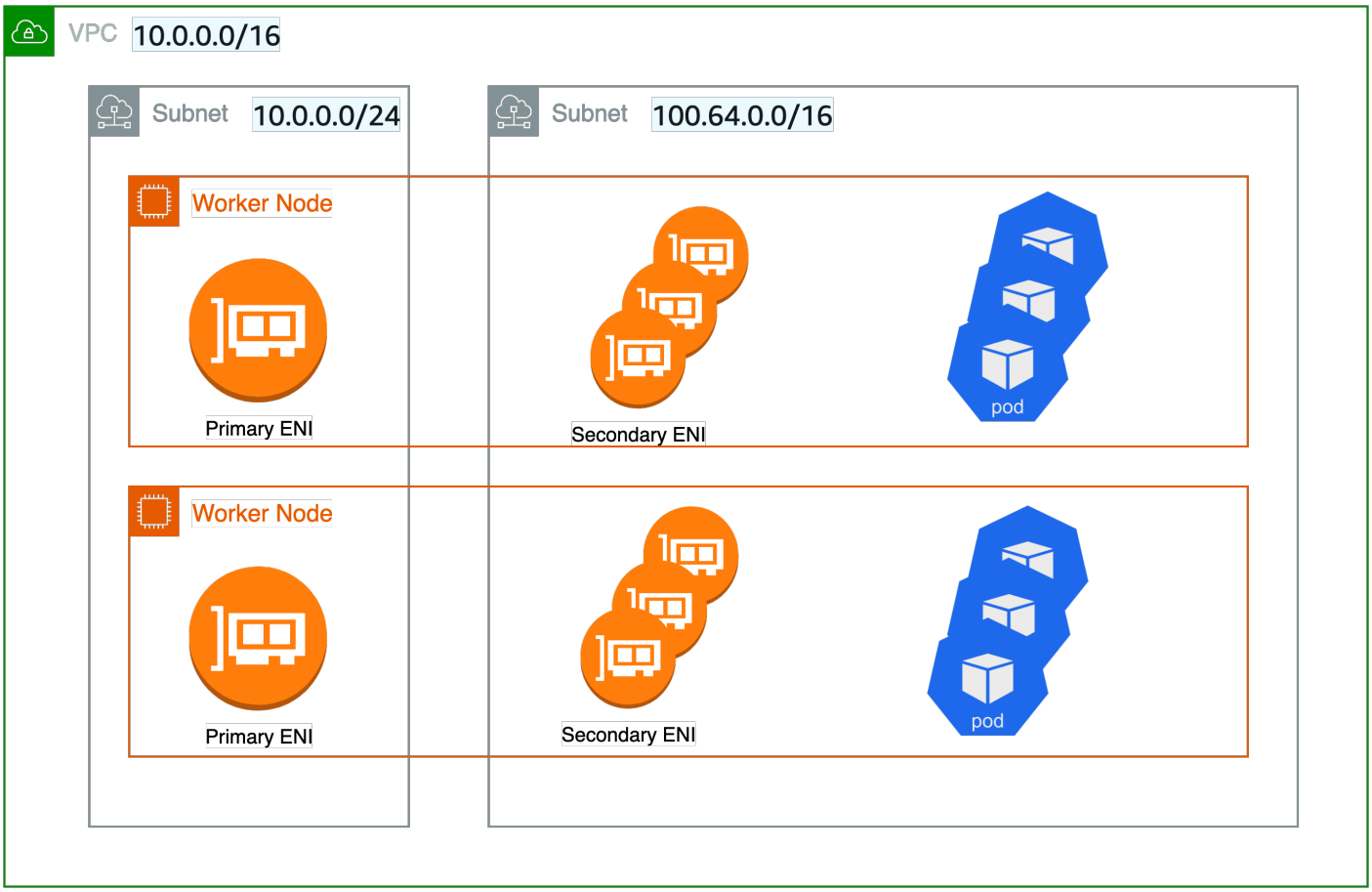
VPC CNI는 ENIsENIConfig를 생성합니다. CNI는 ENIConfig CRD에 정의된 CIDR 범위의 IP 주소를 포드에 할당합니다.

기본 ENI는 사용자 지정 네트워킹에서 사용되지 않으므로 노드에서 실행할 수 있는 최대 포드 수가 더 적습니다. 호스트 네트워크 포드는 기본 ENI에 할당된 IP 주소를 계속 사용합니다. 또한 기본 ENI는 소스 네트워크 변환을 처리하고 노드 외부에서 포드 트래픽을 라우팅하는 데 사용됩니다.

구성의 예제

사용자 지정 네트워킹은 보조 CIDR 범위에 대해 유효한 VPC 범위를 허용하지만 CG-NAT 공간의 CIDRs(/16)을 사용하는 것이 좋습니다. 즉, 100.64.0.0/10 또는 198.19.0.0/16은 다른 RFC1918 범위보다 기업 환경에서 사용될 가능성이 낮기 때문입니다. VPC와 함께 사용할 수 있는 허용 및 제한된 CIDR 블록 연결에 대한 자세한 내용은 VPC 설명서의 VPC 및 서브넷 크기 조정 섹션에서 IPv4 CIDR 블록 연결 제한을 참조하세요.

아래 다이어그램과 같이 작업자 노드의 기본 탄력적 네트워크 인터페이스(ENI)는 여전히 기본 VPC CIDR 범위(이 경우 10.0.0.0/16)를 사용하지만 보조 ENIs 보조 VPC CIDR 범위(이 경우 100.64.0.0/16)를 사용합니다. 이제 포드가 100.64.0.0/16 CIDR 범위를 사용하도록 하려면 사용자 지정 네트워킹을 사용하도록 CNI 플러그인을 구성해야 합니다. 여기에 설명된 단계를 따를 수 있습니다.



CNI가 사용자 지정 네트워킹을 사용하도록 하려면 AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG 환경 변수를 로 설정합니다true.

```
kubectl set env daemonset aws-node -n kube-system
 AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG=true
```

AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG=true인 경우 CNI는에 정의된 서브넷에서 포드 IP 주소를 할당합니다ENIConfig. ENIConfig 사용자 지정 리소스는 포드가 예약될 서브넷을 정의하는 데 사용됩니다.

```
apiVersion : crd.k8s.amazonaws.com/v1alpha1
kind : ENIConfig
metadata:
  name: us-west-2a
spec:
  securityGroups:
    - sg-0dff111a1d11c1c11
```

```
  subnet: subnet-011b111c1f11fdf11
```

ENIconfig 사용자 지정 리소스를 생성할 때 새 작업자 노드를 생성하고 기존 노드를 드레이닝해야 합니다. 기존 작업자 노드와 포드는 영향을 받지 않습니다.

권장 사항

다음과 같은 경우 사용자 지정 네트워킹 사용

IPv4 소진으로 인해 아직 IPv6를 사용할 수 없는 경우 사용자 지정 네트워킹을 고려하는 것이 좋습니다. RFC6598 공간에 대한 Amazon EKS 지원을 통해 RFC1918 이상으로 포드를 확장하여 소진 문제를 해결할 수 있습니다. 노드의 포드 밀도를 높이려면 사용자 지정 네트워킹과 함께 접두사 위임을 사용하는 것이 좋습니다.

보안 요구 사항이 서로 다른 보안 그룹 요구 사항이 있는 다른 네트워크에서 포드를 실행해야 하는 경우 사용자 지정 네트워킹을 고려할 수 있습니다. 사용자 지정 네트워킹을 활성화하면 포드는 ENIConfig에 정의된 것과 노드의 기본 네트워크 인터페이스와 다른 서브넷 또는 보안 그룹을 사용합니다.

사용자 지정 네트워킹은 실제로 여러 EKS 클러스터 및 애플리케이션을 배포하여 온프레미스 데이터 센터 서비스를 연결하는 데 이상적인 옵션입니다. Amazon Elastic Load Balancing 및 NAT-GW와 같은 서비스의 경우 VPC에서 EKS에 액세스할 수 있는 프라이빗 주소(RFC1918) 수를 늘리는 동시에 여러 클러스터의 포드에 라우팅할 수 없는 CG-NAT 공간을 사용할 수 있습니다. 전송 게이트웨이 및 공유 서비스 VPC를 사용한 사용자 지정 네트워킹(고가용성을 위해 여러 가용 영역의 NAT 게이트웨이 포함)을 통해 확장 가능하고 예측 가능한 트래픽 흐름을 제공할 수 있습니다. 이 블로그 게시물에서는 사용자 지정 네트워킹을 사용하여 EKS 포드를 데이터 센터 네트워크에 연결하는 가장 권장되는 방법 중 하나인 아키텍처 패턴을 설명합니다.

다음과 같은 경우 사용자 지정 네트워킹 방지

IPv6 구현 준비 완료

사용자 지정 네트워킹은 IP 소진 문제를 완화할 수 있지만 추가 운영 오버헤드가 필요합니다. 현재 듀얼 스택(IPv4/IPv6) VPC를 배포하거나 플랜에 IPv6 지원이 포함된 경우 대신 IPv6 클러스터를 구현하는 것이 좋습니다. IPv6 EKS 클러스터를 설정하고 앱을 마이그레이션할 수 있습니다. IPv6 EKS 클러스터에서 Kubernetes와 포드는 모두 IPv6 주소를 가져오며 IPv4 및 IPv6 엔드포인트 모두에서 송수신할 수 있습니다. IPv6 EKS 클러스터 실행 모범 사례를 검토하세요.

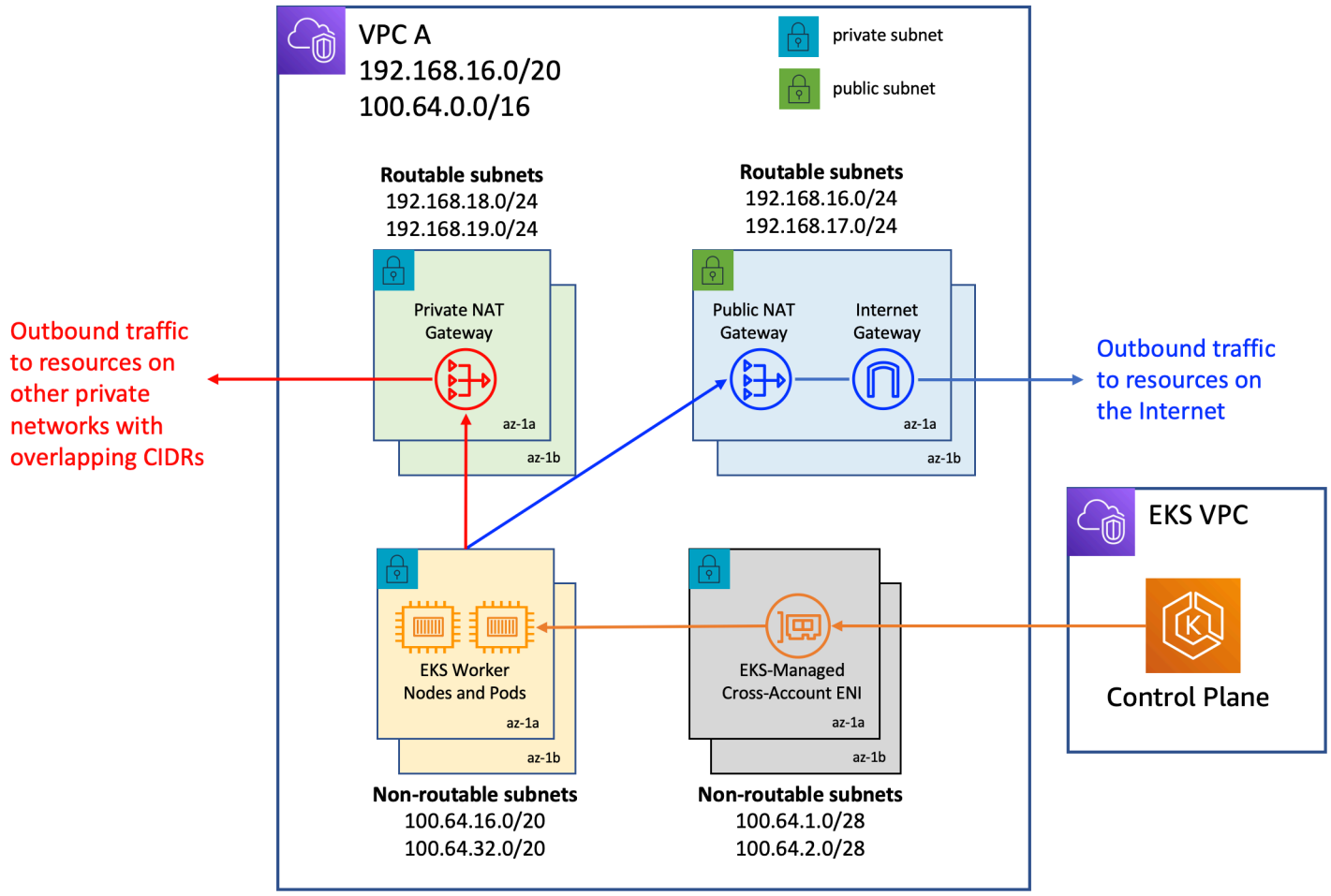
CG-NAT 공간 소진

또한 현재 CG-NAT 스페이스의 CIDRs을 사용하고 있거나 보조 CIDR을 클러스터 VPC와 연결할 수 없는 경우 대체 CNI 사용과 같은 다른 옵션을 탐색해야 할 수 있습니다. 패치를 디버깅하고 오픈 소스 CNI 플러그인 프로젝트에 제출하려면 상용 지원을 받거나 사내 지식을 보유하는 것이 좋습니다. 자세한 내용은 대체 CNI 플러그인 사용 설명서를 참조하세요.

프라이빗 NAT 게이트웨이 사용

Amazon VPC는 이제 프라이빗 NAT 게이트웨이 기능을 제공합니다. Amazon의 프라이빗 NAT 게이트웨이를 사용하면 프라이빗 서브넷의 인스턴스가 중복 CIDR을 사용하여 다른 VPCs 및 온프레미스 네트워크에 연결할 수 있습니다. CIDRs 이 블로그 게시물에 설명된 방법을 활용하여 프라이빗 NAT 게이트웨이를 사용하여 클라이언트가 표현한 중요한 수신 거부인 중복 CIDRs로 인한 EKS 워크로드의 통신 문제를 극복하는 것이 좋습니다. 사용자 지정 네트워킹은 중복되는 CIDR 문제를 자체적으로 해결할 수 없으며 구성 문제에 추가됩니다.

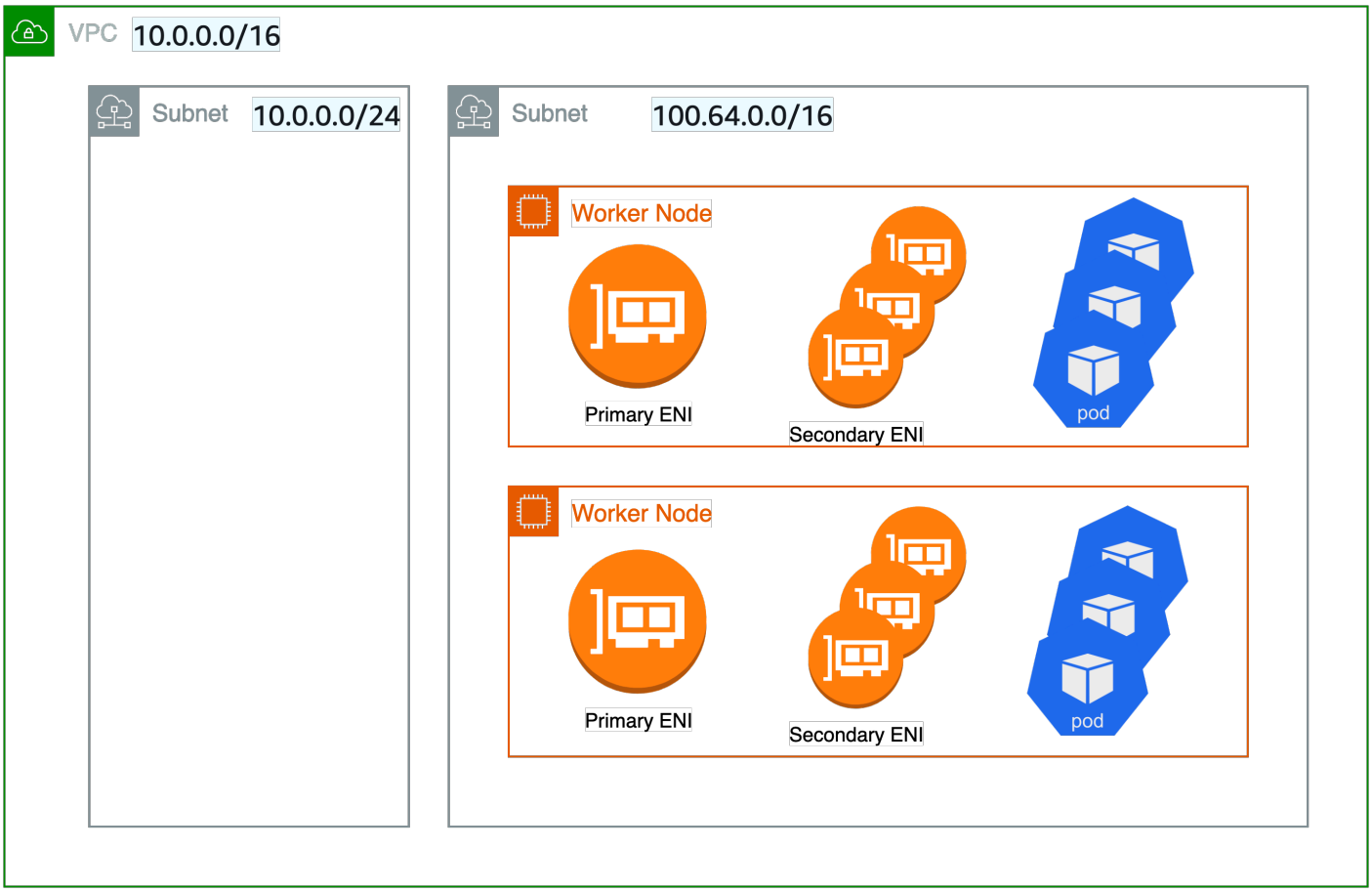
이 블로그 게시물 구현에 사용되는 네트워크 아키텍처는 Amazon VPC 설명서의 중첩 네트워크 간 통신 활성화의 권장 사항을 따릅니다. 이 블로그 게시물에서 볼 수 있듯이, 고객의 프라이빗 IP 소진 문제를 관리하기 위해 RFC6598 주소와 함께 프라이빗 NAT 게이트웨이의 사용을 확장할 수 있습니다. EKS 클러스터, 작업자 노드는 라우팅 불가능한 100.64.0.0/16 VPC 보조 CIDR 범위에 배포되는 반면, 프라이빗 NAT 게이트웨이, NAT 게이트웨이는 라우팅 가능한 RFC1918 CIDR 범위에 배포됩니다. 이 블로그에서는 라우팅할 수 없는 CIDR 범위가 겹치는 VPCs 간 통신을 용이하게 하기 위해 전송 게이트웨이를 사용하여 VPCs 연결하는 방법을 설명합니다. VPC의 라우팅 불가능한 주소 범위의 EKS 리소스가 주소 범위가 겹치지 않는 다른 VPCs와 통신해야 하는 사용 사례의 경우 고객은 VPC 피어링을 사용하여 이러한 VPCs. VPC 피어링 연결을 통한 가용 영역 내 모든 데이터 전송이 이제 무료이므로이 방법을 사용하면 비용을 절감할 수 있습니다.



노드 및 포드의 고유한 네트워크

보안상의 이유로 노드와 포드를 특정 네트워크에 격리해야 하는 경우 더 큰 보조 CIDR 블록(예: 100.64.0.0/8)에서 서브넷에 노드와 포드를 배포하는 것이 좋습니다. VPC에 새 CIDR을 설치한 후 보조 CIDR을 사용하여 다른 노드 그룹을 배포하고 원래 노드를 드레이닝하여 포드를 새 작업자 노드에 자동으로 재배포할 수 있습니다. 이를 구현하는 방법에 대한 자세한 내용은이 블로그 게시물을 참조하세요.

아래 다이어그램에 표시된 설정에서는 사용자 지정 네트워킹이 사용되지 않습니다. 대신 Kubernetes 작업자 노드는 100.64.0.0/10과 같은 VPC의 보조 VPC CIDR 범위의 서브넷에 배포됩니다. EKS 클러스터를 계속 실행할 수 있지만(컨트롤 플레인은 원래 서브넷에 남아 있음) 노드와 포드는 보조 서브넷으로 이동합니다. 이는 통상적이지는 않지만 VPC에서 IP 소진 위험을 완화하는 또 다른 기법입니다. 포드를 새 작업자 노드에 재배포하기 전에 이전 노드를 드레이닝하는 것이 좋습니다.



가용 영역 레이블을 사용하여 구성 자동화

Kubernetes를 활성화하여 작업자 노드 가용 영역(AZ)에 해당하는 ENIConfig를 자동으로 적용할 수 있습니다.

Kubernetes는 [topology.kubernetes.io/zone] 작업자 노드에 태그를 자동으로 추가합니다. Amazon EKS는 AZ당 보조 서브넷(대체 CIDR)이 하나만 있는 경우 가용 영역을 ENI 구성 이름으로 사용할 것을 권장합니다. 그런 다음 ENI 구성 이름을 검색하는 데 사용되는 레이블을 로 설정할 수 있습니다`topology.kubernetes.io/zone`. 태그`failure-domain.beta.kubernetes.io/zone`는 더 이상 사용되지 않으며 태그 로 대체됩니다`topology.kubernetes.io/zone`.

1. `name` 필드를 VPC의 가용 영역으로 설정합니다.
2. 다음 명령을 통해 자동 구성 활성화
3. 다음 명령을 통해 구성 레이블을 설정합니다.

```
kubectl set env daemonset aws-node -n kube-system
 "AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG=true"
kubectl set env daemonset aws-node -n kube-system
 "ENI_CONFIG_LABEL_DEF=topology.kubernetes.io/zone"
```

가용 영역당 보조 서브넷이 여러 개 있는 경우 특정를 생성해야 합니다ENI_CONFIG_LABEL_DEF. k8s.amazonaws.com/eniConfig 및 ENI_CONFIG_LABEL_DEF와 같은 사용자 지정 eniConfig 이름으로 및 레이블 노드로 k8s.amazonaws.com/eniConfig=us-west-2a-subnet-1를 구성하는 것을 고려할 수 있습니다k8s.amazonaws.com/eniConfig=us-west-2a-subnet-2.

보조 네트워킹 구성 시 포드 교체

사용자 지정 네트워킹을 활성화해도 기존 노드는 수정되지 않습니다. 사용자 지정 네트워킹은 방해가 되는 작업입니다. 사용자 지정 네트워킹을 활성화한 후 클러스터의 모든 작업자 노드를 롤링 교체하는 대신, Lambda 함수를 호출하는 사용자 지정 리소스로 EKS 시작 안내서의 AWS CloudFormation 템플릿을 업데이트하여 aws-node Daemonset을 환경 변수로 업데이트하여 작업자 노드가 프로비저닝되기 전에 사용자 지정 네트워킹을 활성화하는 것이 좋습니다.

사용자 지정 CNI 네트워킹 기능으로 전환하기 전에 클러스터에 포드가 실행 중인 노드가 있는 경우 노드를 코딩하고 드레이닝하여 포드를 정상적으로 종료한 다음 노드를 종료해야 합니다. ENIConfig 레이블 또는 주석과 일치하는 새 노드만 사용자 지정 네트워킹을 사용하므로 이러한 새 노드에 예약된 포드에 보조 CIDR의 IP를 할당할 수 있습니다.

노드당 최대 포드 계산

노드의 기본 ENI는 더 이상 포드 IP 주소를 할당하는 데 사용되지 않으므로 지정된 EC2 인스턴스 유형에서 실행할 수 있는 포드 수가 줄어듭니다. 이 제한을 해결하려면 사용자 지정 네트워킹과 함께 접두사 할당을 사용할 수 있습니다. 접두사 할당을 사용하면 각 보조 IP가 보조 ENIs.

사용자 지정 네트워킹이 있는 m5.large 인스턴스의 최대 포드 수를 고려합니다.

접두사 할당 없이 실행할 수 있는 최대 포드 수는 29개입니다.

- `3 ENIs - 1) * (10 secondary IPs per ENI - 1 + 2 = 20`

접두사 연결을 활성화하면 포드 수가 290개로 증가합니다.

- `(3 ENIs - 1) * ((10 secondary IPs per ENI - 1) * 16 + 2 = 290`

그러나 인스턴스의 가상 CPUs 수가 다소 적기 때문에 최대 포드를 290개가 아닌 110으로 설정하는 것이 좋습니다. 더 큰 인스턴스에서 EKS는 최대 포드 값을 250으로 권장합니다. 더 작은 인스턴스 유형(예: m5.large)의 접두사 연결을 사용하는 경우 IP 주소보다 훨씬 먼저 인스턴스의 CPU 및 메모리 리소스를 소진할 수 있습니다.

> **Note**
>
> CNI 접두사가 ENI에 /28 접두사를 할당하는 경우 IP 주소의 연속 블록이어야 합니다. 접두사가 생성되는 서브넷이 고도로 조각화된 경우 접두사 연결이 실패할 수 있습니다. 클러스터에 대한 새 전용 VPC를 생성하거나 서브넷에 접두사 연결 전용 CIDR 세트를 예약하여 이러한 문제를 완화할 수 있습니다. 이 주제에 대한 자세한 내용은 서브넷 CIDR 예약을 참조하세요.

CG-NAT 스페이스의 기존 사용 식별

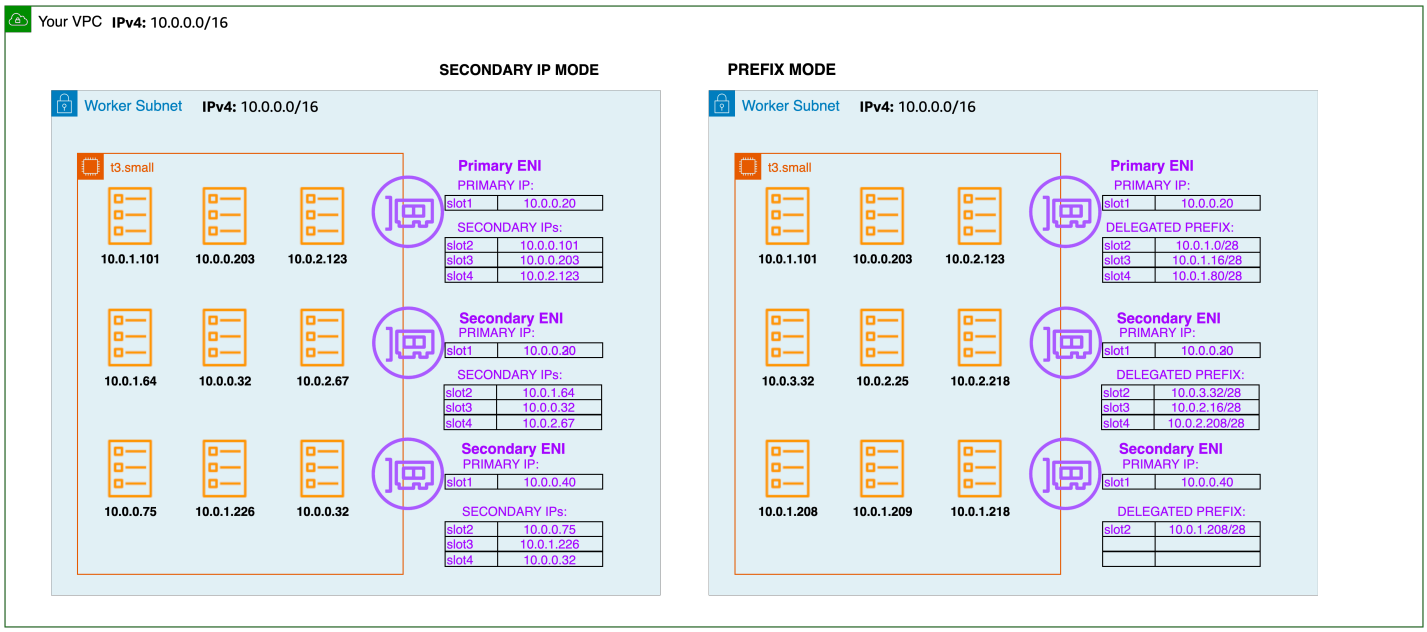
사용자 지정 네트워킹을 사용하면 IP 소진 문제를 완화할 수 있지만 모든 문제를 해결할 수는 없습니다. 클러스터에 이미 CG-NAT 공간을 사용하고 있거나 단순히 보조 CIDR을 클러스터 VPC와 연결할 수 없는 경우 대체 CNI를 사용하거나 IPv6 클러스터로 이동하는 등의 다른 옵션을 탐색하는 것이 좋습니다.

Linux용 접두사 모드

Amazon VPC CNI는 Amazon EC2 네트워크 인터페이스에 네트워크 접두사를 할당하여 노드에 사용할 수 있는 IP 주소 수를 늘리고 노드당 포드 밀도를 높입니다. 네트워크 인터페이스에 개별 보조 IP 주소를 할당하는 대신 IPv4 및 IPv6 CIDRs을 할당하도록 Amazon VPC CNI 추가 기능의 버전 1.9.0 이상을 구성할 수 있습니다.

접두사 모드는 IPv6 클러스터에서 기본적으로 활성화되며 지원되는 유일한 옵션입니다. VPC CNI는 ENI의 슬롯에 /80 IPv6 접두사를 할당합니다. 자세한 내용은 이 가이드의 IPv6 섹션을 참조하세요.

접두사 할당 모드에서 인스턴스 유형당 최대 탄력적 네트워크 인터페이스 수는 동일하게 유지되지만, 이제 네트워크 인터페이스의 슬롯에 개별 IPv4 주소를 할당하는 대신 /28(16 IP 주소) IPv4 주소 접두사를 할당하도록 Amazon VPC CNI를 구성할 수 있습니다. `ENABLE_PREFIX_DELEGATION`이 true VPC로 설정된 경우 CNI는 ENI에 할당된 접두사에서 포드에 IP 주소를 할당합니다. 접두사 IP 모드를 활성화하려면 EKS 사용 설명서에 언급된 지침을 따르세요.



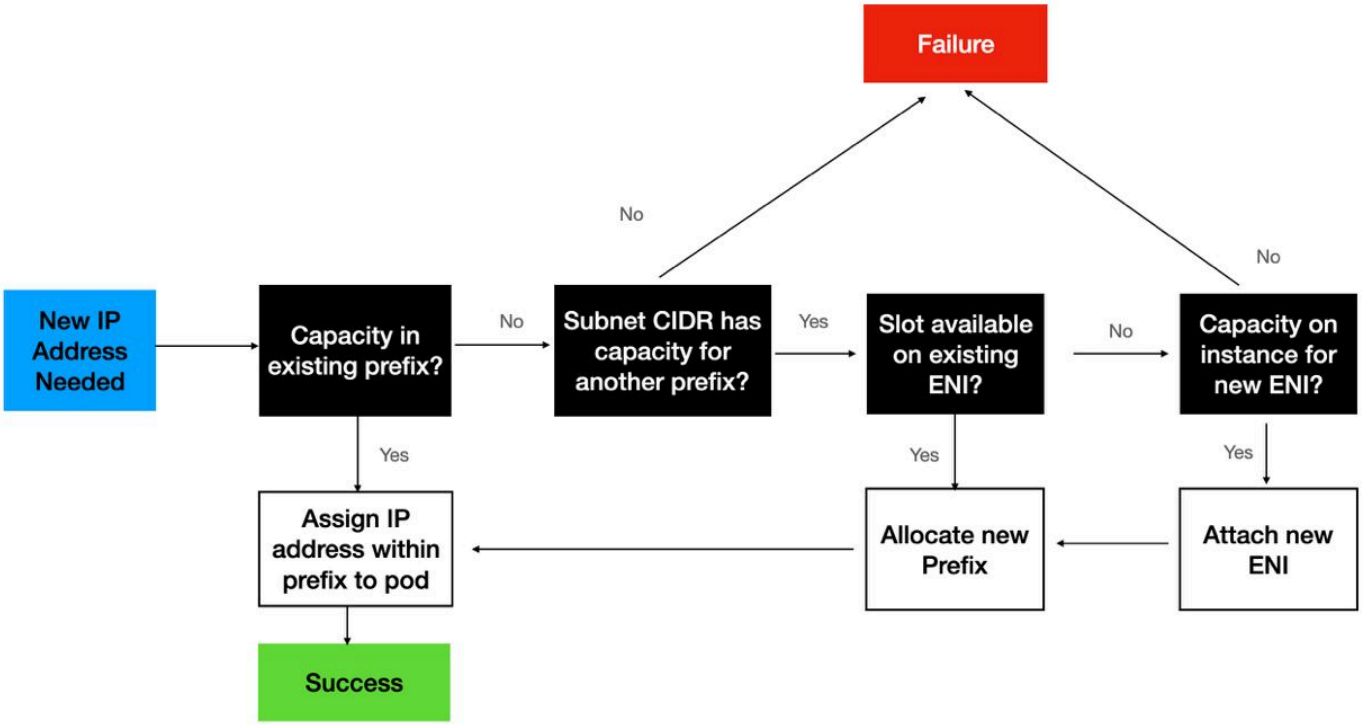
네트워크 인터페이스에 할당할 수 있는 IP 주소의 최대 개수는 인스턴스 유형에 따라 다릅니다. 네트워크 인터페이스에 할당하는 각 접두사는 하나의 IP 주소로 계산됩니다. 예를 들어 `c5.large` 인스턴스에는 네트워크 인터페이스당 `10` 개의 IPv4 주소 제한이 있습니다. 이 인스턴스의 각 네트워크 인터페이스에는 기본 IPv4 주소가 있습니다. 네트워크 인터페이스에 보조 IPv4 주소가 없는 경우 네트워크 인터페이스에 최대 9개의 접두사를 할당할 수 있습니다. 네트워크 인터페이스에 할당한 각 추가 IPv4 주소의 경우 네트워크 인터페이스에 접두사를 하나 더 적게 할당할 수 있습니다. 인스턴스 유형별 네트워크 인터페이스당 IP 주소에 대한 AWS EC2 설명서를 검토하고 네트워크 인터페이스에 접두사를 할당합니다.

작업자 노드를 초기화하는 동안 VPC CNI는 기본 ENI에 하나 이상의 접두사를 할당합니다. CNI는 웜 풀을 유지하여 더 빠른 포드 시작을 위해 접두사를 미리 할당합니다. 환경 변수를 설정하여 웜 풀에 보관할 접두사 수를 제어할 수 있습니다.

- `WARM_PREFIX_TARGET`: 현재 필요를 초과하여 할당할 접두사 수입니다.
- `WARM_IP_TARGET`: 현재 요구 사항을 초과하여 할당할 IP 주소 수입니다.
- `MINIMUM_IP_TARGET`: 언제든지 사용할 수 있는 최소 IP 주소 수입니다.
- `WARM_IP_TARGET` 및를 설정`MINIMUM_IP_TARGET`하면가를 재정의합니다`WARM_PREFIX_TARGET`.

더 많은 포드가 예약되면 기존 ENI에 대한 추가 접두사가 요청됩니다. 먼저 VPC CNI는 기존 ENI에 새 접두사를 할당하려고 시도합니다. ENI가 용량에 도달하면 VPC CNI는 노드에 새 ENI를 할당하려고 시도합니다. 새 ENIs는 최대 ENI 제한(인스턴스 유형으로 정의)에 도달할 때까지 연결됩니다. 새 ENI가

연결되면 ipamd는 , WARM_PREFIX_TARGET WARM_IP_TARGET및 MINIMUM_IP_TARGET 설정을 유지하는 데 필요한 하나 이상의 접두사를 할당합니다.



권장 사항

다음과 같은 경우 접두사 모드 사용

작업자 노드에서 포드 밀도 문제가 발생하는 경우 접두사 모드를 사용합니다. VPC CNI 오류를 방지하려면 접두사 모드로 마이그레이션하기 전에 서브넷에서 /28 접두사의 연속 주소 블록을 검사하는 것이 좋습니다. 서브넷 예약 세부 정보는 " 서브넷 예약을 사용하여 서브넷 조각화 방지(IPv4) " 섹션을 참조하세요.

이전 버전과의 호환성을 위해 최대 포드 제한은 보조 IP 모드를 지원하도록 설정됩니다. 포드 밀도를 높이려면 max-pods 값을 Kubelet 및에 노드의 사용자 데이터--use-max-pods=false로 지정하십시오. max-pod-calculator.sh 스크립트를 사용하여 지정된 인스턴스 유형에 대한 EKS의 권장 최대 포드 수를 계산하는 것이 좋습니다. 예제 사용자 데이터는 EKS 사용 설명서를 참조하세요.

```
./max-pods-calculator.sh --instance-type m5.large --cni-version ``1.9``.0 --cni-prefix-delegation-enabled
```

접두사 할당 모드는 포드에 기본 ENI를 사용하지 않는 CNI 사용자 지정 네트워킹 사용자에게 특히 적합합니다. 접두사 할당을 사용하면 포드에 사용되는 기본 ENI가 없더라도 거의 모든 Nitro 인스턴스 유형에 더 많은 IPs를 연결할 수 있습니다.

다음과 같은 경우 접두사 모드 방지

서브넷이 매우 조각화되어 있고 사용 가능한 IP 주소가 부족하여 /28 접두사를 생성할 수 없는 경우 접두사 모드를 사용하지 마세요. 접두사가 생성되는 서브넷이 조각화된 경우(산란된 보조 IP 주소가 있는 많이 사용되는 서브넷) 접두사 연결이 실패할 수 있습니다. 새 서브넷을 생성하고 접두사를 예약하면이 문제를 피할 수 있습니다.

접두사 모드에서 작업자 노드에 할당된 보안 그룹은 포드에서 공유됩니다. 공유 컴퓨팅 리소스에서 다양한 네트워크 보안 요구 사항이 있는 애플리케이션을 실행하여 규정 준수를 달성해야 하는 보안 요구 사항이 있는 경우 포드용 보안 그룹을 사용하는 것이 좋습니다.

동일한 노드 그룹에서 유사한 인스턴스 유형 사용

노드 그룹에는 여러 유형의 인스턴스가 포함될 수 있습니다. 인스턴스의 최대 포드 수가 낮은 경우 해당 값은 노드 그룹의 모든 노드에 적용됩니다. 노드 그룹의 유사한 인스턴스 유형을 사용하여 노드 사용을 극대화하는 것이 좋습니다. 자동 노드 조정에 Karpenter를 사용하는 경우 프로비저너 API의 요구 사항 부분에서 node.kubernetes.io/instance-type 구성하는 것이 좋습니다.

> **Warning**
>
> 특정 노드 그룹의 모든 노드에 대한 최대 포드 수는 노드 그룹의 단일 인스턴스 유형 중 가장 낮은 최대 포드 수로 정의됩니다.

IPv4 주소를 보존`WARM_PREFIX_TARGET`하도록 구성

에 대한 설치 매니페스트의 기본값은 `1`WARM_PREFIX_TARGET입니다. 대부분의 경우에 대한 권장 값 `1`WARM_PREFIX_TARGET은 인스턴스에 할당된 미사용 IP 주소를 최소화하면서 빠른 포드 시작 시간을 잘 혼합합니다.

노드당 IPv4 주소를 추가로 보존해야 하는 경우 구성 `WARM_PREFIX_TARGET` 시 재정의되는 `WARM_IP_TARGET` 및 `MINIMUM_IP_TARGET` 설정을 사용합니다. `WARM_IP_TARGET`를 16보다 작은 값으로 설정하면 CNI가 전체 초과 접두사를 연결하지 못하도록 할 수 있습니다.

새 ENI를 연결하는 것보다 새 접두사를 할당하는 것이 좋습니다.

기존 ENI에 추가 접두사를 할당하는 것은 새 ENI를 생성하고 인스턴스에 연결하는 것보다 더 빠른 EC2 API 작업입니다. 접두사를 사용하면 IPv4 주소 할당에 신경을 쓰는 동시에 성능이 향상됩니다. 접두사 연결은 일반적으로 1초 이내에 완료되지만 새 ENI 연결에는 최대 10초가 걸릴 수 있습니다. 대부분의 사용 사례에서 CNI는 접두사 모드에서 실행할 때 작업자 노드당 단일 ENI만 필요합니다. 노드당 최대 15IPs를 감당할 수 있는 경우(최악의 경우) 최신 접두사 할당 네트워킹 모드를 사용하고이 모드와 함께 제공되는 성능 및 효율성 이점을 실현하는 것이 좋습니다.

서브넷 조각화를 방지하기 위해 서브넷 예약 사용(IPv4)

EC2가 ENI에 /28 IPv4 접두사를 할당하는 경우 서브넷에서 IP 주소의 연속 블록이어야 합니다. 접두사가 생성된 서브넷이 조각화된 경우(산란된 보조 IP 주소가 있는 많이 사용되는 서브넷) 접두사 연결이 실패할 수 있으며 VPC CNI 로그에 다음과 같은 오류 메시지가 표시됩니다.

```
failed to allocate a private IP/Prefix address: InsufficientCidrBlocks: There are not
 enough free cidr blocks in the specified subnet to satisfy the request.
```

조각화를 방지하고 접두사를 생성하기에 충분한 연속 공간을 확보하려면 VPC 서브넷 CIDR 예약을 사용하여 접두사가 독점적으로 사용할 수 있도록 서브넷 내의 IP 공간을 예약할 수 있습니다. 예약을 생성하면 VPC CNI 플러그인이 EC2 APIs 호출하여 예약 공간에서 자동으로 할당되는 접두사를 할당합니다.

새 서브넷을 생성하고, 접두사를 위한 공간을 예약하고, 해당 서브넷에서 실행되는 작업자 노드에 대해 VPC CNI를 사용하여 접두사 할당을 활성화하는 것이 좋습니다. 새 서브넷이 VPC CNI 접두사 할당이 활성화된 EKS 클러스터에서 실행되는 포드 전용인 경우 접두사 예약 단계를 건너뛸 수 있습니다.

VPC CNI 다운그레이드 방지

접두사 모드는 VPC CNI 버전 1.9.0 이상에서 작동합니다. 접두사 모드가 활성화되고 접두사가 ENIs에 할당되면 Amazon VPC CNI 추가 기능을 1.9.0보다 낮은 버전으로 다운그레이드하지 않아야 합니다. VPC CNI를 다운그레이드하기로 결정한 경우 노드를 삭제하고 다시 생성해야 합니다.

접두사 위임으로 전환하는 동안 모든 노드 교체

기존 작업자 노드를 롤링 교체하는 대신 사용 가능한 IP 주소 수를 늘리려면 새 노드 그룹을 생성하는 것이 좋습니다. 기존 노드를 모두 차단하고 드레이닝하여 기존 포드를 모두 안전하게 제거합니다. 서비스 중단을 방지하려면 중요한 워크로드를 위해 프로덕션 클러스터에 포드 중단 예산을 구현하는 것이

좋습니다. 새 노드의 포드에는 ENI에 할당된 접두사의 IP가 할당됩니다. 포드가 실행 중인지 확인한 후 이전 노드와 노드 그룹을 삭제할 수 있습니다. 관리형 노드 그룹을 사용하는 경우 여기에 언급된 단계에 따라 노드 그룹을 안전하게 삭제하십시오.

Windows용 접두사 모드

Amazon EKS에서는 기본적으로 VPC 리소스 컨트롤러에 의해 Windows 호스트에서 실행되는 각 포드에 보조 IP 주소가 할당됩니다. 이 IP 주소는 호스트의 서브넷에서 할당되는 VPC 라우팅 가능 주소입니다. Linux에서 인스턴스에 연결된 각 ENI에는 보조 IP 주소 또는 /28 CIDR(접두사)로 채울 수 있는 여러 슬롯이 있습니다. 그러나 Windows 호스트는 단일 ENI와 사용 가능한 슬롯만 지원합니다. 보조 IP 주소만 사용하면 할당에 사용할 수 있는 IP 주소가 많더라도 Windows 호스트에서 실행할 수 있는 포드 수를 아티팩트 방식으로 제한할 수 있습니다.

특히 더 작은 인스턴스 유형을 사용할 때 Windows 호스트의 포드 밀도를 높이기 위해 Windows 노드에 대해 접두사 위임을 활성화할 수 있습니다. 접두사 위임이 활성화되면 보조 IP 주소가 아닌 /28 IPv4 접두사가 ENI 슬롯에 할당됩니다. `amazon-vpc-cni` 구성 맵에 `enable-windows-prefix-delegation: "true"` 항목을 추가하여 접두사 위임을 활성화할 수 있습니다. Windows 지원을 활성화하기 위해 `enable-windows-ipam: "true"` 항목을 설정해야 하는 구성 맵과 동일합니다.

Windows 노드에 대해 접두사 위임 모드를 활성화하려면 EKS 사용 설명서에 언급된 지침을 따르세요.

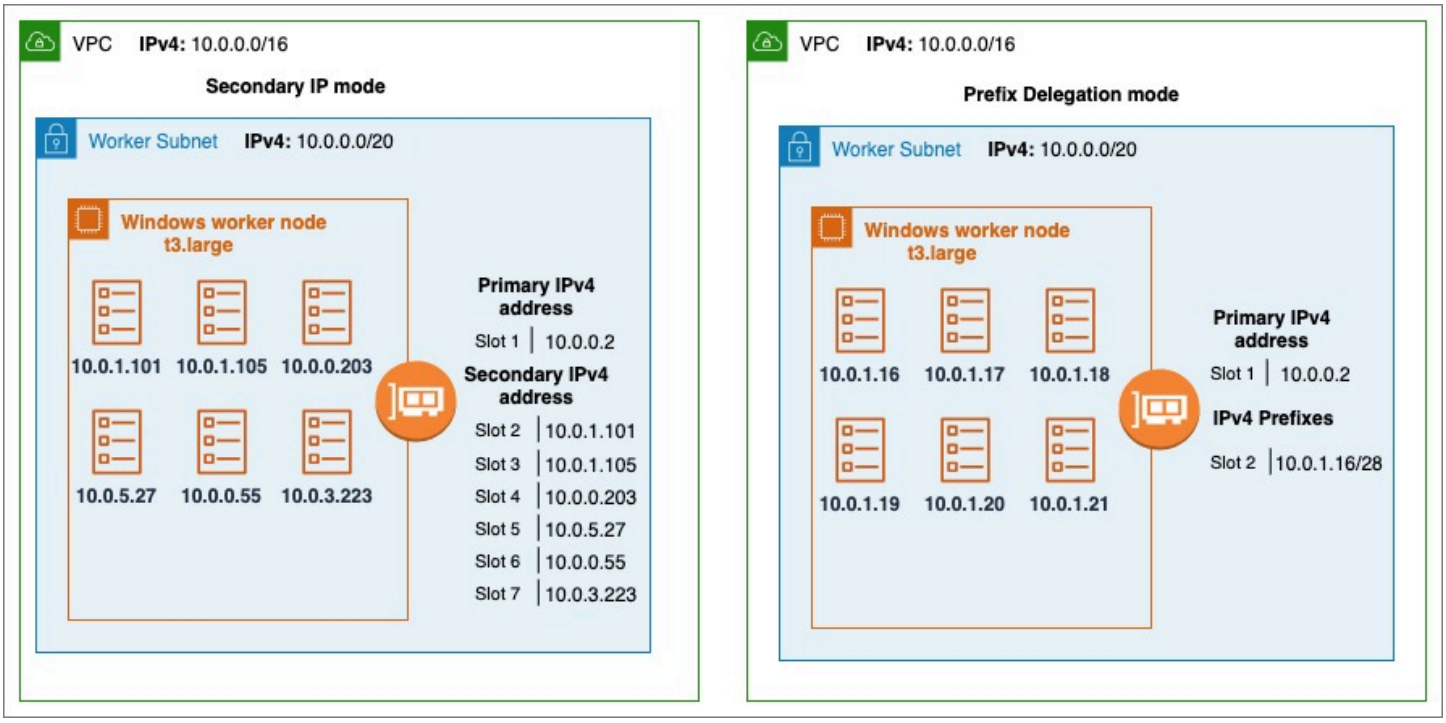


그림: 보조 IP 모드와 접두사 위임 모드 비교

네트워크 인터페이스에 할당할 수 있는 최대 IP 주소 수는 인스턴스 유형과 크기에 따라 다릅니다. 네트워크 인터페이스에 할당된 각 접두사는 사용 가능한 슬롯을 사용합니다. 예를 들어 `c5.large` 인스턴스에는 네트워크 인터페이스당 `10` 슬롯 제한이 있습니다. 네트워크 인터페이스의 첫 번째 슬롯은 항상 인터페이스의 기본 IP 주소에서 사용되므로 접두사 및/또는 보조 IP 주소에 대해 9개의 슬롯이 남아 있습니다. 이러한 슬롯에 접두사가 할당된 경우 노드는 (9 * 16) 144 IP 주소를 지원할 수 있지만 보조 IP 주소가 할당된 경우 9개의 IP 주소만 지원할 수 있습니다. 자세한 내용은 인스턴스 유형별 네트워크 인터페이스당 IP 주소 및 네트워크 인터페이스에 접두사 할당에 대한 설명서를 참조하세요.

작업자 노드를 초기화하는 동안 VPC 리소스 컨트롤러는 IP 주소의 웜 풀을 유지하여 더 빠른 포드 시작을 위해 기본 ENI에 하나 이상의 접두사를 할당합니다. 웜 풀에 보관할 접두사 수는 `amazon-vpc-cni` 구성 맵에서 다음 구성 파라미터를 설정하여 제어할 수 있습니다.

- `warm-prefix-target`: 현재 필요를 초과하여 할당할 접두사 수입니다.
- `warm-ip-target`: 현재 필요를 초과하여 할당할 IP 주소 수입니다.
- `minimum-ip-target`: 언제든지 사용할 수 있는 최소 IP 주소 수입니다.
- `warm-ip-target` 및/또는 설정된 `minimum-ip-target` 경우가를 재정의합니다`warm-prefix-target`.

노드에 더 많은 포드가 예약되면 기존 ENI에 대한 추가 접두사가 요청됩니다. 노드에 포드가 예약되면 VPC 리소스 컨트롤러는 먼저 노드의 기존 접두사에서 IPv4 주소를 할당하려고 시도합니다. 이렇게 할 수 없는 경우 서브넷에 필요한 용량이 있는 한 새 IPv4 접두사가 요청됩니다.

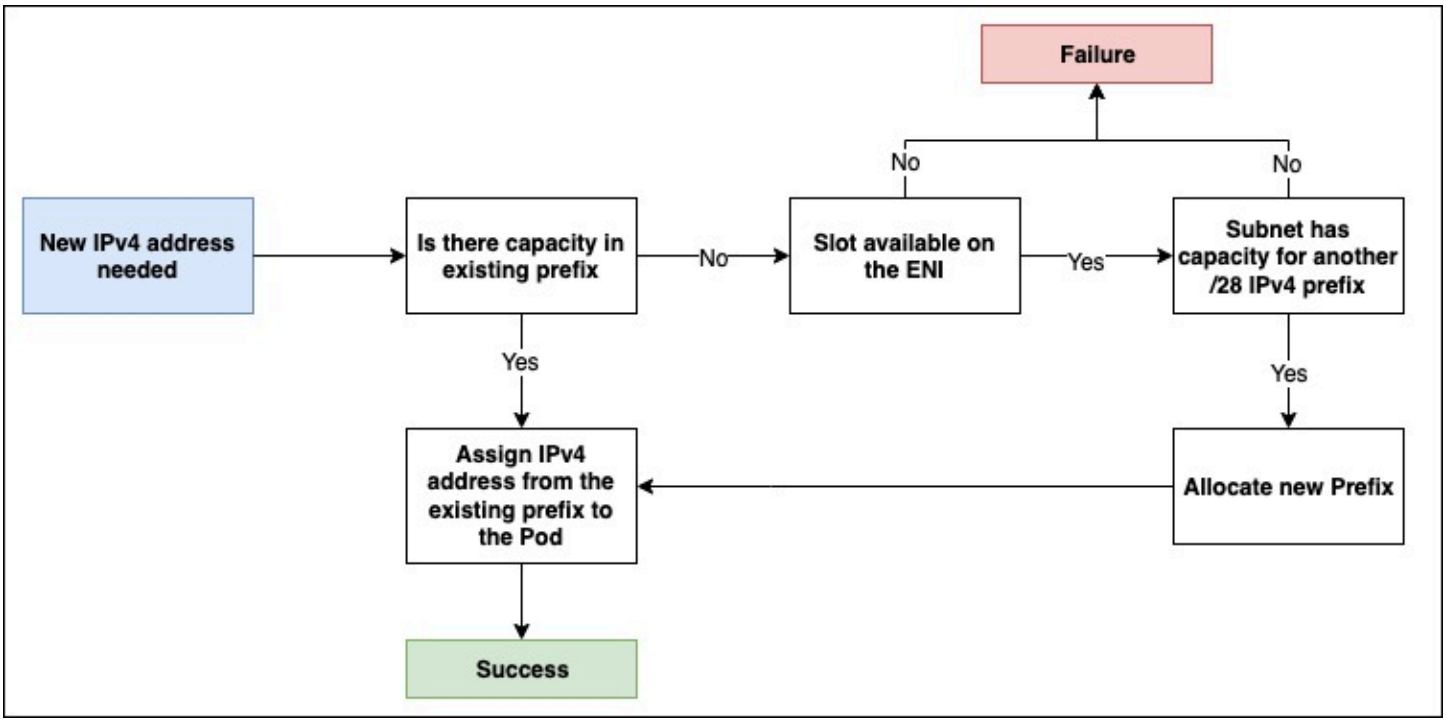


그림: 포드에 IPv4 주소를 할당하는 동안의 워크플로

권장 사항

다음과 같은 경우 접두사 위임 사용

작업자 노드에서 포드 밀도 문제가 발생하는 경우 접두사 위임을 사용합니다. 오류를 방지하려면 접두사 모드로 마이그레이션하기 전에 서브넷에서 /28 접두사의 연속적인 주소 블록을 검사하는 것이 좋습니다. 서브넷 예약 세부 정보는 " 서브넷 예약을 사용하여 서브넷 조각화 방지(IPv4) " 섹션을 참조하세요.

기본적으로 Windows 노드`max-pods`의는 로 설정됩니다`110`. 대부분의 인스턴스 유형에서는 이것으로 충분합니다. 이 제한을 늘리거나 줄이려면 사용자 데이터의 부트스트랩 명령에 다음을 추가합니다.

```
-KubeletExtraArgs '--max-pods=example-value'
```

Windows 노드의 부트스트랩 구성 파라미터에 대한 자세한 내용은 여기의 설명서를 참조하세요.

다음과 같은 경우 접두사 위임 방지

서브넷이 매우 조각화되어 있고 사용 가능한 IP 주소가 부족하여 /28 접두사를 생성할 수 없는 경우 접두사 모드를 사용하지 마세요. 접두사가 생성되는 서브넷이 조각화된 경우(산란된 보조 IP 주소가 있는 많이 사용되는 서브넷) 접두사 연결이 실패할 수 있습니다. 새 서브넷을 생성하고 접두사를 예약하면이 문제를 피할 수 있습니다.

IPv4 주소를 보존하도록 접두사 위임을 위한 파라미터 구성

`warm-prefix-target`, `warm-ip-target`및를 사용하여 사전 크기 조정 및 동적 크기 조정의 동작을 접두사로 미세 조정할 `minimum-ip-target` 수 있습니다. 기본적으로 다음 값이 사용됩니다.

```
warm-ip-target: "1"
minimum-ip-target: "3"
```

이러한 구성 파라미터를 미세 조정하면 IP 주소를 보존하고 IP 주소 할당으로 인한 포드 지연 시간을 줄이는 최적의 균형을 달성할 수 있습니다. 이러한 구성 파라미터에 대한 자세한 내용은 여기의 설명서를 참조하십시오.

서브넷 조각화를 방지하기 위해 서브넷 예약 사용(IPv4)

EC2가 /28 IPv4 접두사를 ENI에 할당하는 경우 서브넷에서 IP 주소의 연속 블록이어야 합니다. 접두사가 생성된 서브넷이 조각화된 경우(산란된 보조 IP 주소가 있는 많이 사용되는 서브넷) 접두사 연결이 실패할 수 있으며 다음 노드 이벤트가 표시됩니다.

```
InsufficientCidrBlocks: The specified subnet does not have enough free cidr blocks to
 satisfy the request
```

조각화를 방지하고 접두사를 생성하기에 충분한 연속 공간을 확보하려면 VPC 서브넷 CIDR 예약을 사용하여 접두사가 독점적으로 사용할 수 있도록 서브넷 내의 IP 공간을 예약합니다. 예약을 생성하면 예약 블록의 IP 주소가 다른 리소스에 할당되지 않습니다. 이렇게 하면 VPC 리소스 컨트롤러는 노드 ENI에 대한 할당 호출 중에 사용 가능한 접두사를 가져올 수 있습니다.

새 서브넷을 생성하고, 접두사를 위한 공간을 예약하고, 해당 서브넷에서 실행되는 작업자 노드에 대한 접두사 할당을 활성화하는 것이 좋습니다. 새 서브넷이 접두사 위임이 활성화된 EKS 클러스터에서 실행되는 포드 전용인 경우 접두사 예약 단계를 건너뛸 수 있습니다.

보조 IP 모드에서 접두사 위임 모드로 또는 그 반대로 마이그레이션할 때 모든 노드 교체

기존 작업자 노드를 롤링 교체하는 대신 사용 가능한 IP 주소 수를 늘리려면 새 노드 그룹을 생성하는 것이 좋습니다.

자체 관리형 노드 그룹을 사용하는 경우 전환 단계는 다음과 같습니다.

- 새 노드가 워크로드를 수용할 수 있도록 클러스터의 용량을 늘립니다.
- Windows에서 접두사 위임 기능 활성화/비활성화
- 모든 기존 노드를 차단하고 드레이닝하여 기존 포드를 모두 안전하게 제거합니다. 서비스 중단을 방지하려면 중요한 워크로드를 위해 프로덕션 클러스터에 포드 중단 예산을 구현하는 것이 좋습니다.
- 포드가 실행 중인지 확인한 후 이전 노드와 노드 그룹을 삭제할 수 있습니다. 새 노드의 포드에는 노드 ENI에 할당된 접두사의 IPv4 주소가 할당됩니다.

관리형 노드 그룹을 사용할 때 전환 단계는 다음과 같습니다.

- Windows에서 접두사 위임 기능 활성화/비활성화
- 여기에 언급된 단계를 사용하여 노드 그룹을 업데이트합니다. 이는 위와 유사한 단계를 수행하지만 EKS에서 관리합니다.

> **Warning**
>
> 동일한 모드의 노드에서 모든 포드 실행

Windows의 경우 보조 IP 모드와 접두사 위임 모드 모두에서 포드를 동시에 실행하지 않는 것이 좋습니다. 이러한 상황은 보조 IP 모드에서 접두사 위임 모드로 마이그레이션하거나 Windows 워크로드를 실행할 때 발생할 수 있습니다.

이는 실행 중인 포드에 영향을 미치지 않지만 노드의 IP 주소 용량과 관련하여 불일치가 있을 수 있습니다. 예를 들어 보조 IPv4 주소에 대해 슬롯이 14개인 t3.xlarge 노드를 가정해 보겠습니다. 포드 10개를 실행하는 경우 보조 IP 주소에서 ENI의 슬롯 10개를 사용합니다. 접두사 위임을 활성화하면 kube-api 서버에 광고되는 용량은 (슬롯 14개 * 접두사당 ip 주소 16개) 244가 되지만 해당 시점의 실제 용량은 (남은 슬롯 4개 * 접두사당 주소 16개) 64가 됩니다. 알려진 용량과 실제 용량(남은 슬롯) 간의 이러한 불일치로 인해 할당할 수 있는 IP 주소보다 많은 포드를 실행하는 경우 문제가 발생할 수 있습니다.

즉, 위에서 설명한 마이그레이션 전략을 사용하여 포드를 보조 IP 주소에서 접두사에서 가져온 주소로 안전하게 전환할 수 있습니다. 모드 간에 전환하면 포드가 정상적으로 계속 실행되며 다음과 같습니다.

- 보조 IP 모드에서 접두사 위임 모드로 전환하면 실행 중인 포드에 할당된 보조 IP 주소가 해제되지 않습니다. 프리 슬롯에 접두사가 할당됩니다. 포드가 종료되면 사용 중인 보조 IP와 슬롯이 릴리스됩니다.
- 접두사 위임 모드에서 보조 IP 모드로 전환하면 범위 내의 모든 IPs가 더 이상 포드에 할당되지 않으면 접두사가 해제됩니다. 접두사의 IP가 포드에 할당된 경우 해당 접두사는 포드가 종료될 때까지 유지됩니다.

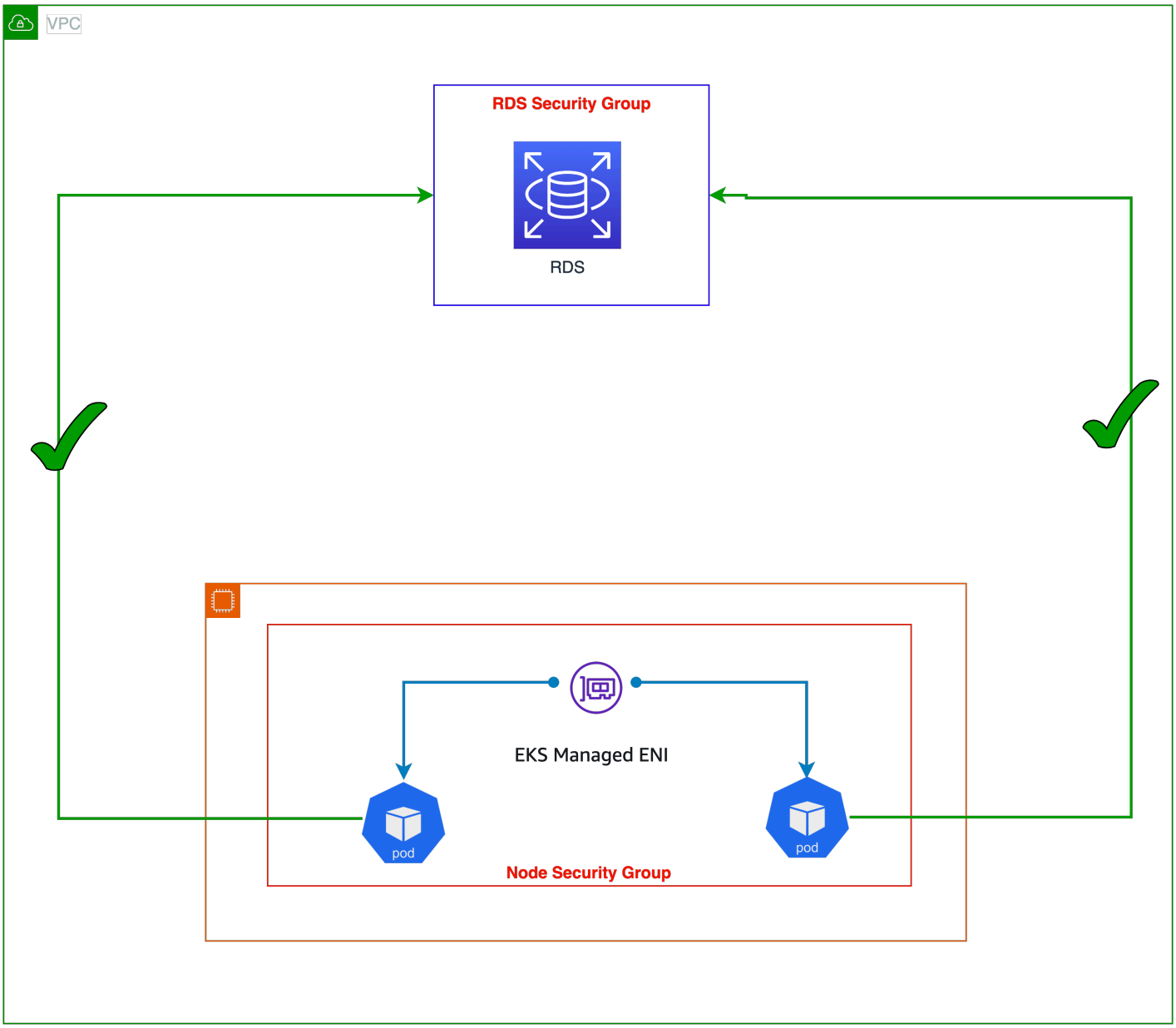
접두사 위임 관련 문제 디버깅

여기에서 디버깅 가이드를 사용하여 Windows에서 접두사 위임과 관련된 문제를 자세히 살펴볼 수 있습니다.

포드당 보안 그룹

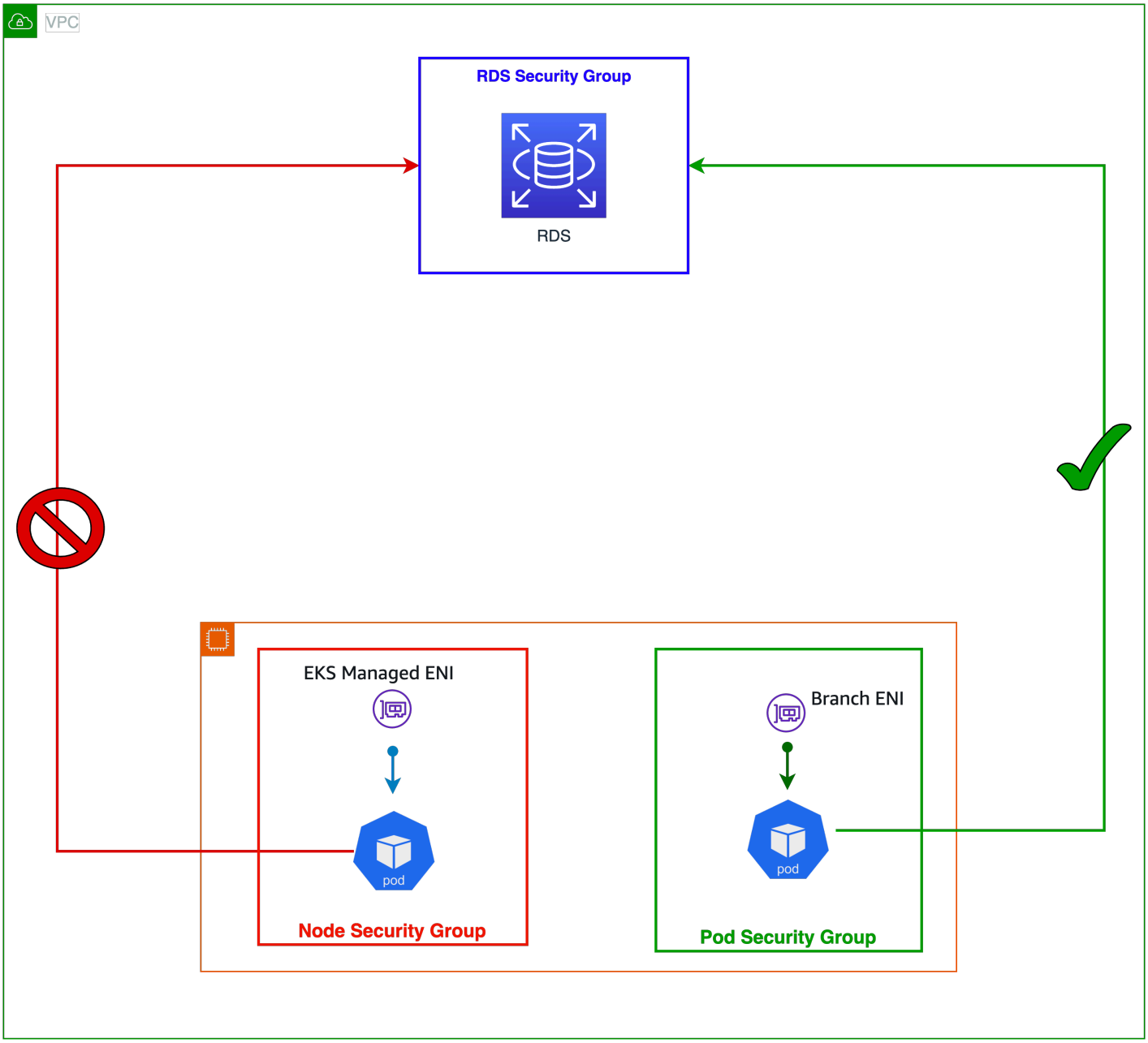
AWS 보안 그룹은 EC2 인스턴스에 대한 가상 방화벽 역할을 하여 인바운드 및 아웃바운드 트래픽을 제어합니다. 기본적으로 Amazon VPC CNI는 노드의 기본 ENI와 연결된 보안 그룹을 사용합니다. 보다 구체적으로, 인스턴스와 연결된 모든 ENI는 동일한 EC2 보안 그룹을 갖게 됩니다. 따라서 노드의 모든 포드는 노드가 실행되는 노드와 동일한 보안 그룹을 공유합니다.

아래 이미지에서 볼 수 있듯이 작업자 노드에서 작동하는 모든 애플리케이션 포드는 RDS 데이터베이스 서비스에 액세스할 수 있습니다(RDS 인바운드가 노드 보안 그룹을 허용함을 고려). 보안 그룹은 노드에서 실행되는 모든 포드에 적용되므로 너무 세분화되어 있습니다. 포드용 보안 그룹은 심층 방어 전략의 필수 부분인 워크로드에 대한 네트워크 세분화를 제공합니다.



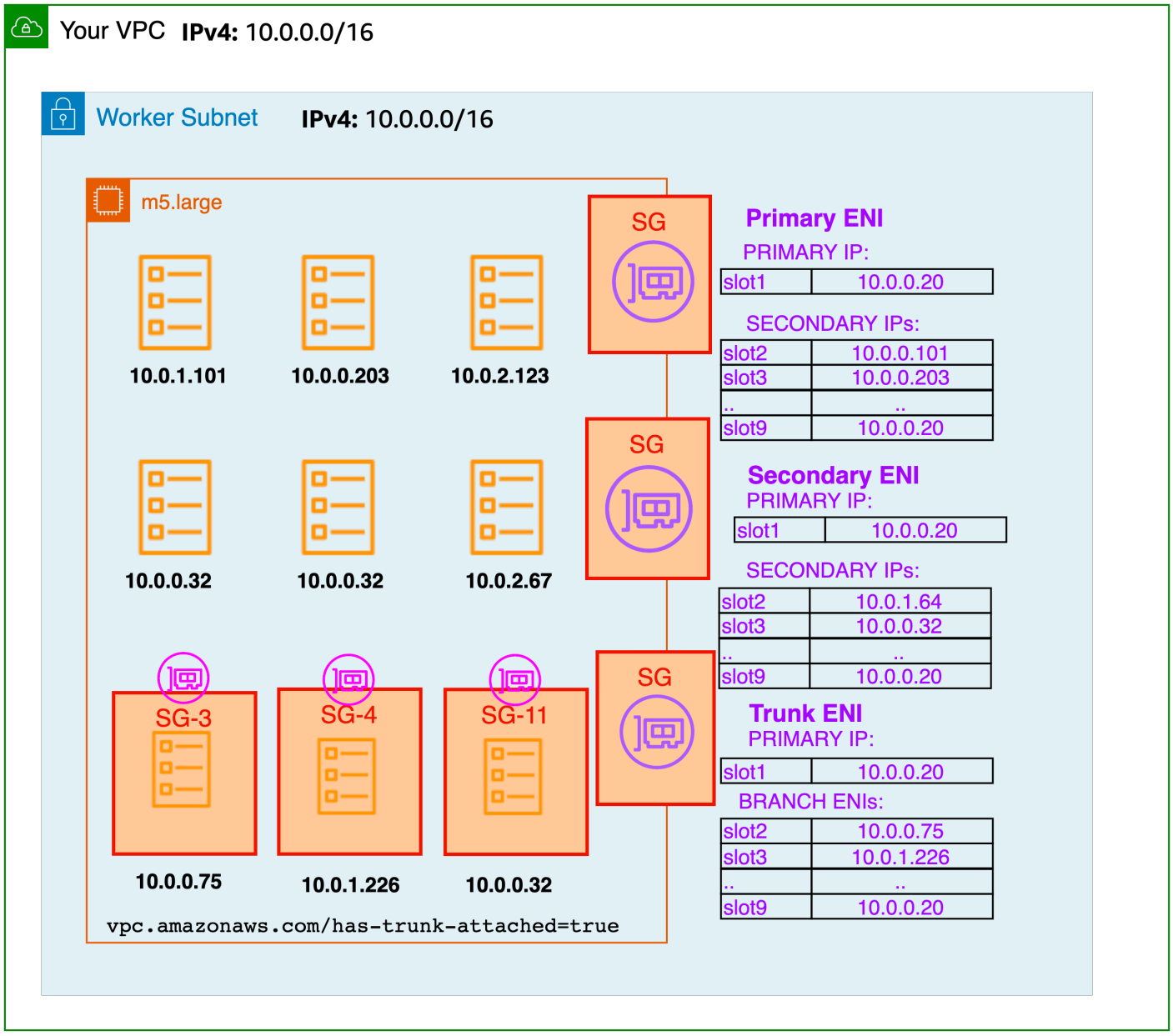
포드용 보안 그룹을 사용하면 공유 컴퓨팅 리소스에서 다양한 네트워크 보안 요구 사항이 있는 애플리케이션을 실행하여 컴퓨팅 효율성을 개선할 수 있습니다. Pod-to-Pod 및 Pod-to-External AWS 서비스와 같은 여러 유형의 보안 규칙을 EC2 보안 그룹을 사용하여 한 곳에서 정의하고 Kubernetes 네이티브 APIs. 아래 이미지는 포드 수준에서 적용된 보안 그룹과 이러한 보안 그룹이 애플리케이션 배포 및

노드 아키텍처를 간소화하는 방법을 보여줍니다. 이제 포드가 Amazon RDS 데이터베이스에 액세스할 수 있습니다.



VPC CNI에를 설정하여 포드ENABLE_POD_ENI=true에 대한 보안 그룹을 활성화할 수 있습니다. 활성화되면 컨트롤 플레인(EKS에서 관리)에서 실행되는 VPC 리소스 컨트롤러는 "`aws-k8s-trunk-eni""라는 트렁크 인터페이스를 생성하여 노드에 연결합니다. 트렁크 인터페이스는 인스턴스에 연결된 표준 네트워크 인터페이스 역할을 합니다. 트렁크 인터페이스를 관리하려면 Amazon EKS 클러스터와 함께 사용되는 클러스터 역할에 AmazonEKSVPCResourceController 관리형 정책을 추가해야 합니다.

또한 컨트롤러는 "aws-k8s-branch-eni"라는 브랜치 인터페이스를 생성하고 트렁크 인터페이스와 연결합니다. 포드에는 SecurityGroupPolicy 사용자 지정 리소스를 사용하여 보안 그룹이 할당되며 브랜치 인터페이스와 연결됩니다. 보안 그룹은 네트워크 인터페이스로 지정되므로 이제 이러한 추가 네트워크 인터페이스에서 특정 보안 그룹이 필요한 포드를 예약할 수 있습니다. 배포 사전 조건을 포함하여 포드의 보안 그룹에 대한 EKS 사용 설명서 섹션을 검토합니다.



브랜치 인터페이스 용량은 보조 IP 주소에 대한 기존 인스턴스 유형 제한에 추가됩니다. 보안 그룹을 사용하는 포드는 최대 포드 공식에서 고려되지 않으며 포드에 보안 그룹을 사용하는 경우 최대 포드 값을 높이는 것을 고려하거나 노드가 실제로 지원할 수 있는 것보다 더 적은 포드를 실행하는 것이 좋습니다.

m5.large에는 최대 9개의 브랜치 네트워크 인터페이스와 표준 네트워크 인터페이스에 최대 27개의 보조 IP 주소가 할당될 수 있습니다. 아래 예제와 같이 m5.large의 기본 최대 포드는 29이며 EKS는 보안 그룹을 사용하는 포드를 최대 포드로 계산합니다. 노드의 최대 포드를 변경하는 방법에 대한 지침은 EKS 사용 설명서를 참조하세요.

포드의 보안 그룹을 사용자 지정 네트워킹과 함께 사용하면 ENIConfig에 지정된 보안 그룹이 아닌 포드의 보안 그룹에 정의된 보안 그룹이 사용됩니다. 따라서 사용자 지정 네트워킹이 활성화되면 포드당 보안 그룹을 사용하는 동안 보안 그룹 순서를 신중하게 평가합니다.

권장 사항

Liveness 프로브에 대한 TCP Early Demux 비활성화

라이브니스 또는 준비 프로브를 사용하는 경우 kubelet이 TCP를 통해 브랜치 네트워크 인터페이스의 포드에 연결할 수 있도록 TCP early demux도 비활성화해야 합니다. 이는 엄격한 모드에서만 필요합니다. 이렇게 하려면 다음 명령을 실행합니다.

```
kubectl edit daemonset aws-node -n kube-system
```

`initContainer` 섹션에서의 값을 `DISABLE_TCP_EARLY_DEMUX` 로 변경합니다. `true`.

포드용 보안 그룹을 사용하여 기존 AWS 구성 투자를 활용합니다.

보안 그룹을 사용하면 RDS 데이터베이스 또는 EC2 인스턴스와 같은 VPC 리소스에 대한 네트워크 액세스를 더 쉽게 제한할 수 있습니다. 포드당 보안 그룹의 한 가지 명확한 이점은 기존 AWS 보안 그룹 리소스를 재사용할 수 있다는 것입니다. 보안 그룹을 네트워크 방화벽으로 사용하여 AWS 서비스에 대한 액세스를 제한하는 경우 브랜치 ENIs를 사용하여 포드에 보안 그룹을 적용하는 것이 좋습니다. EC2 인스턴스에서 EKS로 앱을 전송하고 보안 그룹이 있는 다른 AWS 서비스에 대한 액세스를 제한하는 경우 포드에 대한 보안 그룹을 사용하는 것이 좋습니다.

포드 보안 그룹 적용 모드 구성

Amazon VPC CNI 플러그인 버전 1.11에 라는 새 설정`POD_SECURITY_GROUP_ENFORCING_MODE`(“적용 모드”)이 추가되었습니다. 적용 모드는 포드에 적용되는 보안 그룹과 소스 NAT가 활성화되어 있는지 여부를 모두 제어합니다. 적용 모드를 엄격 또는 표준으로 지정할 수 있습니다. 엄격은가 로 `ENABLE_POD_ENI` 설정된 VPC CNI의 이전 동작을 반영하는 기본값입니다`true`.

엄격한 모드에서는 브랜치 ENI 보안 그룹만 적용됩니다. 소스 NAT도 비활성화됩니다.

표준 모드에서는 기본 ENI 및 브랜치 ENI(포드와 연결됨)와 연결된 보안 그룹이 적용됩니다. 네트워크 트래픽은 두 보안 그룹을 모두 준수해야 합니다.

> **Warning**
>
> 모드 변경은 새로 시작된 포드에만 영향을 미칩니다. 기존 포드는 포드가 생성될 때 구성된 모드를 사용합니다. 고객은 트래픽 동작을 변경하려는 경우 보안 그룹으로 기존 포드를 재활용해야 합니다.

적용 모드: 포드 및 노드 트래픽을 격리하려면 엄격한 모드를 사용합니다.

기본적으로 포드의 보안 그룹은 "엄격 모드"로 설정됩니다. 포드 트래픽을 노드의 나머지 트래픽과 완전히 분리해야 하는 경우이 설정을 사용합니다. 엄격 모드에서는 브랜치 ENI 아웃바운드 보안 그룹을 사용할 수 있도록 소스 NAT가 꺼져 있습니다.

> **Warning**
>
> 엄격 모드가 활성화되면 포드의 모든 아웃바운드 트래픽이 노드에서 나가 VPC 네트워크에 들어갑니다. 동일한 노드의 포드 간 트래픽은 VPC를 통과합니다. 이렇게 하면 VPC 트래픽이 증가하고 노드 기반 기능이 제한됩니다. NodeLocal DNSCache는 엄격한 모드에서 지원되지 않습니다.

적용 모드: 다음 상황에서 표준 모드 사용

포드의 컨테이너에 표시되는 클라이언트 소스 IP

클라이언트 소스 IP를 포드의 컨테이너에 계속 표시해야 하는 경우를 `POD_SECURITY_GROUP_ENFORCING_MODE`로 설정하는 것이 좋습니다`standard`. Kubernetes 서비스는 클라이언트 소스 IP(기본 유형 클러스터)의 보존을 지원하기 위해 externalTrafficPolicy=local을 지원합니다. 이제 표준 모드에서 externalTrafficPolicy가 Local로 설정된 인스턴스 대상을 사용하여 NodePort 및 LoadBalancer 유형의 Kubernetes 서비스를 실행할 수 있습니다.는 클라이언트 소스 IP를 `Local` 보존하고 LoadBalancer 및 NodePort 유형 서비스에 대한 두 번째 홉을 방지합니다.

NodeLocal DNSCache 배포

포드에 보안 그룹을 사용하는 경우 NodeLocal DNSCache를 사용하는 포드를 지원하도록 표준 모드를 구성합니다. NodeLocal DNSCache는 클러스터 노드에서 DNS 캐싱 에이전트를 DaemonSet로 실행하

여 클러스터 DNS 성능을 개선합니다. 이렇게 하면 DNS QPS 요구 사항이 가장 높은 포드가 로컬 캐시가 있는 로컬 kube-dns/CoreDNS를 쿼리하여 지연 시간을 개선하는 데 도움이 됩니다.

노드에 대한 모든 네트워크 트래픽이 VPC에 들어가기 때문에 NodeLocal DNSCache는 엄격한 모드에서 지원되지 않습니다.

Kubernetes 네트워크 정책 지원

연결된 보안 그룹이 있는 포드와 함께 네트워크 정책을 사용할 때는 표준 적용 모드를 사용하는 것이 좋습니다.

클러스터에 속하지 않는 AWS 서비스에 대한 네트워크 수준 액세스를 제한하려면 포드에 대한 보안 그룹을 활용하는 것이 좋습니다. 종종 동부/서부 트래픽이라고 하는 클러스터 내의 포드 간 네트워크 트래픽을 제한하는 네트워크 정책을 고려하세요.

포드당 보안 그룹과의 비호환성 식별

Windows 기반 및 비nitro 인스턴스는 포드에 대한 보안 그룹을 지원하지 않습니다. 포드가 있는 보안 그룹을 활용하려면 인스턴스에 isTrunkingEnabled로 태그를 지정해야 합니다. 포드가 VPC 내부 또는 외부의 AWS 서비스에 의존하지 않는 경우 네트워크 정책을 사용하여 보안 그룹 대신 포드 간 액세스를 관리합니다.

포드당 보안 그룹을 사용하여 AWS 서비스로의 트래픽을 효율적으로 제어

EKS 클러스터 내에서 실행되는 애플리케이션이 RDS 데이터베이스와 같은 VPC 내의 다른 리소스와 통신해야 하는 경우 포드에 SGs를 사용하는 것이 좋습니다. CIDR 또는 DNS 이름을 지정할 수 있는 정책 엔진이 있지만 VPC 내에 엔드포인트가 있는 AWS 서비스와 통신할 때는 최적의 선택이 아닙니다.

반대로 Kubernetes 네트워크 정책은 클러스터 내부와 외부 모두에서 수신 및 송신 트래픽을 제어하는 메커니즘을 제공합니다. 애플리케이션에 다른 AWS 서비스에 대한 종속성이 제한된 경우 Kubernetes 네트워크 정책을 고려해야 합니다. SGs. Kubernetes 네트워크 정책을 사용하여 포드 간(종종 동부/서부 트래픽이라고 함) 및 포드와 외부 서비스 간의 네트워크 트래픽을 제어할 수 있습니다. Kubernetes 네트워크 정책은 OSI 수준 3 및 4에서 구현됩니다.

Amazon EKS를 사용하면 Calico 및 Cilium과 같은 네트워크 정책 엔진을 사용할 수 있습니다. 기본적으로 네트워크 정책 엔진은 설치되지 않습니다. 설정 방법에 대한 지침은 해당 설치 가이드를 참조하십시오. 네트워크 정책을 사용하는 방법에 대한 자세한 내용은 EKS 보안 모범 사례를 참조하세요. DNS 호스트 이름 기능은 네트워크 정책 엔진의 엔터프라이즈 버전에서 사용할 수 있으며, 이는 Kubernetes 서비스/포드와 AWS 외부에서 실행되는 리소스 간의 트래픽을 제어하는 데 유용할 수 있습니다. 또한

기본적으로 보안 그룹을 지원하지 않는 AWS 서비스에 대한 DNS 호스트 이름 지원을 고려할 수 있습니다.

AWS Loadbalancer Controller를 사용하도록 단일 보안 그룹에 태그 지정

포드에 할당된 보안 그룹이 많은 경우 Amazon EKS는 `kubernetes.io/cluster/$name` 공유 또는 소유로 단일 보안 그룹에 태그를 지정할 것을 권장합니다. 태그를 사용하면 AWS Loadbalancer Controller가 보안 그룹의 규칙을 업데이트하여 트래픽을 포드로 라우팅할 수 있습니다. 포드에 보안 그룹이 하나만 지정된 경우 태그 할당은 선택 사항입니다. 보안 그룹에 설정된 권한은 추가적이므로 단일 보안 그룹에 태그를 지정하면 로드밸런서 컨트롤러가 규칙을 찾고 조정할 수 있습니다. 또한 보안 그룹에서 정의한 기본 할당량을 준수하는 데도 도움이 됩니다.

아웃바운드 트래픽에 대한 NAT 구성

보안 그룹이 할당된 포드의 아웃바운드 트래픽에는 소스 NAT가 비활성화됩니다. NAT 게이트웨이 또는 인스턴스로 구성된 프라이빗 서브넷에서 인터넷 시작 작업자 노드에 액세스하고 CNI에서 외부 SNAT를 활성화해야 하는 보안 그룹을 사용하는 포드의 경우.

```
kubectl set env daemonset -n kube-system aws-node AWS_VPC_K8S_CNI_EXTERNALSNAT=true
```

보안 그룹이 있는 포드를 프라이빗 서브넷에 배포

보안 그룹이 할당된 포드는 프라이빗 서브넷에 배포된 노드에서 실행해야 합니다. 퍼블릭 서브넷에 배포된 보안 그룹이 할당된 포드는 인터넷에 액세스할 수 없습니다.

포드 사양 파일에서 terminationGracePeriodSeconds 확인

포드 사양 파일에서가 `0terminationGracePeriodSeconds`이 아닌지 확인합니다(기본값은 30초). 이는 Amazon VPC CNI가 작업자 노드에서 포드 네트워크를 삭제하는 데 필수적입니다. 0으로 설정하면 CNI 플러그인이 호스트에서 포드 네트워크를 제거하지 않으며 브랜치 ENI가 효과적으로 정리되지 않습니다.

Fargate에서 포드에 보안 그룹 사용

Fargate에서 실행되는 포드의 보안 그룹은 EC2 작업자 노드에서 실행되는 포드와 매우 유사하게 작동합니다. 예를 들어 Fargate 포드와 연결하는 SecurityGroupPolicy에서 참조하기 전에 보안 그룹을 생성해야 합니다. 기본적으로 SecurityGroupPolicy를 Fargate 포드에 명시적으로 할당하지 않으면 클러스터 보안 그룹이 모든 Fargate 포드에 연결됩니다. 간소화를 위해 Fagate Pod의 SecurityGroupPolicy에 클러스터 보안 그룹을 추가할 수 있습니다. 그렇지 않으면 보안 그룹에 최소 보안 그룹 규칙을 추가해야 합니다. describe-cluster API를 사용하여 클러스터 보안 그룹을 찾을 수 있습니다.

```
 aws eks describe-cluster --name CLUSTER_NAME --query
 'cluster.resourcesVpcConfig.clusterSecurityGroupId'
```

```
cat >my-fargate-sg-policy.yaml <<EOF
apiVersion: vpcresources.k8s.aws/v1beta1
kind: SecurityGroupPolicy
metadata:
  name: my-fargate-sg-policy
  namespace: my-fargate-namespace
spec:
  podSelector:
    matchLabels:
      role: my-fargate-role
  securityGroups:
    groupIds:
      - cluster_security_group_id
      - my_fargate_pod_security_group_id
EOF
```

여기에 최소 보안 그룹 규칙이 나열되어 있습니다. 이러한 규칙을 통해 Fargate 포드는 kube-apiserver, kubelet 및 CoreDNS와 같은 클러스터 내 서비스와 통신할 수 있습니다. 또한 Fargate 포드와의 인바운드 및 아웃바운드 연결을 허용하는 규칙을 추가해야 합니다. 이렇게 하면 포드가 VPC의 다른 포드 또는 리소스와 통신할 수 있습니다. 또한 Fargate가 Amazon ECR 또는 DockerHub와 같은 다른 컨테이너 레지스트리에서 컨테이너 이미지를 가져오는 규칙을 포함해야 합니다. 자세한 내용은 AWS 일반 참조의 AWS IP 주소 범위를 참조하세요.

아래 명령을 사용하여 Fargate 포드에 적용된 보안 그룹을 찾을 수 있습니다.

```
kubectl get pod FARGATE_POD -o jsonpath='{.metadata.annotations.fargate\.amazonaws
\.com/pod-sg}{"\n"}'
```

위 명령의 eniId를 기록해 둡니다.

```
aws ec2 describe-network-interfaces --network-interface-ids ENI_ID --query
 'NetworkInterfaces[*].Groups[*]'
```

새 보안 그룹을 적용하려면 기존 Fargate 포드를 삭제하고 다시 생성해야 합니다. 예를 들어 다음 명령은 example-app의 배포를 시작합니다. 특정 포드를 업데이트하려면 아래 명령에서 네임스페이스와 배포 이름을 변경할 수 있습니다.

```
kubectl rollout restart -n example-ns deployment example-pod
```

로드 밸런싱

로드 밸런서는 수신 트래픽을 수신하고 EKS 클러스터에서 호스팅되는 의도한 애플리케이션의 대상에 분산합니다. 이렇게 하면 애플리케이션의 복원력이 향상됩니다. EKS 클러스터에 배포하면 AWS Load Balancer 컨트롤러가 해당 클러스터에 대한 AWS Elastic Load Balancer를 생성하고 관리합니다. LoadBalancer 유형의 Kubernetes 서비스가 생성되면 AWS Load Balancer 컨트롤러는 OSI 모델의 계층 4에서 수신된 트래픽을 로드 밸런싱하는 Network Load Balancer(NLB)를 생성합니다. Kubernetes Ingress 객체가 생성되는 동안 AWS Load Balancer Controller는 OSI 모델의 계층 7에서 트래픽을 로드 밸런싱하는 Application Load Balancer(ALB)를 생성합니다.

Load Balancer 유형 선택

AWS Elastic Load Balancing(ELB) 포트폴리오는 Application Load Balancer(ALB), Network Load Balancer(NLB), Gateway Load Balancer(GWLB), Classic Load Balancer(CLB) 등의 로드 밸런서를 지원합니다. 이 모범 사례 섹션에서는 EKS 클러스터와 가장 관련성이 높은 ALB 및 NLB에 중점을 둡니다.

로드 밸런서 유형을 선택할 때 고려해야 할 주요 사항은 워크로드 요구 사항입니다.

자세한 내용과 모든 AWS 로드 밸런서에 대한 참조는 제품 비교를 참조하세요.

워크로드가 HTTP/HTTPS인 경우 Application Load Balancer(ALB)를 선택합니다.

워크로드에 OSI 모델의 계층 7에서 로드 밸런싱이 필요한 경우 AWS Load Balancer Controller를 사용하여 ALB를 프로비저닝할 수 있습니다. 다음 섹션에서 프로비저닝을 다룹니다. ALB는 앞서 언급한 수신 리소스에 의해 제어 및 구성되며 HTTP 또는 HTTPS 트래픽을 클러스터 내의 다른 포드로 라우팅합니다. ALB는 고객에게 애플리케이션 트래픽 라우팅 알고리즘을 변경할 수 있는 유연성을 제공합니다. 기본 라우팅 알고리즘은 미해결 요청 라우팅 알고리즘이 가장 적은 라운드 로빈이기도 합니다.

워크로드가 TCP이거나 워크로드에 클라이언트의 소스 IP 보존이 필요한 경우 Network Load Balancer(NLB)를 선택합니다.

Network Load Balancer는 OSI(Open Systems Interconnection) 모델의 네 번째 계층(전송)에서 작동합니다. TCP 및 UDP 기반 워크로드에 적합합니다. 또한 Network Load Balancer는 기본적으로 포드에 트래픽을 제공할 때 클라이언트 주소의 소스 IP를 보존합니다.

워크로드가 DNS를 활용할 수 없는 경우 Network Load Balancer(NLB)를 선택합니다.

NLB를 사용하는 또 다른 주요 이유는 클라이언트가 DNS를 사용할 수 없는 경우입니다. 이 경우 Network Load Balancer의 IPs가 정적이므로 NLB가 워크로드에 더 적합할 수 있습니다. 클라이언트는 로드 밸런서에 연결할 때 도메인 이름을 IP 주소로 확인할 때 DNS를 사용하는 것이 좋지만, 클라이언트의 애플리케이션이 DNS 확인을 지원하지 않고 하드 코딩된 IPs만 허용하는 경우 IP가 정적 IPs 이고 NLB 수명 동안 동일하게 유지되므로 NLB가 더 적합합니다.

로드 밸런서 프로비저닝

워크로드에 가장 적합한 Load Balancer 결정한 후 고객은 로드 밸런서를 프로비저닝할 수 있는 여러 옵션이 있습니다.

AWS Load Balancer 컨트롤러를 배포하여 Load Balancer 프로비저닝

EKS 클러스터 내에 로드 밸런서를 프로비저닝하는 두 가지 주요 방법이 있습니다.

- AWS 클라우드 공급자에서 서비스 컨트롤러 활용(레거시)
- AWS Load Balancer 컨트롤러 활용(권장)

기본적으로 인트리 컨트롤러라고도 하는 Kubernetes 서비스 컨트롤러는 LoadBalancer 유형의 Kubernetes 서비스 리소스를 조정합니다. 이 컨트롤러는 Kubernetes Cloud Controller Manager 역할을 하는 AWS Cloud Provider 구성 요소에 내장되어 있습니다. https://kubernetes.io/docs/concepts/architecture/cloud-controller/

프로비저닝된 Elastic Load Balancer의 구성은 Kubernetes Service 매니페스트에 추가해야 하는 주석에 의해 제어됩니다. 서비스 컨트롤러와 AWS Load Balancer 컨트롤러에서 사용하는 주석은 다릅니다.

서비스 컨트롤러는 레거시이며 현재 중요한 버그 수정만 수신하고 있습니다. LoadBalancer 유형의 Kubernetes Service를 생성할 때 서비스 컨트롤러는 기본적으로 AWS CLB를 생성하지만 올바른 주석을 사용하는 경우에도 AWS NLB를 생성할 수 있습니다. 서비스 컨트롤러는 Kubernetes Ingress 리소스를 지원하지 않으며 IPv6도 지원하지 않는다는 점에 유의해야 합니다.

EKS 클러스터에서 AWS Load Balancer Controller를 사용하여 Kubernetes Service 및 Ingress 리소스를 조정하는 것이 좋습니다. AWS Load Balancer Controller가 조정 프로세스를 소유하도록 Kubernetes Service 또는 Ingress 매니페스트에 올바른 주석을 사용해야 합니다(서비스 컨트롤러 대신).

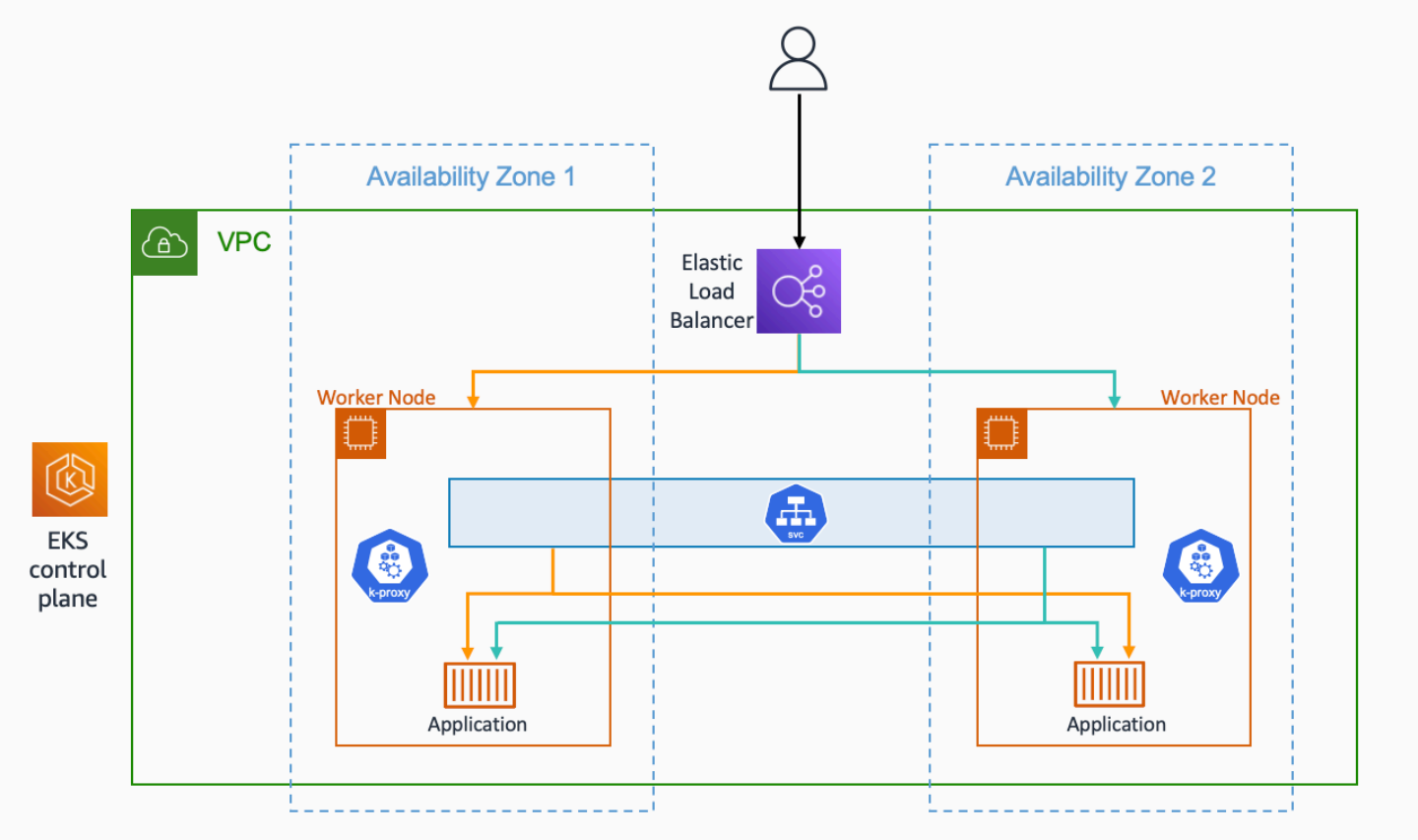
EKS Auto Mode를 사용하는 경우 AWS Load Balancer Controller가 자동으로 제공되므로 설치할 필요가 없습니다.

Load Balancer 대상 유형 선택

IP Target-Type을 사용하여 포드를 대상으로 등록

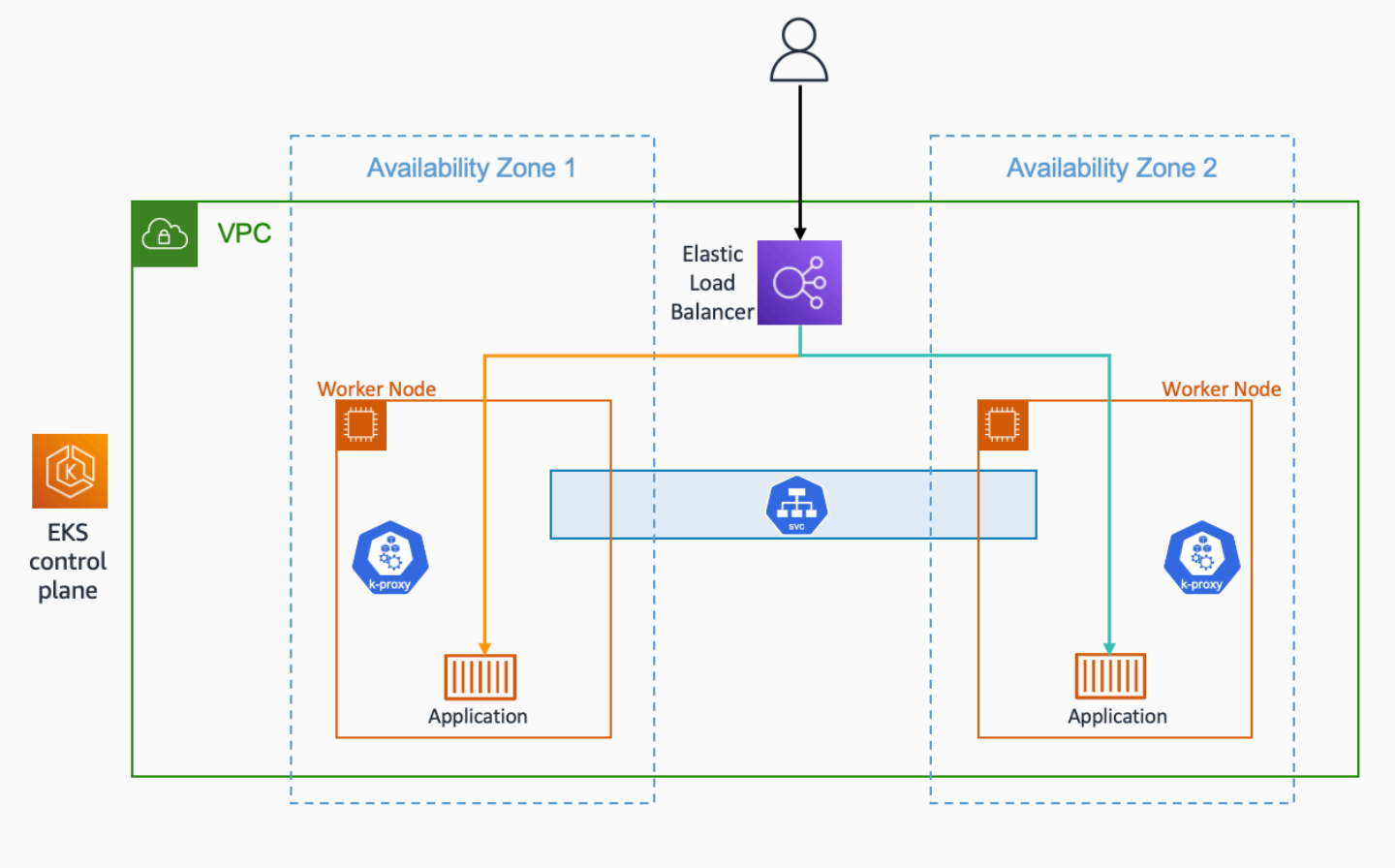
AWS Elastic Load Balancer: 네트워크 및 애플리케이션은 수신된 트래픽을 대상 그룹에 등록된 대상으로 전송합니다. EKS 클러스터의 경우 대상 그룹에 등록할 수 있는 대상에는 두 가지 유형이 있습니다. 인스턴스 및 IP는 어떤 대상 유형이 사용되는지 등록되는 항목과 트래픽이 Load Balancer에서 포드로 라우팅되는 방식에 영향을 미칩니다. 기본적으로 AWS Load Balancer 컨트롤러는 "인스턴스" 유형을 사용하여 대상을 등록하고이 대상은 작업자 노드의 IP 및 NodePort가 되며, 이는 다음과 같습니다.

- Load Balancer의 트래픽은 NodePort, 이는 iptables 규칙(노드에서 실행되는 kube-proxy로 구성됨)에 의해 처리되고 ClusterIP의 서비스(노드에 있음)로 전달됩니다. 마지막으로 서비스는 등록된 포드를 임의로 선택하고 트래픽을 전달합니다. 이 흐름에는 여러 홉이 포함되며 특히 서비스가 다른 AZ에도 있을 수 있는 다른 작업자 노드에서 실행 중인 포드를 선택하기 때문에 추가 지연 시간이 발생할 수 있습니다.
- Load Balancer 작업자 노드를 대상으로 등록하므로 대상으로 전송되는 상태 확인은 포드에서 직접 수신되지 않지만 NodePort의 작업자 노드와 상태 확인 트래픽은 위에서 설명한 것과 동일한 경로를 따릅니다.
- Load Balancer서가 전달하는 트래픽은 포드로 직접 전송되지 않으므로 모니터링 및 문제 해결은 더 복잡하며, 적절한 문제 해결을 위해 패킷의 경로에 대한 전체 end-to-end 가시성을 확보하려면 작업자 노드에서 수신된 패킷을 Service ClusterIP 및 포드와 신중하게 상호 연관시켜야 합니다.



반대로 대상 유형을 "IP"로 구성하면 다음과 같은 영향을 미치는 것이 좋습니다.

- Load Balancer의 트래픽은 포드로 직접 전달되므로 작업자 노드 및 서비스 클러스터 IP의 이전 추가 홉을 우회할 때 네트워크 경로가 간소화되고, 서비스가 다른 AZ의 포드로 트래픽을 전달하고 마지막으로 작업자 노드에서 iptables 규칙 오버헤드 처리를 제거하는 경우 발생할 수 있는 지연 시간이 줄어듭니다.
- Load Balancer서의 상태 확인은 포드에 의해 직접 수신되고 응답됩니다. 즉, 대상 상태 "정상" 또는 "비정상"은 포드의 상태를 직접 나타냅니다.
- 모니터링 및 문제 해결이 더 쉽고 패킷 IP 주소를 캡처하는 데 사용되는 모든 도구는 소스 및 대상 필드의 Load Balancer서와 포드 간의 양방향 트래픽을 직접 표시합니다.



IP 대상을 사용하는 AWS Elastic Load Balancing을 생성하려면 다음을 추가합니다.

- `alb.ingress.kubernetes.io/target-type: ip` Kubernetes Ingress(Application Load Balancer)를 구성할 때 Ingress 매니페스트에 주석 달기
- `service.beta.kubernetes.io/aws-load-balancer-nlb-target-type: ip` LoadBalancer(Network Load Balancer) 유형의 Kubernetes 서비스를 구성할 때 서비스 매니페스트에 주석을 추가합니다.

Load Balancer서 상태 확인 구성

Kubernetes는 자체 상태 확인 메커니즘(다음 단원에서 설명)을 제공하지만 Kubernetes 컨트롤 플레인 외부에서 작동하는 보완적 보호 수단으로 ELB 상태 확인을 구현하는 것이 좋습니다. 이 독립 계층은 다음과 같은 경우에도 애플리케이션을 계속 모니터링합니다.

- Kubernetes 컨트롤 플레인 중단
- 프로브 실행 지연
- kubelet과 포드 간의 네트워크 파티션

위에서 언급한 시나리오 중에 최대 가용성 가속화 복구가 필요한 중요한 워크로드의 경우 ELB 상태 확인은 Kubernetes의 기본 메커니즘 대신 작동하는 필수 안전망을 제공합니다.

ELB에서 상태 확인을 구성하고 미세 조정하려면 서비스 컨트롤러 또는 AWS Load Balancer 컨트롤러에서 조정할 Kubernetes 서비스 또는 수신 매니페스트의 주석을 사용해야 합니다.

가용성 및 포드 수명 주기

애플리케이션 업그레이드 중에 사용자가 가동 중지 시간을 경험하지 않도록 요청을 처리할 수 있도록 애플리케이션을 항상 사용할 수 있는지 확인해야 합니다. 이 시나리오의 일반적인 과제 중 하나는 Kubernetes 계층과 외부 로드 밸런서와 같은 인프라 간에 워크로드의 가용성 상태를 동기화하는 것입니다. 다음 몇 단원에서는 이러한 시나리오를 해결하기 위한 모범 사례를 강조합니다.

> **Note**
>
> 아래 설명은 Kubernetes의 엔드포인트를 대체하는 것이 권장되므로 EndpointSlices를 기반으로 합니다. https://kubernetes.io/docs/concepts/services-networking/service/#endpoints 아래 다루는 시나리오의 맥락에서 두 가지 간의 차이는 무시할 수 있습니다. AWS Load Balancer Controller는 기본적으로 엔드포인트를 사용하므로 컨트롤러에서 enable-endpoint-sliceflag를 활성화하여 EndpointSlices를 활성화할 수 있습니다.

상태 확인 사용

Kubernetes는 기본적으로 프로세스 상태 확인을 실행하여 노드의 kubelet 프로세스가 컨테이너의 기본 프로세스가 실행 중인지 여부를 확인합니다. 그렇지 않으면 기본적으로 해당 컨테이너가 다시 시작됩니다. 그러나 컨테이너 프로세스가 실행 중이지만 교착 상태인 경우 또는 애플리케이션이 성공적으로 시작되었는지 여부를 식별하도록 Kubernetes 프로브를 구성할 수도 있습니다. 프로브는 exec, grpc, httpGet 및 tcpSocket 메커니즘을 기반으로 할 수 있습니다. 프로브의 유형과 결과에 따라 컨테이너를 다시 시작할 수 있습니다.

포드 생성 프로세스의 이벤트 시퀀스를 다시 확인하려면 아래 부록 섹션의 포드 생성을 참조하세요.

준비 프로브 사용

기본적으로 포드 내의 모든 컨테이너가 실행 중인 경우 포드 조건은 "Ready"로 간주됩니다. 그러나 애플리케이션이 여전히 클라이언트 요청을 처리하지 못할 수 있습니다. 예를 들어 애플리케이션은 요청을 처리하기 위해 외부 리소스에서 일부 데이터 또는 구성을 가져와야 할 수 있습니다. 이러한 상태에서는 애플리케이션을 종료하거나 요청을 전달하지 않습니다. 준비 프로브를 사용하면 포드가 "Ready"로 간주되지 않도록 할 수 있습니다. 즉, 프로브 결과가 될 때까지 EndpointSlice 객체에 추

가되지 않습니다`success`. 반면 프로브가 라인에서 더 이상 실패하면 EndpointSlice 객체에서 포드가 제거됩니다. 각 컨테이너의 포드 매니페스트에서 준비 프로브를 구성할 수 있습니다. 각 노드의 `kubelet` 프로세스는 해당 노드의 컨테이너에 대해 준비 프로브를 실행합니다.

포드 준비 게이트 사용

준비 프로브의 한 가지 측면은 외부 피드백/영향 메커니즘이 없다는 것입니다. 노드의 kubelet 프로세스는 프로브를 실행하고 프로브의 상태를 정의합니다. EndpointSlice 컨트롤러는 엔드포인트(포드) 목록을 항상 최신 상태로 유지하기 때문에 Kubernetes 계층(동서 트래픽)의 마이크로서비스 자체 간 요청에 영향을 주지 않습니다. 외부 메커니즘이 필요한 이유와 시기는 무엇입니까?

Kubernetes Service 유형의 Load Balancer 또는 Kubernetes Ingress(북부 - 남부 트래픽용)를 사용하여 애플리케이션을 노출하는 경우 로드 밸런서에 최신 목록 대상도 있도록 해당 Kubernetes Service의 포드 IPs 목록을 외부 인프라 로드 밸런서에 전파해야 합니다. AWS Load Balancer Controller는 여기에 격차를 해소합니다. AWS Load Balancer Controller를 사용하고 `target group: IP` `kube-proxy`를 활용하는 경우 AWS Load Balancer Controller도 업데이트를 수신한 다음(를 통해`watch`) ELB API와 통신하여 ELB에서 포드 IP를 대상으로 등록을 구성하고 시작합니다.

배포의 롤링 업데이트를 수행하면 새 포드가 생성되고 새 포드의 조건이 "준비"가 되는 즉시 이전/기존 포드가 종료됩니다. 이 프로세스 중에 Kubernetes EndpointSlice 객체는 ELB가 새 포드를 대상으로 등록하는 데 걸리는 시간보다 빠르게 업데이트됩니다. 대상 등록을 참조하세요. 잠시 동안 Kubernetes 계층과 클라이언트 요청이 삭제될 수 있는 인프라 계층 간에 상태 불일치가 발생할 수 있습니다. Kubernetes 계층 내의이 기간 동안 새 포드는 요청을 처리할 준비가 되지만 ELB 관점에서는 처리되지 않습니다.

포드 준비 게이트를 사용하면 포드 조건이 "준비"로 간주되기 전에 충족해야 하는 추가 요구 사항을 정의할 수 있습니다. AWS ELB의 경우 AWS Load Balancer 컨트롤러는 AWS ELB에서 대상(포드)의 상태를 모니터링하고 대상 등록이 완료되고 상태가 “정상”으로 바뀌면 컨트롤러는 포드의 조건을 “준비 완료”로 업데이트합니다. 이 접근 방식을 사용하면 AWS ELB의 대상 상태인 외부 네트워크의 상태를 기반으로 포드 조건에 영향을 미칩니다. 포드 준비 게이트는 롤링 업데이트 시나리오에서 매우 중요합니다. 새로 생성된 포드 대상 상태가 AWS ELB에서 "정상"이 될 때까지 배포의 롤링 업데이트가 이전 포드를 종료하는 것을 방지할 수 있기 때문입니다.

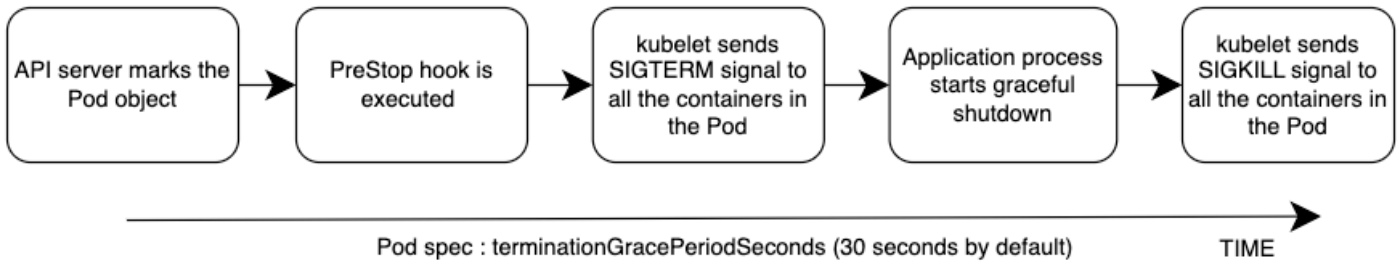
애플리케이션을 정상적으로 종료합니다.

애플리케이션은 클라이언트가 가동 중지 시간을 경험하지 않도록 정상적인 종료를 시작하여 SIGTERM 신호에 응답해야 합니다. 즉, 애플리케이션은 데이터 저장, 파일 설명자 닫기, 데이터베이스 연결 닫기, 진행 중인 요청 완료와 같은 정리 절차를 실행하고 적시에 종료하여 포드 종료 요청을 이행

해야 합니다. 정리가 완료될 수 있도록 유예 기간을 충분히 길게 설정해야 합니다. SIGTERM 신호에 응답하는 방법을 알아보려면 애플리케이션에 사용하는 각 프로그래밍 언어의 리소스를 참조하세요.

SIGTERM 신호를 수신할 때 애플리케이션이 정상적으로 종료되지 않거나 신호를 무시하거나 수신하지 않는 경우, 대신 PreStop 후크를 활용하여 애플리케이션의 정상적인 종료를 시작할 수 있습니다. 프리스톱 후크는 SIGTERM 신호가 전송되기 직전에 실행되며 애플리케이션 코드 자체에서 이러한 작업을 구현하지 않고도 임의 작업을 수행할 수 있습니다.

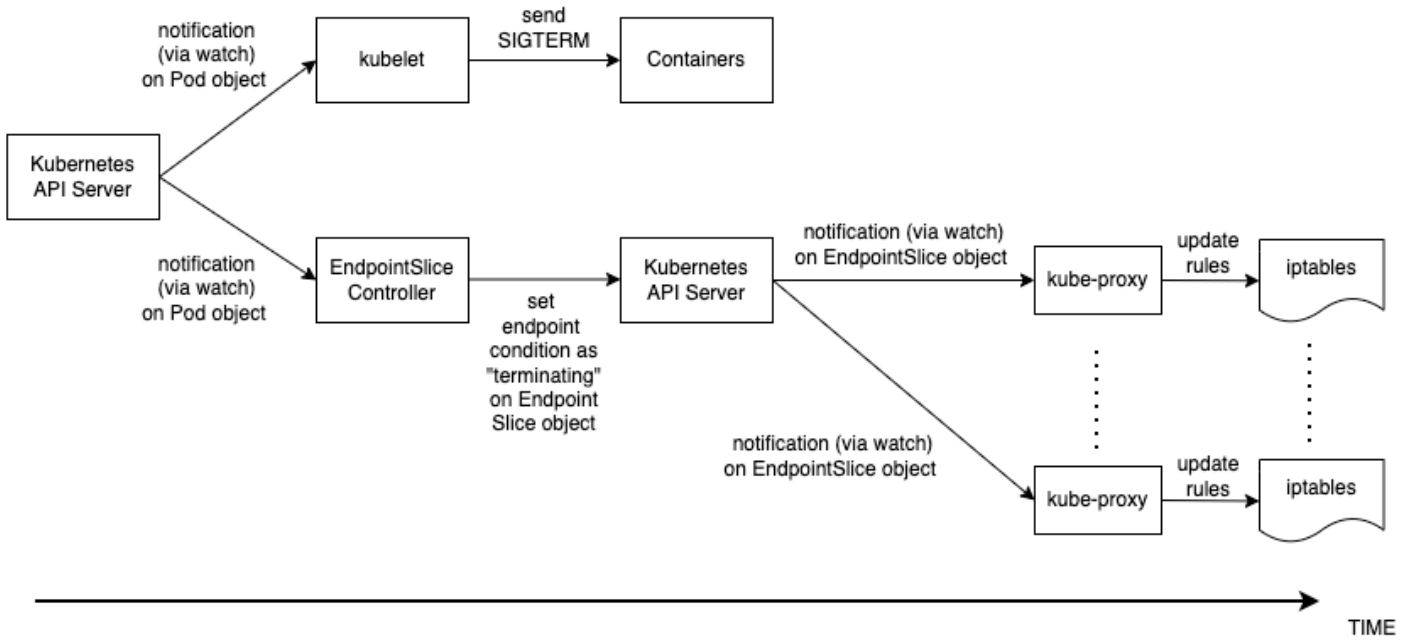
이벤트의 전체 순서는 아래 다이어그램에 나와 있습니다. 참고: 애플리케이션의 정상적인 종료 절차의 결과 또는 PreStop 후크의 결과에 관계없이 애플리케이션 컨테이너는 최종적으로 SIGKILL을 통해 유예 기간이 끝날 때 종료됩니다.



포드 삭제 프로세스의 이벤트 시퀀스를 다시 확인하려면 아래 부록 섹션의 포드 삭제를 참조하세요.

클라이언트 요청을 원활하게 처리

포드 삭제의 이벤트 시퀀스는 포드 생성과 다릅니다. 포드가 생성되면 Kubernetes API에서 포드 IP가 `kubelet` 업데이트되고 EndpointSlice 객체만 업데이트됩니다. 반면 포드가 종료되면 Kubernetes API는 kubelet 및 EndpointSlice 컨트롤러에 동시에 알립니다. 이벤트 시퀀스를 보여주는 다음 다이어그램을 주의 깊게 검사합니다.



상태가 API 서버에서 위에 설명된 노드의 iptables 규칙으로 전파되는 방식은 흥미로운 레이스 조건을 생성합니다. 컨테이너가 각 노드의 kube-proxy가 로컬 iptables 규칙을 업데이트하는 것보다 훨씬 일찍 SIGKILL 신호를 수신할 가능성이 높기 때문입니다. 이러한 경우 언급할 가치가 있는 두 가지 시나리오는 입니다.

- 애플리케이션이 SIGTERM을 수신할 때 진행 중인 요청 및 연결을 즉시 무딘 방식으로 삭제하는 경우 클라이언트는 전체 위치에서 50x 오류를 볼 수 있습니다.
- 애플리케이션이 SIGTERM 수신 시 모든 진행 중인 요청 및 연결이 완전히 처리되도록 하더라도 iptables 규칙이 아직 업데이트되지 않을 수 있으므로 유예 기간 동안 새 클라이언트 요청이 애플리케이션 컨테이너로 계속 전송됩니다. 정리 절차가 컨테이너의 서버 소켓을 닫을 때까지 이러한 새 요청은 새 연결을 초래합니다. 유예 기간이 종료되면 SIGTERM 이후에 설정된 연결은 SIGKILL이 전송되기 때문에 무조건 삭제됩니다.

포드 사양에서 유예 기간을 충분히 길게 설정하면이 문제를 해결할 수 있지만 전파 지연 및 실제 클라이언트 요청 수에 따라 애플리케이션이 연결을 정상적으로 종료하는 데 걸리는 시간을 예상하기 어렵습니다. 따라서 여기서 완벽하지는 않지만 가장 가능한 접근 방식은 PreStop 후크를 사용하여 iptables 규칙이 업데이트될 때까지 SIGTERM 신호를 지연시켜 새 클라이언트 요청이 애플리케이션에 전송되지 않고 기존 연결만 계속되도록 하는 것입니다. PreStop 후크는와 같은 간단한 Exec 핸들러일 수 있습니다`sleep 10`.

위에서 언급한 동작과 권장 사항은 Kubernetes Service 유형의 Load Balancer를 사용하여 애플리케이션을 노출하거나 AWS Load Balancer Controller를 사용하여 Kubernetes Ingress(북부 - 남부 트래픽의 경우)를 노출하고 `target group: IP`를 활용하는 경우에도 동일하게 적용됩니다. `kube-proxy` AWS Load Balancer Controller는 EndpointSlice 객체에 대한 업데이트(시계를 통해)도 수신한 다음 ELB API와 통신하여 ELB에서 포드 IP 등록 취소를 시작하기 때문입니다. 그러나 Kubernetes API 또는 ELB API의 로드에 따라 시간이 걸릴 수 있으며 SIGTERM이 이미 애플리케이션으로 전송되었을 수 있습니다. ELB가 대상 등록 취소를 시작하면 애플리케이션이 새 요청을 수신하지 않도록 해당 대상에 대한 요청 전송을 중지하고 ELB는 기본적으로 300초인 등록 취소 지연도 시작합니다. 등록 취소 프로세스 중에 대상은 기본적으로 ELB가 해당 대상에 대한 진행 중인 요청/기존 연결이 드레이닝될 때까지 대기하는 `draining` 곳입니다. 등록 취소 지연이 만료되면 대상이 사용되지 않고 해당 대상에 대한 진행 중인 요청이 강제로 삭제됩니다.

포드 중단 예산 사용

애플리케이션에 대한 포드 중단 예산(PDB)을 구성합니다. PDBlimits. https://kubernetes.io/docs/concepts/workloads/pods/disruptions/#voluntary-and-involuntary-disruptions 이를 통해 최소 포드 수 또는 백분율을 StatefulSet 또는 배포에서 계속 사용할 수 있습니다. 예를 들어 쿼럼 기반 애플리케이션은 실행 중인 복제본 수가 쿼럼에 필요한 수보다 적지 않도록 해야 합니다. 또는 웹 프런트 엔드를 사용하면 로드를 처리하는 복제본 수가 합계의 특정 백분율 아래로 떨어지지 않도록 할 수 있습니다. PDB는 드레이닝되는 노드 또는 롤아웃되는 배포의 새 버전과 같은 작업으로부터 애플리케이션을 보호합니다. PDB는 노드 운영 체제 장애 또는 네트워크 연결 손실과 같은 비자발적 중단으로부터 애플리케이션을 보호하지 않습니다. 자세한 내용은 Kubernetes에서 애플리케이션에 대한 중단 예산 지정 설명서를 참조하세요.

참조

- KubeCon Europe 2019 세션 - 준비되셨나요? 서비스 상태를 위한 포드 준비 게이트 심층 분석
- 도서 - 실행 중인 Kubernetes
- AWS 블로그 - EKS 기반 ALB를 사용하여 애플리케이션을 빠르게 확장하는 방법(트래픽 손실 없이)

부록

포드 생성

포드가 배포된 후 클라이언트 요청을 수신하고 처리할 준비가 된 시나리오에서 이벤트 시퀀스가 무엇인지 이해하는 것이 중요합니다. 이벤트 시퀀스에 대해 살펴보겠습니다.

1. 포드는 Kubernetes 컨트롤 플레인에 생성됩니다(예: kubectl 명령, 배포 업데이트 또는 조정 작업).

2. kube-scheduler는 클러스터의 노드에 포드를 할당합니다.
3. 할당된 노드에서 실행되는 kubelet 프로세스는 업데이트를 수신하고(를 통해watch) 컨테이너 런타임과 통신하여 포드 사양에 정의된 컨테이너를 시작합니다.
4. 컨테이너가 실행되기 시작하면 kubelet은 Kubernetes API의 포드 객체에서와 같이 포드 조건을 업데이트합니다. Ready
5. EndpointSlice 컨트롤러는 포드 조건 업데이트(를 통해watch)를 수신하고 포드 IP/포트를 각 Kubernetes 서비스의 EndpointSlice 객체(포드 IPs 목록)에 새 엔드포인트로 추가합니다.
6. 각 노드의 kube-proxy 프로세스는 EndpointSlice 객체에 대한 업데이트(를 통해watch)를 수신한 다음 새 포드 IP/포트로 각 노드의 iptables 규칙을 업데이트합니다.

포드 삭제

포드 생성과 마찬가지로 포드 삭제 중에 이벤트 시퀀스가 무엇인지 이해해야 합니다. 이벤트 시퀀스에 대해 살펴보겠습니다.

1. 포드 삭제 요청은 Kubernetes API 서버로 전송됩니다(예: kubectl 명령, 배포 업데이트 또는 조정 작업).
2. Kubernetes API 서버는 포드 객체에서 deletionTimestamp 필드를 설정하여 기본적으로 30초의 유예 기간을 시작합니다. (Grace 기간은를 통해 포드 사양에서 구성할 수 있음terminationGracePeriodSeconds)
3. 노드에서 실행되는 kubelet 프로세스는 포드 객체에 대한 업데이트(감시를 통해)를 수신하고 SIGTERM 신호를 전송하여 해당 포드의 각 컨테이너 내에서 식별자 1(PID 1)을 처리합니다. 그런 다음를 감시합니다terminationGracePeriodSeconds.
4. 또한 EndpointSlice 컨트롤러는 2단계에서 업데이트(를 통해watch)를 수신하고 각 Kubernetes 서비스의 EndpointSlice 객체(포드 IPs 목록)에서 엔드포인트 조건을 "종료"로 설정합니다.
5. 각 노드의 kube-proxy 프로세스는 EndpointSlice 객체에 대한 업데이트(를 통해watch)를 수신한 다음 kube-proxy에서 각 노드의 iptables 규칙을 업데이트하여 클라이언트 요청을 포드로 전달하는 것을 중지합니다.
6. 가 terminationGracePeriodSeconds 만료되면는 포드에 있는 각 컨테이너의 상위 프로세스에 SIGKILL 신호를 kubelet 보내고 강제로 종료합니다.
7. TheEndpointSlice 컨트롤러는 EndpointSlice 객체에서 엔드포인트를 제거합니다.
8. API 서버는 포드 객체를 삭제합니다.

네트워크 성능 문제에 대한 EKS 워크로드 모니터링

CoreDNS 트래픽에서 DNS 제한 문제 모니터링

DNS 집약적 워크로드를 실행하면 DNS 제한으로 인해 간헐적인 CoreDNS 장애가 발생할 수 있으며, 이는 때때로 UnknownHostException 오류가 발생할 수 있는 애플리케이션에 영향을 미칠 수 있습니다.

CoreDNS 배포에는 클러스터의 개별 작업자 노드에서 CoreDNS 인스턴스를 실행하도록 Kubernetes 스케줄러에 지시하는 안티 친화도 정책이 있습니다. 즉, 동일한 작업자 노드에서 복제본을 공동 배치하지 않아야 합니다. 이렇게 하면 각 복제본의 트래픽이 다른 ENI를 통해 라우팅되므로 네트워크 인터페이스당 DNS 쿼리 수가 효과적으로 줄어듭니다. 초당 1024개의 패킷 제한으로 인해 DNS 쿼리가 제한되는 경우 1) CoreDNS 복제본 수를 늘리거나 2) NodeLocal DNSCache를 구현할 수 있습니다. 자세한 내용은 CoreDNS 지표 모니터링을 참조하세요.

챌린지

- 패킷 드롭은 몇 초 만에 발생하며 DNS 제한이 실제로 발생하고 있는지 확인하기 위해 이러한 패턴을 올바르게 모니터링하는 것은 어려울 수 있습니다.
- DNS 쿼리는 탄력적 네트워크 인터페이스 수준에서 제한됩니다. 따라서 제한된 쿼리는 쿼리 로깅에 표시되지 않습니다.
- 흐름 로그는 모든 IP 트래픽을 캡처하지는 않습니다. 예: 인스턴스가 Amazon DNS 서버에 연결할 때 생성한 트래픽. 자체 DNS 서버를 사용하는 경우 해당 DNS 서버에 대한 모든 트래픽이 로깅됩니다.

Solution

작업자 노드에서 DNS 제한 문제를 쉽게 식별할 수 있는 방법은 `linklocal_allowance_exceeded` 지표를 캡처하는 것입니다. linklocal_allowance_exceeded는 로컬 프록시 서비스에 대한 트래픽의 PPS가 네트워크 인터페이스의 최대값을 초과했기 때문에 삭제된 패킷 수입니다. 이는 DNS 서비스, Instance Metadata Service 및 Amazon Time Sync Service에 대한 트래픽에 영향을 미칩니다. 이 이벤트를 실시간으로 추적하는 대신이 지표를 Amazon Managed Service for Prometheus로 스트리밍하고 Amazon Managed Grafana에서 시각화할 수 있습니다.

Conntrack 지표를 사용하여 DNS 쿼리 지연 모니터링

CoreDNS 제한/쿼리 지연을 모니터링하는 데 도움이 될 수 있는 또 다른 지표는 `conntrack_allowance_available` 및 입니다`conntrack_allowance_exceeded`. 연결 추적 허용량을 초과하여 발생한 연결 실패는 다른 허용량을 초과하여 발생한 것보다 더 큰 영향을 미칠 수 있습니다. TCP를 사용하여 데이터를 전송하는 경우 대역폭, PPS 등과 같은 EC2 인스턴스 네트워크 허

용량을 초과하여 대기열에 추가되거나 삭제된 패킷은 일반적으로 TCP의 정체 제어 기능 덕분에 정상적으로 처리됩니다. 영향을 받는 흐름이 느려지고 손실된 패킷이 다시 전송됩니다. 그러나 인스턴스가 연결 추적 허용량을 초과하면 기존 연결 중 일부가 닫혀 새 연결을 위한 공간을 확보할 때까지 새 연결을 설정할 수 없습니다.

`conntrack_allowance_available` 및는 고객이 모든 인스턴스에 따라 달라지는 연결 추적 허용량을 모니터링하는 데 `conntrack_allowance_exceeded` 도움이 됩니다. 이러한 네트워크 성능 지표를 통해 고객은 Network Bandwidth, Packets-Per-Second(PPS), Connections Tracked 및 Link-local 서비스 액세스(Amazon DNS, Instance Meta Data Service, Amazon Time Sync)와 같은 인스턴스의 네트워킹 허용량을 초과할 때 대기 중이거나 삭제된 패킷 수를 파악할 수 있습니다.

`conntrack_allowance_available`는 해당 인스턴스 유형의 연결 추적 허용량(nitro 기반 인스턴스에만 지원됨)에 도달하기 전에 인스턴스에서 설정할 수 있는 추적된 연결 수입니다. `conntrack_allowance_exceeded`는 연결 추적이 인스턴스의 최대값을 초과하여 새 연결을 설정할 수 없기 때문에 삭제된 패킷 수입니다.

기타 중요한 네트워크 성능 지표

기타 중요한 네트워크 성능 지표는 다음과 같습니다.

`bw_in_allowance_exceeded` (지표의 이상적인 값은 0이어야 함)는 인바운드 집계 대역폭이 인스턴스의 최대값을 초과했기 때문에 대기 및/또는 삭제된 패킷 수입니다.

`bw_out_allowance_exceeded` (지표의 이상적인 값은 0이어야 함)는 아웃바운드 집계 대역폭이 인스턴스의 최대값을 초과했기 때문에 대기 및/또는 삭제된 패킷 수입니다.

`pps_allowance_exceeded` (지표의 이상적인 값은 0이어야 함)는 양방향 PPS가 인스턴스의 최대값을 초과했기 때문에 대기 및/또는 삭제된 패킷 수입니다.

지표를 캡처하여 워크로드의 네트워크 성능 문제 모니터링

Elastic Network Adapter(ENA) 드라이버는 활성화된 인스턴스에서 위에서 설명한 네트워크 성능 지표를 게시합니다. 모든 네트워크 성능 지표는 CloudWatch 에이전트를 사용하여 CloudWatch에 게시할 수 있습니다. 자세한 내용은 블로그를 참조하세요.

이제 위에서 설명한 지표를 캡처하여 Amazon Managed Service for Prometheus에 저장하고 Amazon Managed Grafana를 사용하여 시각화해 보겠습니다.

사전 조건

- ethtool - 작업자 노드에 ethtool이 설치되어 있는지 확인합니다.

- AWS 계정에 구성된 AMP 워크스페이스입니다. 지침은 AMP 사용 설명서의 워크스페이스 생성을 참조하세요.
- Amazon Managed Grafana Workspace

Prometheus Node Exporter 배포

작업자 노드에 대한 ethtool 지표를 수집하려면 지표를 수집하고 Prometheus 형식으로 노출하는 수집기를 배포해야 합니다. 다음 명령은 `prometheus-community` Helm 리포지토리를 추가하고 `ethtool` 수집기가 활성화된 `prometheus-node-exporter` 차트를 설치합니다.

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

helm upgrade -i prometheus-node-exporter prometheus-community/prometheus-node-exporter
 \
  --set extraArgs[0]="--collector.ethtool" \
  --set extraArgs[1]="--collector.ethtool.device-include=(eth|em|eno|ens|enp)[0-9s]+" \
  --set extraArgs[2]="--collector.ethtool.metrics-include=.*" \
  --set podAnnotations."prometheus\.io/scrape='true'" \
  --set podAnnotations."prometheus\.io/port='9100'"
```

ADOT 수집기를 배포하여 ethtool 지표를 스크레이프하고 Amazon Managed Service for Prometheus 워크스페이스에 저장

AWS Distro for OpenTelemetry(ADOT)를 설치하는 각 클러스터에는 Amazon Managed Service for Prometheus에 지표를 저장할 수 있는 권한을 AWS 서비스 계정에 부여하려면이 역할이 있어야 합니다. 다음 단계에 따라 IAM 역할을 생성하고 IRSA를 사용하여 이 역할을 Amazon EKS 서비스 계정에 연결합니다.

```
eksctl create iamserviceaccount --name adot-collector --namespace default
 --cluster <CLUSTER_NAME> --attach-policy-arn arn:aws:iam::aws:policy/
AmazonPrometheusRemoteWriteAccess --attach-policy-arn arn:aws:iam::aws:policy/
AWSXrayWriteOnlyAccess --attach-policy-arn arn:aws:iam::aws:policy/
CloudWatchAgentServerPolicy --region <REGION> --approve  --override-existing-
serviceaccounts
```

ADOT 수집기를 배포하여 prometheus ethtool 내보내기에서 메타시를 스크레이프하고 Amazon Managed Service for Prometheus에 저장해 보겠습니다.

다음 절차에서는 배포를 모드 값으로 사용하는 예제 YAML 파일을 사용합니다. 이 모드는 기본 모드이며 독립 실행형 애플리케이션과 유사하게 ADOT Collector를 배포합니다. 이 구성은 샘플 애플리케

이션에서 OTLP 지표를 수신하고 클러스터의 포드에서 스크레이핑된 Amazon Managed Service for Prometheus 지표를 수신합니다.

```
curl -o collector-config-amp.yaml https://raw.githubusercontent.com/aws-observability/
aws-otel-community/master/sample-configs/operator/collector-config-amp.yaml
```

collector-config-amp.yaml에서 다음을 고유한 값으로 바꿉니다.

- 모드: 배포
- serviceAccount: adot-collector
- 엔드포인트: <YOUR_REMOTE_WRITE_ENDPOINT>
- 리전: <YOUR_AWS_REGION>
- 이름: adot-collector

```
kubectl apply -f collector-config-amp.yaml
```

Adot Collector가 배포되면 지표가 Amazon Prometheus에 성공적으로 저장됩니다.

알림을 보내도록 Amazon Managed Service for Prometheus에서 알림 관리자 구성

Amazon Managed Service for Prometheus의 알림 관리자를 사용하여 중요한 알림에 대한 알림 규칙을 설정한 다음 Amazon SNS 주제에 알림을 보낼 수 있습니다. 지금까지 설명한 지표를 확인하도록 기록 규칙과 알림 규칙을 구성해 보겠습니다.

Amazon Managed Service for Prometheus용 ACK 컨트롤러를 사용하여 알림 및 기록 규칙을 프로비저닝합니다.

Amazon Managed Service for Prometheus 서비스에 대한 ACL 컨트롤러를 배포해 보겠습니다.

```
export SERVICE=prometheusservice
export RELEASE_VERSION=`curl -sL https://api.github.com/repos/aws-controllers-k8s/
$SERVICE-controller/releases/latest | grep '"tag_name":' | cut -d'"' -f4`
export ACK_SYSTEM_NAMESPACE=ack-system
export AWS_REGION=us-east-1
aws ecr-public get-login-password --region us-east-1 | helm registry login --username
 AWS --password-stdin public.ecr.aws
helm install --create-namespace -n $ACK_SYSTEM_NAMESPACE ack-$SERVICE-controller \
oci://public.ecr.aws/aws-controllers-k8s/$SERVICE-chart --version=$RELEASE_VERSION --
set=aws.region=$AWS_REGION
```

명령을 실행하면 잠시 후 다음 메시지가 표시됩니다.

```
You are now able to create Amazon Managed Service for Prometheus (AMP) resources!

The controller is running in "cluster" mode.

The controller is configured to manage AWS resources in region: "us-east-1"

The ACK controller has been successfully installed and ACK can now be used to provision
 an Amazon Managed Service for Prometheus workspace.
```

이제 알림 관리자 정의 및 규칙 그룹을 프로비저닝하기 위한 yaml 파일을 생성해 보겠습니다. 아래 파일을 로 저장 `rulegroup.yaml`

```
apiVersion: prometheusservice.services.k8s.aws/v1alpha1
kind: RuleGroupsNamespace
metadata:
   name: default-rule
spec:
   workspaceID: <Your WORKSPACE-ID>
   name: default-rule
   configuration: |
     groups:
     - name: ppsallowance
       rules:
       - record: metric:pps_allowance_exceeded
         expr: rate(node_net_ethtool{device="eth0",type="pps_allowance_exceeded"}[30s])
       - alert: PPSAllowanceExceeded
         expr: rate(node_net_ethtool{device="eth0",type="pps_allowance_exceeded"}
 [30s]) > 0
         labels:
           severity: critical

         annotations:
           summary: Connections dropped due to total allowance exceeding for the
 (instance {{ $labels.instance }})
           description: "PPSAllowanceExceeded is greater than 0"
     - name: bw_in
       rules:
       - record: metric:bw_in_allowance_exceeded
         expr: rate(node_net_ethtool{device="eth0",type="bw_in_allowance_exceeded"}
[30s])
       - alert: BWINAllowanceExceeded
```

```
          expr: rate(node_net_ethtool{device="eth0",type="bw_in_allowance_exceeded"}
 [30s]) > 0
          labels:
            severity: critical

          annotations:
            summary: Connections dropped due to total allowance exceeding for the
 (instance {{ $labels.instance }})
            description: "BWInAllowanceExceeded is greater than 0"
      - name: bw_out
        rules:
        - record: metric:bw_out_allowance_exceeded
          expr: rate(node_net_ethtool{device="eth0",type="bw_out_allowance_exceeded"}
[30s])
        - alert: BWOutAllowanceExceeded
          expr: rate(node_net_ethtool{device="eth0",type="bw_out_allowance_exceeded"}
 [30s]) > 0
          labels:
            severity: critical

          annotations:
            summary: Connections dropped due to total allowance exceeding for the
 (instance {{ $labels.instance }})
            description: "BWoutAllowanceExceeded is greater than 0"
      - name: conntrack
        rules:
        - record: metric:conntrack_allowance_exceeded
          expr: rate(node_net_ethtool{device="eth0",type="conntrack_allowance_exceeded"}
[30s])
        - alert: ConntrackAllowanceExceeded
          expr: rate(node_net_ethtool{device="eth0",type="conntrack_allowance_exceeded"}
 [30s]) > 0
          labels:
            severity: critical

          annotations:
            summary: Connections dropped due to total allowance exceeding for the
 (instance {{ $labels.instance }})
            description: "ConnTrackAllowanceExceeded is greater than 0"
      - name: linklocal
        rules:
        - record: metric:linklocal_allowance_exceeded
          expr: rate(node_net_ethtool{device="eth0",type="linklocal_allowance_exceeded"}
[30s])
```

```
      - alert: LinkLocalAllowanceExceeded
        expr: rate(node_net_ethtool{device="eth0",type="linklocal_allowance_exceeded"}
[30s]) > 0
        labels:
          severity: critical

        annotations:
          summary: Packets dropped due to PPS rate allowance exceeded for local
services  (instance {{ $labels.instance }})
          description: "LinkLocalAllowanceExceeded is greater than 0"
```

WORKSPACE-ID를 사용 중인 워크스페이스의 워크스페이스 ID로 바꿉니다.

이제 알림 관리자 정의를 구성해 보겠습니다. 아래 파이를 로 저장 `alertmanager.yaml`

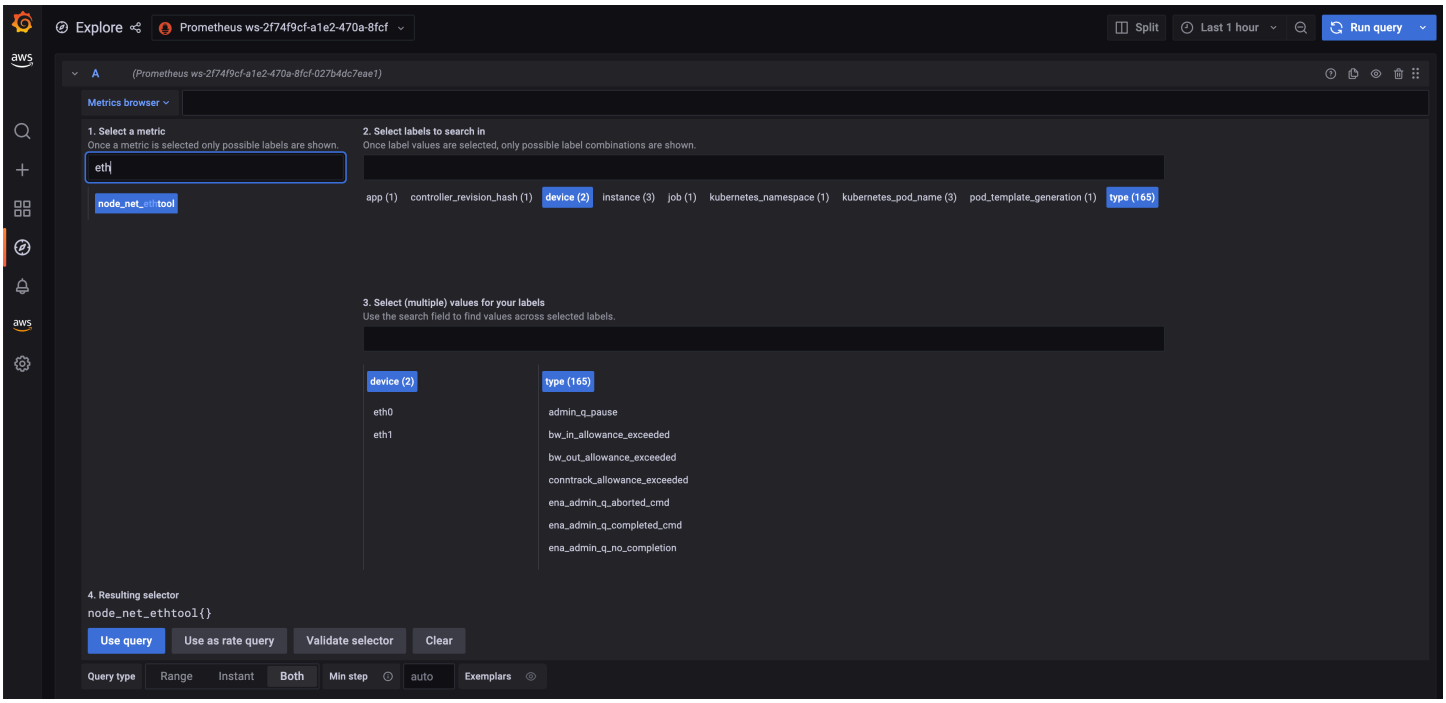
```
apiVersion: prometheusservice.services.k8s.aws/v1alpha1
kind: AlertManagerDefinition
metadata:
  name: alert-manager
spec:
  workspaceID: <Your WORKSPACE-ID >
  configuration: |
    alertmanager_config: |
      route:
         receiver: default_receiver
       receivers:
       - name: default_receiver
          sns_configs:
          - topic_arn: TOPIC-ARN
            sigv4:
              region: REGION
            message: |
              alert_type: {{ .CommonLabels.alertname }}
              event_type: {{ .CommonLabels.event_type }}
```

WORKSPACE-ID를 새 워크스페이스의 Workspace ID로, TOPIC-ARN을 알림을 보내려는 Amazon Simple Notification Service 주제의 ARN으로, REGION을 워크로드의 현재 리전으로 바꿉니다. 워크스페이스에 Amazon SNS로 메시지를 보낼 수 있는 권한이 있는지 확인합니다.

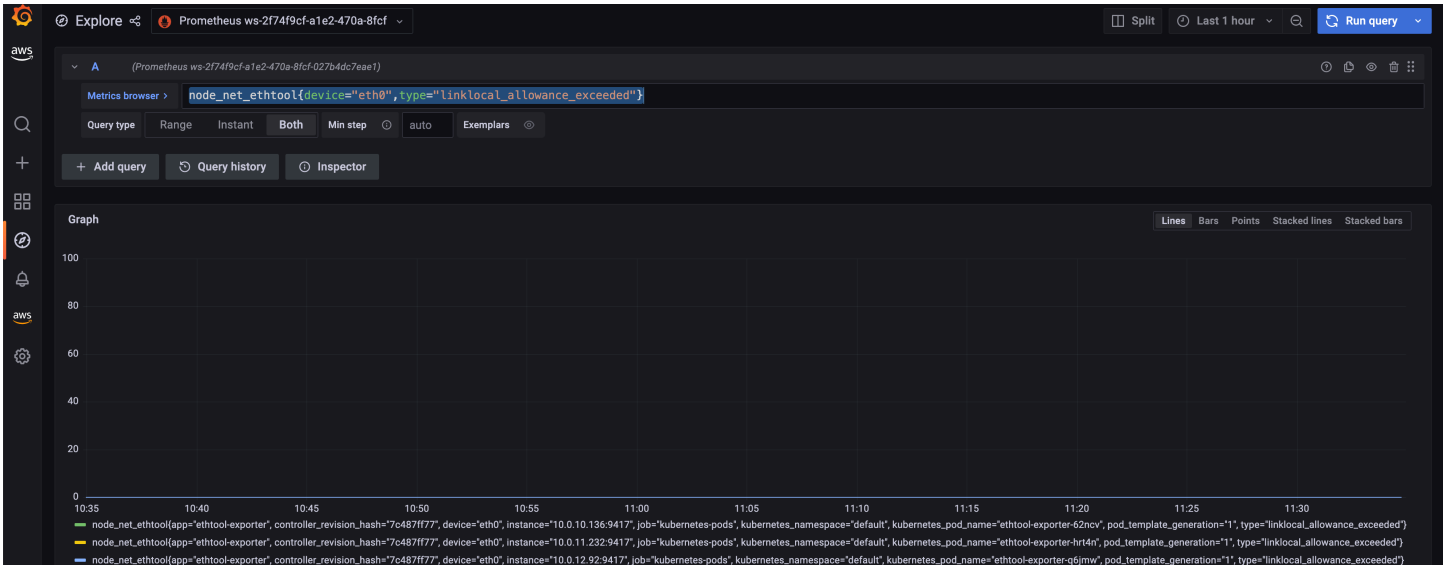
Amazon Managed Grafana에서 ethtool 지표 시각화

Amazon Managed Grafana 내의 지표를 시각화하고 대시보드를 구축해 보겠습니다. Amazon Managed Service for Prometheus를 Amazon Managed Grafana 콘솔 내에서 데이터 소스로 구성합니다. 지침은 Amazon Prometheus를 데이터 소스로 추가를 참조하세요.

이제 Amazon Managed Grafana의 지표를 살펴보겠습니다. 탐색 버튼을 클릭하고 ethtool을 검색합니다.



쿼리를 사용하여 linklocal_allowance_exceeded 지표에 대한 대시보드를 구축해 보겠습니다`rate(node_net_ethtool{device="eth0",type="linklocal_allowance_exceeded"}[30s])`. 그러면 아래 대시보드가 표시됩니다.

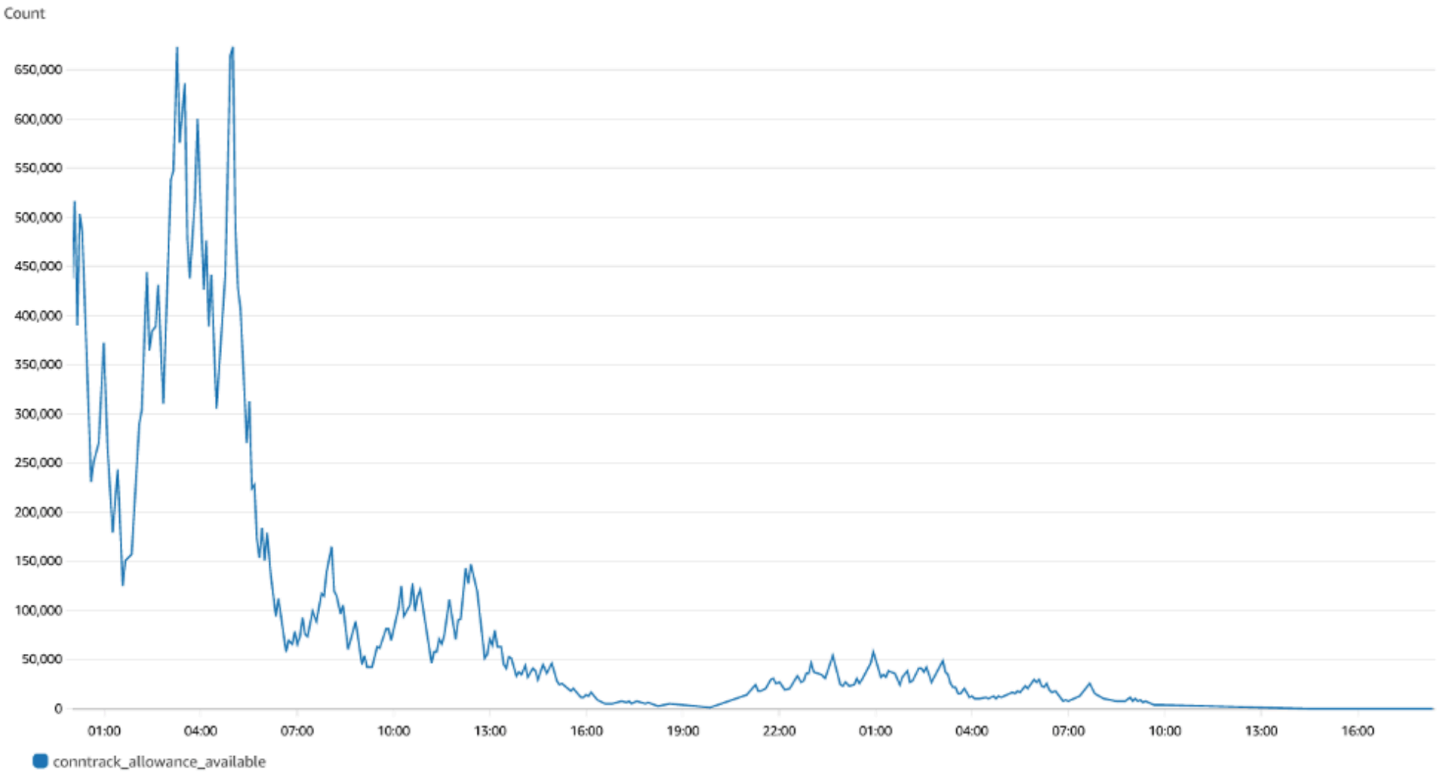


값이 00이므로 삭제된 패킷이 없음을 명확하게 알 수 있습니다.

쿼리를 사용하여 conntrack_allowance_exceeded 지표에 대한 대시보드를 구축해 보겠습니다.`rate(node_net_ethtool{device="eth0",type="conntrack_allowance_exceeded"}[30s])`. 그러면 아래 대시보드가 표시됩니다.



여기에 설명된 대로 Cloudwatch 에이전트를 실행하면 CloudWatch에서 지표를 시각화`conntrack_allowance_exceeded`할 수 있습니다. CloudWatch의 결과 대시보드는 다음과 같습니다.



값이 0이므로 삭제된 패킷이 없음을 명확하게 알 수 있습니다. Nitro 기반 인스턴스를 사용하는 경우에 대한 유사한 대시보드를 생성하고 EC2 인스턴스의 연결을 `conntrack_allowance_available` 사전에 모니터링할 수 있습니다. Amazon Managed Grafana에서 알림을 구성하여 Slack, SNS, Pagerduty 등에 알림을 보내 이를 더욱 확장할 수 있습니다.

IPVS 모드에서 kube-proxy 실행

IP 가상 서버(IPVS) 모드의 EKS는 레거시 iptables 모드에서 실행되는 kube-proxy가 있는 1,000개 이상의 서비스를 사용하여 대규모 클러스터를 실행할 때 자주 나타나는 네트워크 지연 시간 문제를 해결합니다. 이 성능 문제는 각 패킷에 대한 iptables 패킷 필터링 규칙을 순차적으로 처리한 결과입니다. 이 지연 시간 문제는 iptables의 후속인 nftables에서 해결되었습니다. 그러나이 작성 시점을 기준으로 kube-proxy는 여전히 nftables를 사용하기 위해 개발 중입니다. 이 문제를 해결하려면 IPVS 모드에서 실행되도록 클러스터`kube-proxy`를 구성할 수 있습니다.

개요

Kubernetes 버전 1.11 이후 GA였던 IPVS는 선형 검색 대신 해시 테이블을 사용하여 패킷을 처리하므로 수천 개의 노드와 서비스가 있는 클러스터에 효율성을 제공합니다. IPVS는 로드 밸런싱을 위해 설계되었으므로 Kubernetes 네트워킹 성능 문제에 적합한 솔루션입니다.

IPVS는 트래픽을 백엔드 포드로 분산하기 위한 몇 가지 옵션을 제공합니다. 각 옵션에 대한 자세한 내용은 공식 Kubernetes 설명서에서 확인할 수 있지만 간단한 목록은 아래에 나와 있습니다. 라운드 로빈 및 최소 연결은 Kubernetes에서 IPVS 로드 밸런싱 옵션에 가장 많이 사용되는 옵션 중 하나입니다.

```
- rr (Round Robin)
- wrr (Weighted Round Robin)
- lc (Least Connections)
- wlc (Weighted Least Connections)
- lblc (Locality Based Least Connections)
- lblcr (Locality Based Least Connections with Replication)
- sh (Source Hashing)
- dh (Destination Hashing)
- sed (Shortest Expected Delay)
- nq (Never Queue)
```

구현

EKS 클러스터에서 IPVS를 활성화하려면 몇 단계만 필요합니다. 먼저 EKS 작업자 노드 이미지에 Linux 가상 서버 관리 `ipvsadm` 패키지가 설치되어 있는지 확인해야 합니다. Amazon Linux 2023과 같은 Fedora 기반 이미지에이 패키지를 설치하려면 작업자 노드 인스턴스에서 다음 명령을 실행할 수 있습니다.

```
sudo dnf install -y ipvsadm
```

Ubuntu와 같은 Debian 기반 이미지에서 설치 명령은 다음과 같습니다.

```
sudo apt-get install ipvsadm
```

그런 다음 위에 나열된 IPVS 구성 옵션에 대한 커널 모듈을 로드해야 합니다. 이러한 모듈은 재부팅 후에도 유지되도록 `/etc/modules-load.d/` 디렉터리 내부의 파일에 쓰는 것이 좋습니다.

```
sudo sh -c 'cat << EOF > /etc/modules-load.d/ipvs.conf
ip_vs
ip_vs_rr
ip_vs_wrr
ip_vs_lc
ip_vs_wlc
ip_vs_lblc
ip_vs_lblcr
ip_vs_sh
```

```
ip_vs_dh
ip_vs_sed
ip_vs_nq
nf_conntrack
EOF'
```

다음 명령을 실행하여 이미 실행 중인 시스템에 이러한 모듈을 로드할 수 있습니다.

```
sudo modprobe ip_vs
sudo modprobe ip_vs_rr
sudo modprobe ip_vs_wrr
sudo modprobe ip_vs_lc
sudo modprobe ip_vs_wlc
sudo modprobe ip_vs_lblc
sudo modprobe ip_vs_lblcr
sudo modprobe ip_vs_sh
sudo modprobe ip_vs_dh
sudo modprobe ip_vs_sed
sudo modprobe ip_vs_nq
sudo modprobe nf_conntrack
```

Note

사용자 데이터 스크립트를 통해 또는 사용자 지정 작업자 노드 AMI를 빌드하기 위해 실행된 빌드 스크립트에서 작업자 노드의 부트스트래핑 프로세스의 일부로 이러한 작업자 노드 단계를 실행하는 것이 좋습니다.

다음으로 IPVS 모드에서 실행되도록 클러스터의 `kube-proxy` DaemonSet를 구성합니다. 이는 예를 들어 라운드 로빈의 `rr` 경우를 `kube-proxy` `mode` 로 `ipvs` 설정하고 `ipvs scheduler`를 위에 나열된 로드 밸런싱 옵션 중 하나로 설정하여 수행됩니다.

Warning

이는 중단적인 변경이며 업무 외 시간에 수행해야 합니다. 영향을 최소화하려면 초기 EKS 클러스터 생성 중에 이러한 변경을 수행하는 것이 좋습니다.

`kube-proxy` EKS 추가 기능을 업데이트하여 AWS CLI 명령을 실행하여 IPVS를 활성화할 수 있습니다.

```
aws eks update-addon --cluster-name $CLUSTER_NAME --addon-name kube-proxy \
  --configuration-values '{"ipvs": {"scheduler": "rr"}, "mode": "ipvs"}' \
  --resolve-conflicts OVERWRITE
```

또는 클러스터에서 kube-proxy-config ConfigMap을 수정하여이 작업을 수행할 수 있습니다.

```
kubectl -n kube-system edit cm kube-proxy-config
```

아래에서 scheduler 설정을 찾고 값을 위에 나열된 ipvs 로드 밸런싱 옵션 중 하나로 ipvs 설정합니다. 예: rr for Round Robin. 기본값이 인 mode 설정을 찾아 값을 로 iptables변경합니다ipvs. 두 옵션의 결과는 아래 구성과 비슷해야 합니다.

```
  iptables:
    masqueradeAll: false
    masqueradeBit: 14
    minSyncPeriod: 0s
    syncPeriod: 30s
  ipvs:
    excludeCIDRs: null
    minSyncPeriod: 0s
    scheduler: "rr"
    syncPeriod: 30s
  kind: KubeProxyConfiguration
  metricsBindAddress: 0.0.0.0:10249
  mode: "ipvs"
  nodePortAddresses: null
  oomScoreAdj: -998
  portRange: ""
  udpIdleTimeout: 250ms
```

이러한 변경 전에 작업자 노드가 클러스터에 조인된 경우 kube-proxy DaemonSet를 다시 시작해야 합니다.

```
kubectl -n kube-system rollout restart ds kube-proxy
```

검증

작업자 노드 중 하나에서 다음 명령을 실행하여 클러스터 및 작업자 노드가 IPVS 모드에서 실행 중인지 확인할 수 있습니다.

```
sudo ipvsadm -L
```

최소한 아래 결과와 비슷한 결과가 표시되며의 Kubernetes API Server 서비스 `10.100.0.1` 및의 CoreDNS 서비스에 대한 항목이 표시됩니다`10.100.0.10`.

```
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  ip-10-100-0-1.us-east-1. rr
  -> ip-192-168-113-81.us-eas Masq        1      0          0
  -> ip-192-168-162-166.us-ea Masq        1      1          0
TCP  ip-10-100-0-10.us-east-1 rr
  -> ip-192-168-104-215.us-ea Masq        1      0          0
  -> ip-192-168-123-227.us-ea Masq        1      0          0
UDP  ip-10-100-0-10.us-east-1 rr
  -> ip-192-168-104-215.us-ea Masq        1      0          0
  -> ip-192-168-123-227.us-ea Masq        1      0          0
```

Note

이 예제 출력은 서비스 IP 주소 범위가 인 EKS 클러스터에서 가져옵니다`10.100.0.0/16`.

EKS 확장성 모범 사례

이 가이드에서는 EKS 클러스터 크기 조정에 대한 조언을 제공합니다. EKS 클러스터 규모 조정의 목표는 단일 클러스터가 수행할 수 있는 작업량을 극대화하는 것입니다. 하나의 대규모 EKS 클러스터를 사용하면 여러 클러스터를 사용하는 것에 비해 운영 부하를 줄일 수 있지만 다중 리전 배포, 테넌트 격리 및 클러스터 업그레이드와 같은 항목에 대한 장단점이 있습니다. 이 문서에서는 단일 클러스터로 최대 확장성을 달성하는 방법에 중점을 둡니다.

이 설명서의 사용법

이 가이드는 AWS에서 EKS 클러스터를 생성하고 관리하는 개발자와 관리자를 대상으로 합니다. 몇 가지 일반적인 Kubernetes 조정 관행에 초점을 맞추지만 EKS Anywhere를 사용하여 AWS 리전 외부에서 실행되는 자체 관리형 Kubernetes 클러스터에 대한 세부 정보는 없습니다.

각 주제에는 간략한 개요와 대규모 EKS 클러스터 운영에 대한 권장 사항 및 모범 사례가 나와 있습니다. 주제는 특정 순서로 읽을 필요가 없으며 클러스터에서 작동하는지 테스트하고 확인하지 않고 권장 사항을 적용해서는 안 됩니다.

크기 조정 차원 이해

확장성은 성능 및 안정성과 다르므로 클러스터 및 워크로드 요구 사항을 계획할 때 세 가지를 모두 고려해야 합니다. 클러스터가 확장되면 모니터링이 필요하지만이 가이드에서는 모니터링 모범 사례를 다루지 않습니다. EKS는 대규모로 확장할 수 있지만 300개의 노드 또는 5,000개의 포드 이상으로 클러스터를 확장하는 방법을 계획해야 합니다. 이는 절대 숫자는 아니지만이 안내서를 여러 사용자, 엔지니어 및 지원 전문가와 협업하여 얻은 것입니다.

Kubernetes에서의 조정은 다차원적이며 모든 상황에서 작동하는 특정 설정이나 권장 사항은 없습니다. 조정에 대한 지침을 제공할 수 있는 주요 영역은 다음과 같습니다.

EKS 클러스터의 Kubernetes 컨트롤 플레인에는 AWS가 자동으로 실행하고 확장하는 모든 서비스(예: Kubernetes API 서버)가 포함됩니다. 제어 플레인 확장은 AWS의 책임이지만 제어 플레인을 책임감 있게 사용하는 것은 사용자의 책임입니다.

Kubernetes Data Plane 조정은 클러스터 및 워크로드에 필요한 AWS 리소스를 처리하지만 EKS 제어 플레인 외부에 있습니다. EC2 인스턴스, kubelet 및 스토리지를 포함한 리소스는 모두 클러스터 규모에 따라 규모를 조정해야 합니다.

클러스터 서비스는 클러스터 내에서 실행되고 클러스터 및 워크로드에 기능을 제공하는 Kubernetes 컨트롤러 및 애플리케이션입니다. 규정 준수 및 통합을 위해 설치하는 EKS 추가 기능 및 기타 서비스 또는 차트 Helm일 수 있습니다. 이러한 서비스는 워크로드에 따라 달라지는 경우가 많으며 워크로드가 확장됨에 따라 클러스터 서비스가 워크로드에 따라 확장되어야 합니다.

워크로드는 클러스터가 있는 이유이며 클러스터에 따라 수평적으로 확장해야 합니다. Kubernetes에는 클러스터 확장에 도움이 될 수 있는 워크로드의 통합 및 설정이 있습니다. 네임스페이스 및 서비스와 같은 Kubernetes 추상화에는 아키텍처 고려 사항도 있습니다.

엑스트라 라지 스케일링

단일 클러스터를 1,000개의 노드 또는 50,000개의 포드 이상으로 확장하는 경우, 이에 대해 알려 드리겠습니다. 지원 팀 또는 기술 계정 관리자에게 문의하여이 가이드에 제공된 정보 이상으로 계획하고 확장하는 데 도움을 줄 수 있는 전문가에게 문의하는 것이 좋습니다. 온보딩 대상으로 선택된 경우 Amazon EKS는 단일 클러스터에서 최대 100,000개의 노드를 지원할 수 있습니다.

Kubernetes 컨트롤 플레인

Kubernetes 컨트롤 플레인은 Kubernetes API Server, Kubernetes Controller Manager, Scheduler 및 Kubernetes가 작동하는 데 필요한 기타 구성 요소로 구성됩니다. 이러한 구성 요소의 확장성 제한은 클러스터에서 실행 중인 항목에 따라 다르지만 조정에 가장 큰 영향을 미치는 영역에는 Kubernetes 버전, 사용률 및 개별 노드 조정이 포함됩니다.

워크로드 및 노드 버스팅 제한

> **Important**
>
> 컨트롤 플레인의 API 제한에 도달하지 않으려면 클러스터 크기를 한 번에 두 자릿수 백분율씩 늘리는 조정 스파이크를 제한해야 합니다(예: 노드 1,000개에서 노드 1,100개 또는 포드 4,000개에서 포드 4,500개로).

EKS 컨트롤 플레인은 클러스터가 증가함에 따라 자동으로 확장되지만, 확장 속도에는 제한이 있습니다. EKS 클러스터를 처음 생성할 때 컨트롤 플레인은 수백 개의 노드 또는 수천 개의 포드로 즉시 확장할 수 없습니다. EKS의 조정 개선 방식에 대한 자세한 내용은 이 블로그 게시물을 참조하세요.

대규모 애플리케이션을 확장하려면 인프라를 완전히 준비되도록 조정해야 합니다(예: 워밍 로드 밸런서). 조정 속도를 제어하려면 애플리케이션에 적합한 지표를 기반으로 조정해야 합니다. CPU 및 메모

리 조정은 애플리케이션 제약 조건을 정확하게 예측하지 못할 수 있으며 Kubernetes Horizontal Pod Autoscaler(HPA)에서 사용자 지정 지표(예: 초당 요청 수)를 사용하는 것이 더 나은 조정 옵션일 수 있습니다.

사용자 지정 지표를 사용하려면 Kubernetes 설명서의 예제를 참조하세요. 고급 조정 요구 사항이 있거나 외부 소스(예: AWS SQS 대기열)를 기반으로 조정해야 하는 경우 이벤트 기반 워크로드 조정에 KEDA를 사용합니다.

노드 및 포드를 안전하게 축소

장기 실행 인스턴스 교체

노드를 교체하면 추가 가동 시간 후에만 발생하는 문제(예: 느린 메모리 누수)와 구성 드리프트를 방지하여 클러스터가 정기적으로 정상 상태로 유지됩니다. 자동 교체는 노드 업그레이드 및 보안 패치를 위한 좋은 프로세스와 사례를 제공합니다. 클러스터의 모든 노드를 정기적으로 교체하는 경우 지속적인 유지 관리를 위해 별도의 프로세스를 유지 관리하는 데 필요한 부담이 줄어듭니다.

Karpenter의 TTL(Time to Live) 설정을 사용하여 지정된 시간 동안 실행된 인스턴스를 교체합니다. 자체 관리형 노드 그룹은 `max-instance-lifetime` 설정을 사용하여 노드를 자동으로 순환할 수 있습니다. 관리형 노드 그룹에는 현재이 기능이 없지만 GitHub에서 요청을 추적할 수 있습니다.

사용률이 낮은 노드 제거

Kubernetes Cluster Autoscaler에서 `--scale-down-utilization-threshold` 또는 Karpenter에서 `ttlSecondsAfterEmpty` 프로비저닝 설정을 사용하여 스케일 다운 임계값을 사용하여 실행 중인 워크로드가 없는 노드를 제거할 수 있습니다.

포드 중단 예산 및 안전한 노드 종료 사용

Kubernetes 클러스터에서 포드 및 노드를 제거하려면 컨트롤러가 여러 리소스(예: EndpointSlices)를 업데이트해야 합니다. 이렇게 자주 또는 너무 빨리 수행하면 변경 사항이 컨트롤러에 전파될 때 API 서버 제한 및 애플리케이션 중단이 발생할 수 있습니다. 포드 중단 예산은 클러스터에서 노드가 제거되거나 다시 예약될 때 워크로드 가용성을 보호하기 위해 이탈 속도를 늦추는 모범 사례입니다.

Kubectl을 실행할 때 클라이언트 측 캐시 사용

kubectl 명령을 비효율적으로 사용하면 Kubernetes API 서버에 추가 로드를 추가할 수 있습니다. kubectl을 반복적으로(예: 루프용에서) 사용하는 스크립트 또는 자동화를 실행하거나 로컬 캐시 없이 명령을 실행하지 않아야 합니다.

kubectl 에는 필요한 API 호출 양을 줄이기 위해 클러스터에서 검색 정보를 캐싱하는 클라이언트 측 캐시가 있습니다. 캐시는 기본적으로 활성화되어 있으며 10분마다 새로 고쳐집니다.

컨테이너에서 또는 클라이언트 측 캐시 없이 kubectl을 실행하는 경우 API 제한 문제가 발생할 수 있습니다. 불필요한 API 호출을 방지하려면를 탑재하여 클러스터 캐시`--cache-dir`를 유지하는 것이 좋습니다.

kubectl 압축 비활성화

kubeconfig 파일에서 kubectl 압축을 비활성화하면 API 및 클라이언트 CPU 사용량을 줄일 수 있습니다. 기본적으로 서버는 클라이언트로 전송된 데이터를 압축하여 네트워크 대역폭을 최적화합니다. 이렇게 하면 모든 요청에 대해 클라이언트 및 서버에 CPU 부하가 추가되고 적절한 대역폭이 있는 경우 압축을 비활성화하면 오버헤드와 지연 시간을 줄일 수 있습니다. 압축을 비활성화하려면 `--disable-compression=true` 플래그를 사용하거나 kubeconfig 파일에 `disable-compression: true`를 설정할 수 있습니다.

```
apiVersion: v1
clusters:
- cluster:
    server: serverURL
    disable-compression: true
  name: cluster
```

샤드 클러스터 오토스케일러

Kubernetes Cluster Autoscaler는 최대 1,000개의 노드를 확장하도록 테스트되었습니다. 노드가 1,000개 이상인 대규모 클러스터에서는 샤드 모드에서 Cluster Autoscaler의 여러 인스턴스를 실행하는 것이 좋습니다. 각 Cluster Autoscaler 인스턴스는 노드 그룹 집합을 조정하도록 구성됩니다. 다음 예제에서는 각 스케일 4 노드 그룹에 구성된 클러스터 Autoscaling 구성 2개를 보여줍니다.

ClusterAutoscaler-1

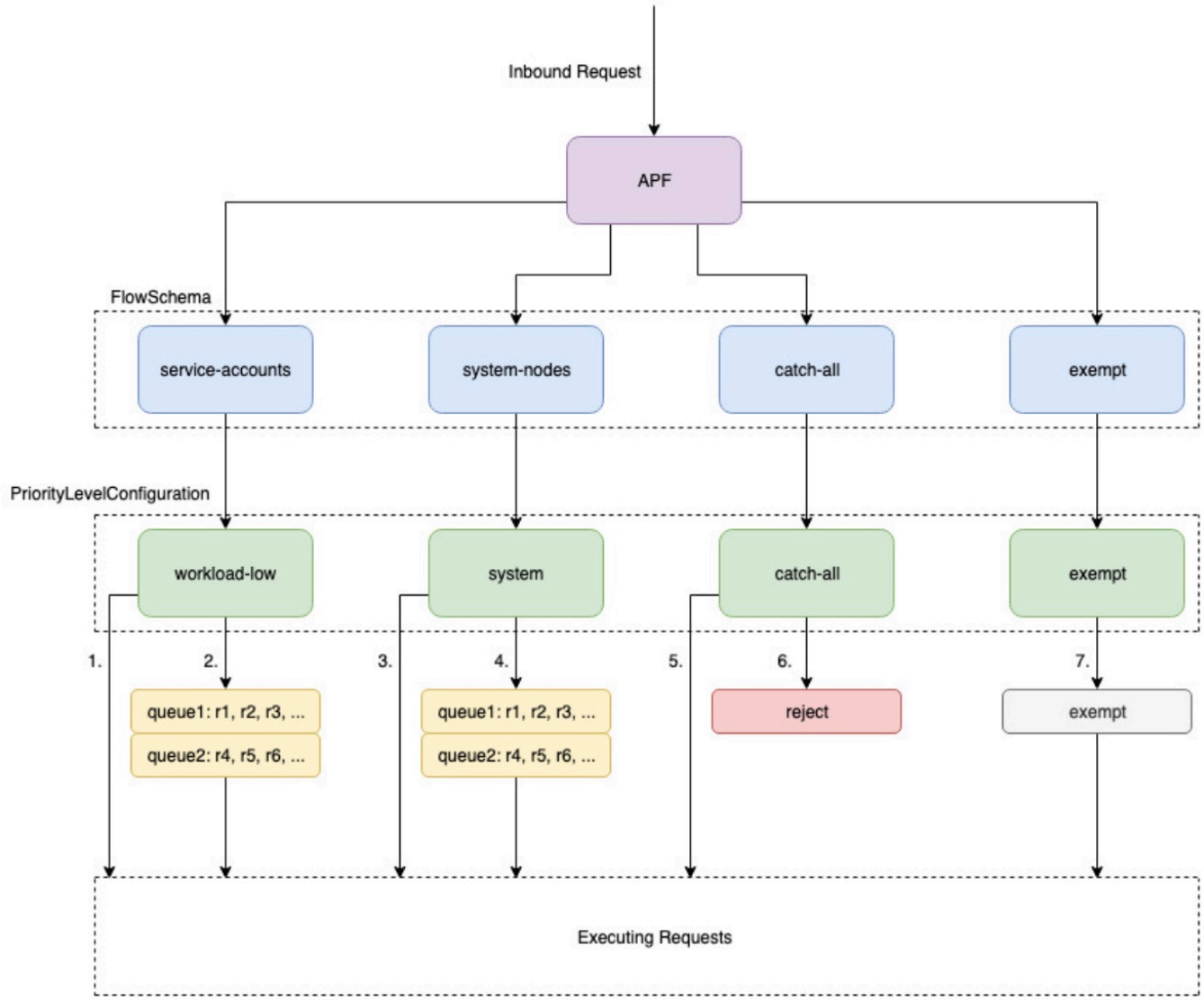
```
autoscalingGroups:
- name: eks-core-node-grp-20220823190924690000000011-80c1660e-030d-476d-cb0d-
d04d585a8fcb
  maxSize: 50
  minSize: 2
- name: eks-data_m1-20220824130553925600000011-5ec167fa-ca93-8ca4-53a5-003e1ed8d306
  maxSize: 450
```

```
  minSize: 2
- name: eks-data_m2-20220824130733258600000015-aac167fb-8bf7-429d-d032-e195af4e25f5
  maxSize: 450
  minSize: 2
- name: eks-data_m3-20220824130553914900000003-18c167fa-ca7f-23c9-0fea-f9edefbda002
  maxSize: 450
  minSize: 2
```

ClusterAutoscaler-2

```
autoscalingGroups:
- name: eks-data_m4-20220824130553925500000000f-5ec167fa-ca86-6b83-ae9d-1e07ade3e7c4
  maxSize: 450
  minSize: 2
- name: eks-data_m5-20220824130744542100000017-02c167fb-a1f7-3d9e-a583-43b4975c050c
  maxSize: 450
  minSize: 2
- name: eks-data_m6-20220824130553924300000000d-9cc167fa-ca94-132a-04ad-e43166cef41f
  maxSize: 450
  minSize: 2
- name: eks-data_m7-20220824130553921000000009-96c167fa-ca91-d767-0427-91c879ddf5af
  maxSize: 450
  minSize: 2
```

API 우선 순위 및 공정성



개요

요청이 증가하는 기간 동안 오버로드되지 않도록 API 서버는 지정된 시간에 미해결 상태일 수 있는 진행 중인 요청 수를 제한합니다. 이 한도를 초과하면 API 서버가 요청 거부를 시작하고 "요청이 너무 많

음"에 대한 429 HTTP 응답 코드를 클라이언트에 반환합니다. 서버가 요청을 삭제하고 클라이언트가 나중에 다시 시도하도록 하는 것은 요청 수에 서버 측 제한이 없고 컨트롤 플레인에 과부하를 가하여 성능이 저하되거나 사용할 수 없게 될 수 있습니다.

Kubernetes가 이러한 진행 중인 요청을 서로 다른 요청 유형으로 나누는 방법을 구성하는 데 사용하는 메커니즘을 API 우선 순위 및 공정성이라고 합니다. API 서버는 `--max-requests-inflight` 및 `--max-mutating-requests-inflight` 플래그로 지정된 값을 합산하여 수락할 수 있는 총 진행 중인 요청 수를 구성합니다. EKS는 이러한 플래그에 대해 400 및 200 요청의 기본값을 사용하므로 지정된 시간에 총 600개의 요청을 디스패치할 수 있습니다. 그러나 사용률 및 워크로드 이탈 증가에 대응하여 컨트롤 플레인을 더 큰 크기로 확장하면 이에 따라 2000년까지 진행 중인 요청 할당량이 증가합니다(변경될 수 있음). APF는 이러한 진행 중인 요청 할당량을 서로 다른 요청 유형으로 더 세분화하는 방법을 지정합니다. EKS 컨트롤 플레인은 각 클러스터에 등록된 API 서버가 2개 이상 있는 고가용성입니다. 즉, 클러스터가 처리할 수 있는 총 기내 요청 수가 kube-apiserver당 설정된 기내 할당량의 두 배(또는 수평적으로 더 확장된 경우 더 높음)입니다. 이는 가장 큰 EKS 클러스터에서 초당 수천 개의 요청에 해당합니다.

PriorityLevelConfigurations 및 FlowSchemas라는 두 종류의 Kubernetes 객체는 총 요청 수를 서로 다른 요청 유형으로 나누는 방법을 구성합니다. 이러한 객체는 API Server에서 자동으로 유지 관리되며 EKS는 지정된 Kubernetes 마이너 버전에 대해 이러한 객체의 기본 구성을 사용합니다. PriorityLevelConfigurations는 허용된 총 요청 수의 일부를 나타냅니다. 예를 들어 워크로드가 높은 PriorityLevelConfiguration에는 총 600개의 요청 중 98개가 할당됩니다. 모든 PriorityLevelConfigurations에 할당된 요청의 합계는 600과 같습니다(또는 지정된 레벨에 요청의 일부가 부여되면 API 서버가 반올림되기 때문에 600보다 약간 높음). 클러스터의 PriorityLevelConfigurations와 각 클러스터에 할당된 요청 수를 확인하려면 다음 명령을 실행할 수 있습니다. 다음은 EKS 1.32의 기본값입니다.

```
$ kubectl get --raw /metrics | grep apiserver_flowcontrol_nominal_limit_seats
apiserver_flowcontrol_nominal_limit_seats{priority_level="catch-all"} 13
apiserver_flowcontrol_nominal_limit_seats{priority_level="exempt"} 0
apiserver_flowcontrol_nominal_limit_seats{priority_level="global-default"} 49
apiserver_flowcontrol_nominal_limit_seats{priority_level="leader-election"} 25
apiserver_flowcontrol_nominal_limit_seats{priority_level="node-high"} 98
apiserver_flowcontrol_nominal_limit_seats{priority_level="system"} 74
apiserver_flowcontrol_nominal_limit_seats{priority_level="workload-high"} 98
apiserver_flowcontrol_nominal_limit_seats{priority_level="workload-low"} 245
```

두 번째 유형의 객체는 FlowSchemas입니다. 지정된 속성 집합이 있는 API Server 요청은 동일한 FlowSchema로 분류됩니다. 이러한 속성에는 인증된 사용자 또는 API 그룹, 네임스페이스 또는 리소스와 같은 요청의 속성이 포함됩니다. FlowSchema는이 유형의 요청을 매핑해야 하는

PriorityLevelConfiguration도 지정합니다. 두 객체는 함께 "이 유형의 요청이이 진행 중인 요청 공유에 포함되기를 원합니다."라고 말합니다. 요청이 API 서버에 도달하면 필요한 모든 속성과 일치하는 FlowSchemas를 확인합니다. 여러 FlowSchemas가 요청과 일치하는 경우 API 서버는 객체의 속성으로 지정된 가장 작은 일치 우선 순위로 FlowSchema를 선택합니다.

FlowSchemas를 PriorityLevelConfigurations에 매핑하는 방법은 다음 명령을 사용하여 확인할 수 있습니다.

```
$ kubectl get flowschemas
NAME                           PRIORITYLEVEL     MATCHINGPRECEDENCE
 DISTINGUISHERMETHOD   AGE     MISSINGPL
exempt                         exempt            1                  <none>
     7h19m   False
eks-exempt                     exempt            2                  <none>
     7h19m   False
probes                         exempt            2                  <none>
     7h19m   False
system-leader-election         leader-election   100                ByUser
     7h19m   False
endpoint-controller            workload-high     150                ByUser
     7h19m   False
workload-leader-election       leader-election   200                ByUser
     7h19m   False
system-node-high               node-high         400                ByUser
     7h19m   False
system-nodes                   system            500                ByUser
     7h19m   False
kube-controller-manager        workload-high     800                ByNamespace
     7h19m   False
kube-scheduler                 workload-high     800                ByNamespace
     7h19m   False
kube-system-service-accounts   workload-high     900                ByNamespace
     7h19m   False
eks-workload-high              workload-high     1000               ByUser
     7h14m   False
service-accounts               workload-low      9000               ByUser
     7h19m   False
global-default                 global-default    9900               ByUser
     7h19m   False
catch-all                      catch-all         10000              ByUser
     7h19m   False
```

PriorityLevelConfigurations에는 대기열, 거부 또는 면제 유형이 있을 수 있습니다. 대기열 및 거부 유형의 경우 해당 우선 순위 수준의 최대 진행 중 요청 수에 제한이 적용되지만, 해당 제한에 도달하면 동작이 달라집니다. 예를 들어 워크로드가 높은 PriorityLevelConfiguration은 대기열 유형을 사용하며 컨트롤러 관리자, 엔드포인트 컨트롤러, 스케줄러,eks 관련 컨트롤러 및 kube 시스템 네임스페이스에서 실행되는 포드에서 사용할 수 있는 98개의 요청이 있습니다. 대기열 유형이 사용되므로 API 서버는 요청을 메모리에 유지하려고 시도하며 이러한 요청의 제한 시간이 초과되기 전에 진행 중인 요청 수가 98개 미만으로 감소하기를 기대합니다. 지정된 요청이 대기열에서 시간 초과되거나 너무 많은 요청이 이미 대기열에 있는 경우 API 서버는 요청을 삭제하고 클라이언트에 429를 반환할 수밖에 없습니다. 대기열에 추가하면 요청이 429를 수신하지 못할 수 있지만 요청에 대한 end-to-end 지연 시간이 증가한다는 단점이 있습니다.

이제 거부 유형으로 catch-all PriorityLevelConfiguration에 매핑되는 catch-all FlowSchema를 살펴보겠습니다. 클라이언트가 13개의 진행 중인 요청 한도에 도달하면 API 서버는 대기열을 실행하지 않고 429 응답 코드를 사용하여 요청을 즉시 삭제합니다. 마지막으로 유형이 Exempt인 PriorityLevelConfiguration에 매핑된 요청은 429를 수신하지 않으며 항상 즉시 디스패치됩니다. 이는 healthz 요청 또는 system:masters 그룹에서 오는 요청과 같은 우선순위가 높은 요청에 사용됩니다.

APF 및 삭제된 요청 모니터링

APF로 인해 요청이 삭제되고 있는지 확인하기 위해에 대한 API 서버 지표를 모니터링하여 영향을 받는 FlowSchemas 및 PriorityLevelConfigurations를 확인할 `apiserver_flowcontrol_rejected_requests_total` 수 있습니다. 예를 들어이 지표는 워크로드가 적은 대기열에서 요청 제한 시간이 초과되어 서비스 계정 FlowSchema의 요청 100개가 삭제되었음을 보여줍니다.

```
% kubectl get --raw /metrics | grep apiserver_flowcontrol_rejected_requests_total
apiserver_flowcontrol_rejected_requests_total{flow_schema="service-
accounts",priority_level="workload-low",reason="time-out"} 100
```

주어진 PriorityLevelConfiguration이 429를 수신하거나 대기열에 추가되어 지연 시간이 증가하는지 확인하려면 동시성 제한과 사용 중인 동시성 간의 차이를 비교할 수 있습니다. 이 예제에는 요청 100개의 버퍼가 있습니다.

```
% kubectl get --raw /metrics | grep
 'apiserver_flowcontrol_nominal_limit_seats.*workload-low'
apiserver_flowcontrol_nominal_limit_seats{priority_level="workload-low"} 245

% kubectl get --raw /metrics | grep
 'apiserver_flowcontrol_current_executing_seats.*workload-low'
```

```
apiserver_flowcontrol_current_executing_seats{flow_schema="service-
accounts",priority_level="workload-low"} 145
```

지정된 PriorityLevelConfiguration에서 대기열에 추가되지만 반드시 삭제되지는 않는 요청이 있는지 확인하기 위해에 대한 지표를 참조할 `apiserver_flowcontrol_current_inqueue_requests` 수 있습니다.

```
% kubectl get --raw /metrics | grep
 'apiserver_flowcontrol_current_inqueue_requests.*workload-low'
apiserver_flowcontrol_current_inqueue_requests{flow_schema="service-
accounts",priority_level="workload-low"} 10
```

기타 유용한 Prometheus 지표는 다음과 같습니다.

- apiserver_flowcontrol_dispatched_requests_total
- apiserver_flowcontrol_request_execution_seconds
- apiserver_flowcontrol_request_wait_duration_seconds

APF 지표의 전체 목록은 업스트림 설명서를 참조하세요.

삭제된 요청 방지

워크로드를 변경하여 429s 방지

지정된 PriorityLevelConfiguration이 허용되는 최대 기내 요청 수를 초과하여 APF가 요청을 삭제하는 경우 영향을 받는 FlowSchemas의 클라이언트는 지정된 시간에 실행되는 요청 수를 줄일 수 있습니다. 이는 429개가 발생하는 기간 동안 이루어진 총 요청 수를 줄여 수행할 수 있습니다. 값비싼 목록 호출과 같은 장기 실행 요청은 실행 중인 전체 기간 동안 진행 중인 요청으로 계산되므로 특히 문제가 됩니다. 이러한 비용이 많이 드는 요청 수를 줄이거나 이러한 목록 호출의 지연 시간을 최적화하면(예: 요청당 가져온 객체 수를 줄이거나 감시 요청을 사용하여 로 전환) 주어진 워크로드에 필요한 총 동시성을 줄이는 데 도움이 될 수 있습니다.

APF 설정을 변경하여 429 방지

> **Warning**
>
> 수행 중인 작업을 알고 있는 경우에만 기본 APF 설정을 변경합니다. APF 설정을 잘못 구성하면 API 서버 요청이 삭제되고 워크로드가 크게 중단될 수 있습니다.

삭제된 요청을 방지하는 또 다른 방법은 EKS 클러스터에 설치된 기본 FlowSchemas 또는 PriorityLevelConfigurations를 변경하는 것입니다. EKS는 지정된 Kubernetes 마이너 버전에 대해 FlowSchemas 및 PriorityLevelConfigurations에 대한 업스트림 기본 설정을 설치합니다. 객체에 대한 다음 주석이 false로 설정되지 않는 한 API 서버는 이러한 객체를 수정하면 자동으로 기본값으로 다시 조정합니다.

```
metadata:
  annotations:
    apf.kubernetes.io/autoupdate-spec: "false"
```

상위 수준에서 APF 설정은 다음 중 하나로 수정할 수 있습니다.

- 관심 있는 요청에 더 많은 기내 용량을 할당합니다.
- 다른 요청 유형에 대한 용량을 부족하게 할 수 있는 필수적이지 않거나 비용이 많이 드는 요청을 격리합니다.

이는 기본 FlowSchemas 및 PriorityLevelConfigurations를 변경하거나 이러한 유형의 새 객체를 생성하여 수행할 수 있습니다. 연산자는 관련 PriorityLevelConfigurations 객체의 assuredConcurrencyShares 값을 늘려 할당된 진행 중인 요청의 비율을 늘릴 수 있습니다. 또한 애플리케이션이 전송되기 전에 대기 중인 요청으로 인한 추가 지연 시간을 처리할 수 있는 경우 지정된 시간에 대기할 수 있는 요청 수도 증가할 수 있습니다.

또는 고객의 워크로드에 고유한 새 FlowSchema 및 PriorityLevelConfigurations 객체를 생성할 수 있습니다. 기존 PriorityLevelConfigurations 또는 새 PriorityLevelConfigurations에 더 많은 assuredConcurrencyShares를 할당하면 전체 제한이 API 서버당 600개의 진행 중 상태로 유지되므로 다른 버킷에서 처리할 수 있는 요청 수가 감소합니다.

APF 기본값을 변경할 때 비프로덕션 클러스터에서 이러한 지표를 모니터링하여 설정을 변경해도 의도하지 않은 429가 발생하지 않도록 해야 합니다.

1. 모든 FlowSchemas에 대해에 대한 지표를 모니터링하여 요청을 삭제하기 시작하는 버킷이 없는지 확인해야 `apiserver_flowcontrol_rejected_requests_total` 합니다.
2. `apiserver_flowcontrol_nominal_limit_seats` 및의 값을 비교하여 사용 중인 동시성이 해당 우선 순위 수준에 대한 제한을 위반할 위험이 없도록 `apiserver_flowcontrol_current_executing_seats`해야 합니다.

새 FlowSchema 및 PriorityLevelConfiguration을 정의하는 일반적인 사용 사례 중 하나는 격리용입니다. 장기 실행 목록 이벤트 호출을 포드에서 자체 요청 공유로 격리하려고 한다고 가정해 보겠습니다.

이렇게 하면 기존 서비스 계정 FlowSchema를 사용하는 포드의 중요한 요청이 429를 수신하고 요청 용량이 부족해지는 것을 방지할 수 있습니다. 총 진행 중인 요청 수는 유한하지만,이 예제에서는 지정된 워크로드에 대한 요청 용량을 더 잘 나누도록 APF 설정을 수정할 수 있음을 보여줍니다.

목록 이벤트 요청을 격리하는 FlowSchema 객체의 예:

```
apiVersion: flowcontrol.apiserver.k8s.io/v1
kind: FlowSchema
metadata:
  name: list-events-default-service-accounts
spec:
  distinguisherMethod:
    type: ByUser
  matchingPrecedence: 8000
  priorityLevelConfiguration:
    name: catch-all
  rules:
  - resourceRules:
    - apiGroups:
      - '*'
      namespaces:
      - default
      resources:
      - events
      verbs:
      - list
    subjects:
    - kind: ServiceAccount
      serviceAccount:
        name: default
        namespace: default
```

- 이 FlowSchema는 기본 네임스페이스에서 서비스 계정이 수행한 모든 목록 이벤트 호출을 캡처합니다.
- 일치하는 우선 순위 8000은 기존 서비스 계정 FlowSchema에서 사용하는 값 9000보다 낮으므로 이러한 목록 이벤트 호출은 서비스 list-events-default-service-accounts 일치합니다.
- catch-all PriorityLevelConfiguration을 사용하여 이러한 요청을 격리합니다. 이 버킷은 이러한 장기 실행 목록 이벤트 호출에서 13개의 진행 중인 요청만 사용하도록 허용합니다. 포드는 13개 이상의 요청을 동시에 발행하려고 시도하는 즉시 429개를 수신하기 시작합니다.

API 서버에서 리소스 검색

API 서버에서 정보를 가져오는 것은 모든 크기의 클러스터에서 예상되는 동작입니다. 클러스터의 리소스 수를 조정하면 요청 빈도와 데이터 볼륨이 컨트롤 플레인의 병목 현상이 빠르게 발생하여 API 지연 시간과 속도가 느려질 수 있습니다. 지연 시간의 심각도에 따라 주의하지 않으면 예기치 않은 가동 중지가 발생합니다.

요청하는 내용과 빈도를 인식하면 이러한 유형의 문제를 방지하는 첫 번째 단계가 됩니다. 다음은 조정 모범 사례에 따라 쿼리 볼륨을 제한하는 지침입니다. 이 섹션의 제안은 최상의 규모 조정으로 알려진 옵션부터 순서대로 제공됩니다.

공유 인포머 사용

Kubernetes API와 통합되는 컨트롤러 및 자동화를 구축할 때 Kubernetes 리소스에서 정보를 가져와야 하는 경우가 많습니다. 이러한 리소스를 정기적으로 폴링하면 API 서버에 상당한 부하가 발생할 수 있습니다.

client-go 라이브러리의 정보 제공자를 사용하면 변경 사항을 폴링하는 대신 이벤트를 기반으로 리소스의 변경 사항을 감시할 수 있는 이점을 얻을 수 있습니다. 정보 제공자는 이벤트 및 변경 사항에 공유 캐시를 사용하여 로드를 추가로 줄이므로 동일한 리소스를 감시하는 여러 컨트롤러가 추가 로드를 추가하지 않습니다.

컨트롤러는 특히 대규모 클러스터에서 레이블 및 필드 선택기 없이 클러스터 전체 리소스를 폴링하지 않아야 합니다. 필터링되지 않은 각 폴링은 API 서버를 통해 etcd에서 많은 불필요한 데이터를 전송해야 클라이언트가 필터링할 수 있습니다. 레이블 및 네임스페이스를 기준으로 필터링하면 API 서버가 클라이언트로 전송된 요청 및 데이터를 가득 채우기 위해 수행해야 하는 작업량을 줄일 수 있습니다.

Kubernetes API 사용량 최적화

사용자 지정 컨트롤러 또는 자동화를 사용하여 Kubernetes API를 호출할 때는 호출을 필요한 리소스로만 제한하는 것이 중요합니다. 제한이 없으면 API 서버 등에 불필요한 부하가 발생할 수 있습니다.

가능하면 감시 인수를 사용하는 것이 좋습니다. 인수가 없는 경우 기본 동작은 객체를 나열하는 것입니다. 목록 대신 감시를 사용하려면 API 요청 끝에 `?watch=true`를 추가할 수 있습니다. 예를 들어, 감시를 사용하여 기본 네임스페이스의 모든 포드를 가져오려면 다음을 수행합니다.

```
/api/v1/namespaces/default/pods?watch=true
```

객체를 나열하는 경우 나열하는 항목의 범위와 반환되는 데이터의 양을 제한해야 합니다. 요청에 `limit=500` 인수를 추가하여 반환된 데이터를 제한할 수 있습니다. `fieldSelector` 인수와 `/`

namespace/ 경로는 목록의 범위를 필요에 따라 좁히는 데 유용할 수 있습니다. 예를 들어 기본 네임스페이스에서 실행 중인 포드만 나열하려면 다음 API 경로와 인수를 사용합니다.

```
/api/v1/namespaces/default/pods?fieldSelector=status.phase=Running&limit=500
```

또는 다음을 사용하여 실행 중인 모든 포드를 나열합니다.

```
/api/v1/pods?fieldSelector=status.phase=Running&limit=500
```

감시 호출 또는 나열된 객체를 제한하는 또 다른 옵션은 resourceVersions Kubernetes 설명서에서 읽을 수 있는를 사용하는 것입니다. resourceVersion 인수가 없으면 데이터베이스에서 가장 비싸고 가장 느린 읽기인 etcd 쿼럼 읽기가 필요한 사용 가능한 최신 버전을 받게 됩니다. resourceVersion은 쿼리하려는 리소스에 따라 다르며 metadata.resourseVersion 필드에서 찾을 수 있습니다. 이는 통화 나열뿐만 아니라 감시 통화를 사용하는 경우에도 권장됩니다.

API 서버 캐시의 결과를 반환하는 특별한 기능이 resourceVersion=0 있습니다. 이렇게 하면 etcd 부하를 줄일 수 있지만 페이지 매김을 지원하지 않습니다.

```
/api/v1/namespaces/default/pods?resourceVersion=0
```

resourceVersion이 이전 목록 또는 감시에서 수신한 가장 최근 알려진 값으로 설정된 감시를 사용하는 것이 좋습니다. 이는 client-go에서 자동으로 처리됩니다. 그러나 다른 언어로 k8s 클라이언트를 사용하는 경우 다시 확인하는 것이 좋습니다.

```
/api/v1/namespaces/default/pods?watch=true&resourceVersion=362812295
```

인수 없이 API를 호출하면 API 서버 및 등에 가장 많은 리소스가 사용됩니다. 이 호출은 페이지 매김이나 범위 제한 없이 모든 네임스페이스의 모든 포드를 가져오고 etcd에서 쿼럼 읽기가 필요합니다.

```
/api/v1/pods
```

DaemonSet 천둥 군집 방지

DaemonSet는 모든(또는 일부) 노드가 포드의 복사본을 실행하도록 합니다. 노드가 클러스터에 조인하면 데몬 세트 컨트롤러가 해당 노드에 대한 포드를 생성합니다. 노드가 클러스터에서 나가면 해당 포드가 가비지 수집됩니다. DaemonSet를 삭제하면 생성된 포드가 정리됩니다.

DaemonSet의 일반적인 용도는 다음과 같습니다.

- 모든 노드에서 클러스터 스토리지 데몬 실행
- 모든 노드에서 로그 수집 데몬 실행
- 모든 노드에서 노드 모니터링 데몬 실행

수천 개의 노드가 있는 클러스터에서 새 DaemonSet를 생성하거나, DaemonSet를 업데이트하거나, 노드 수를 늘리면 컨트롤 플레인에 부하가 높아질 수 있습니다. DaemonSet 포드가 포드 시작 시 비용이 많이 드는 API 서버 요청을 발행하면 많은 수의 동시 요청으로 인해 컨트롤 플레인에서 리소스 사용량이 높아질 수 있습니다.

정상 작동에서는를 사용하여 새 DaemonSet 포드가 점진적으로 롤아웃되도록 `RollingUpdate`할 수 있습니다. `RollingUpdate` 업데이트 전략을 사용하면 DaemonSet 템플릿을 업데이트한 후 컨트롤러는 오래된 DaemonSet 포드를 종료하고 제어된 방식으로 새 DaemonSet 포드를 자동으로 생성합니다. DaemonSet의 포드는 전체 업데이트 프로세스 중에 각 노드에서 최대 1개까지 실행됩니다. 를 1, `maxUnavailable`를 0, `maxSurge`를 60으로 설정하여 점진적 롤아웃`minReadySeconds`을 수행할 수 있습니다. 업데이트 전략을 지정하지 않으면 Kubernetes는 기본적으로가 1, `maxUnavailable`가 0, `RollingUpdate`가 0`maxSurge`인 `minReadySeconds`를 생성합니다.

```
minReadySeconds: 60
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 0
    maxUnavailable: 1
```

DaemonSet가 이미 생성되어 있고 모든 노드`RollingUpdate`에서 예상 포드 수가 있는 경우는 새 DaemonSet `Ready` 포드를 점진적으로 롤아웃합니다. `RollingUpdate` 전략에서 다루지 않는 특정 조건에서는 군집 문제를 우회할 수 있습니다.

DaemonSet 생성 시 천둥 군집 방지

기본적으로 `RollingUpdate` 구성에 관계없이 kube-controller-manager는 새 DaemonSet를 생성할 때 일치하는 모든 노드에 대한 포드를 동시에 생성합니다. DaemonSet를 생성한 후 포드를 점진적으로 롤아웃하려면 `NodeSelector` 또는를 사용할 수 있습니다`NodeAffinity`. 그러면 0개의 노드와 일치하는 DaemonSet가 생성되고 제어된 속도로 DaemonSet에서 포드를 실행할 수 있도록 노드를 점진적으로 업데이트할 수 있습니다. 다음 접근 방식을 따를 수 있습니다.

- 의 모든 노드에 레이블을 추가합니다`run-daemonset=false`.

```
kubectl label nodes --all run-daemonset=false
```

- `run-daemonset=false` 레이블이 없는 노드와 일치하도록 `NodeAffinity` 설정으로 DaemonSet를 생성합니다. 처음에는 DaemonSet에 해당 포드가 없습니다.

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: run-daemonset
          operator: NotIn
          values:
          - "false"
```

- 노드에서 제어된 속도로 `run-daemonset=false` 레이블을 제거합니다. 이 bash 스크립트를 예로 사용할 수 있습니다.

```
#!/bin/bash

nodes=$(kubectl get --raw "/api/v1/nodes" | jq -r '.items | .[].metadata.name')

for node in ${nodes[@]}; do
   echo "Removing run-daemonset label from node $node"
   kubectl label nodes $node run-daemonset-
   sleep 5
done
```

- 필요에 따라 DaemonSet 객체에서 `NodeAffinity` 설정을 제거합니다. DaemonSet 템플릿이 변경되었으므로 이렇게 하면 도 트리거`RollingUpdate`되고 기존의 모든 DaemonSet 포드가 점진적으로 교체됩니다.

노드 스케일 아웃 시 천둥 군집 방지

DaemonSet 생성과 마찬가지로 빠른 속도로 새 노드를 생성하면 많은 수의 DaemonSet 포드가 동시에 시작될 수 있습니다. 컨트롤러가 동일한 속도로 DaemonSet 포드를 생성하도록 제어된 속도로 새 노드를 생성해야 합니다. 이렇게 할 수 없는 경우를 사용하여 새 노드를 처음에 기존 DaemonSet에 적합하

지 않게 만들 수 있습니다NodeAffinity. 그런 다음 데몬 세트 컨트롤러가 제어된 속도로 포드를 생성하도록 새 노드에 레이블을 점진적으로 추가할 수 있습니다. 다음 접근 방식을 따를 수 있습니다.

- 의 모든 기존 노드에 레이블 추가 `run-daemonset=true`

```
kubectl label nodes --all run-daemonset=true
```

- DaemonSet를 `run-daemonset=true` 레이블이 있는 노드와 일치하도록 `NodeAffinity` 설정으로 업데이트합니다. DaemonSet 템플릿이 변경되었기 때문에 도 트리거RollingUpdate되고 기존의 모든 DaemonSet 포드가 점진적으로 교체됩니다. 다음 단계로 진행하기 전에이 `RollingUpdate` 완료될 때까지 기다려야 합니다.

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: run-daemonset
          operator: In
          values:
          - "true"
```

- 클러스터에 새 노드를 생성합니다. 이러한 노드에는 `run-daemonset=true` 레이블이 없으므로 DaemonSet가 해당 노드와 일치하지 않습니다.
- 제어된 속도로 새 노드(현재 `run-daemonset=true` 레이블이 없음)에 `run-daemonset` 레이블을 추가합니다. 이 bash 스크립트를 예로 사용할 수 있습니다.

```
#!/bin/bash

nodes=$(kubectl get --raw "/api/v1/nodes?labelSelector=%21run-daemonset" | jq -r
 '.items | .[].metadata.name')

for node in ${nodes[@]}; do
   echo "Adding run-daemonset=true label to node $node"
   kubectl label nodes $node run-daemonset=true
   sleep 5
done
```

- 필요에 따라 DaemonSet 객체에서 NodeAffinity 설정을 제거하고 모든 노드에서 run-daemonset 레이블을 제거합니다.

DaemonSet 업데이트에서 폭동 군집 방지

RollingUpdate 정책은 인 DaemonSet 포드에 대한 maxUnavailable 설정만 준수합니다Ready. DaemonSet에 NotReady 포드만 있거나 NotReady 포드의 비율이 높고 템플릿을 업데이트하는 경우 데몬 세트 컨트롤러는 모든 포드에 대해 새 NotReady 포드를 동시에 생성합니다. 이로 인해 NotReady 포드가 지속적으로 충돌 루프 중이거나 이미지를 가져오지 못하는 경우와 같이 포드 수가 많은 경우 군집 문제가 폭증할 수 있습니다.

DaemonSet를 업데이트할 때 포드를 점진적으로 롤아웃하려면 DaemonSet의 업데이트 전략을에서 RollingUpdate로 일시적으로 변경할 NotReady 수 있습니다OnDelete. 를 사용하면 DaemonSet 템플릿을 업데이트OnDelete한 후 컨트롤러는 이전 포드를 수동으로 삭제한 후 새 포드를 생성하므로 새 포드의 롤아웃을 제어할 수 있습니다. 다음 접근 방식을 따를 수 있습니다.

- DaemonSet에 NotReady 포드가 있는지 확인합니다.
- 그렇지 않은 경우 DaemonSet 템플릿을 안전하게 업데이트할 수 있으며 RollingUpdate 전략은 점진적인 롤아웃을 보장합니다.
- 그렇다면 먼저 OnDelete 전략을 사용하도록 DaemonSet를 업데이트해야 합니다.

```
updateStrategy:
  type: OnDelete
```

- 그런 다음 DaemonSet 템플릿을 필요한 변경 사항으로 업데이트합니다.
- 이 업데이트 후 제어된 속도로 포드 삭제 요청을 실행하여 이전 DaemonSet 포드를 삭제할 수 있습니다. 이 bash 스크립트를 kube-system 네임스페이스에서 DaemonSet 이름이 fluentd-elasticsearch인 예제로 사용할 수 있습니다.

```
#!/bin/bash

daemonset_pods=$(kubectl get --raw "/api/v1/namespaces/kube-system/pods?
labelSelector=name%3Dfluentd-elasticsearch" | jq -r '.items | .[].metadata.name')

for pod in ${daemonset_pods[@]}; do
   echo "Deleting pod $pod"
   kubectl delete pod $pod -n kube-system
```

```
    sleep 5
done
```

- 마지막으로 DaemonSet를 이전 `RollingUpdate` 전략으로 다시 업데이트할 수 있습니다.

Kubernetes 데이터 영역

EC2 인스턴스 유형을 선택하는 것은 워크로드가 여러 개인 클러스터에서 발생하기 때문에 고객이 직면하는 가장 어려운 결정 중 하나일 수 있습니다. 모든 솔루션에 one-size-fits 솔루션은 없습니다. 다음은 컴퓨팅 크기 조정과 관련된 일반적인 위험을 방지하는 데 도움이 되는 몇 가지 팁입니다.

자동 노드 Autoscaling

토일을 줄이고 Kubernetes와 깊이 통합되는 노드 Auto Scaling을 사용하는 것이 좋습니다. 대규모 클러스터에는 관리형 노드 그룹과 Karpenter를 사용하는 것이 좋습니다.

관리형 노드 그룹은 관리형 업그레이드 및 구성에 대한 추가 이점과 함께 Amazon EC2 Auto Scaling 그룹의 유연성을 제공합니다. Kubernetes Cluster Autoscaler로 확장할 수 있으며 다양한 컴퓨팅 요구 사항이 있는 클러스터의 일반적인 옵션입니다.

Karpenter는 AWS에서 생성한 오픈 소스 워크로드 네이티브 노드 오토스케일러입니다. 노드 그룹을 관리하지 않고 리소스(예: GPU) 및 테인트 및 허용 오차(예: 영역 분산)에 대한 워크로드 요구 사항에 따라 클러스터의 노드를 확장합니다. 노드는 그룹당 450개의 노드라는 기본 노드 그룹 할당량을 피하고 운영 오버헤드를 줄이면서 인스턴스 선택 유연성을 제공하는 EC2에서 직접 생성됩니다. 가능하면 Karpenter를 사용하는 것이 좋습니다.

다양한 EC2 인스턴스 유형 사용

각 AWS 리전에는 인스턴스 유형당 사용 가능한 인스턴스 수가 제한됩니다. 하나의 인스턴스 유형만 사용하는 클러스터를 생성하고 리전의 용량을 초과하여 노드 수를 조정하면 사용 가능한 인스턴스가 없다는 오류가 발생합니다. 이 문제를 방지하려면 클러스터에서 사용할 수 있는 인스턴스 유형을 임의로 제한해서는 안 됩니다.

Karpenter는 기본적으로 광범위한 호환 인스턴스 유형을 사용하며 보류 중인 워크로드 요구 사항, 가용성 및 비용에 따라 프로비저닝 시간에 인스턴스를 선택합니다. NodePools의 `karpenter.k8s.aws/instance-category` 키에 사용되는 인스턴스 유형 목록을 확장할 수 있습니다. NodePools

Kubernetes Cluster Autoscaler를 사용하려면 노드 그룹을 비슷한 크기로 조정하여 일관되게 확장할 수 있어야 합니다. CPU 및 메모리 크기에 따라 여러 그룹을 생성하고 독립적으로 확장해야 합니다. ec2-instance-selector를 사용하여 노드 그룹에 맞게 비슷한 크기의 인스턴스를 식별합니다.

```
ec2-instance-selector --service eks --vcpus-min 8 --memory-min 16
a1.2xlarge
a1.4xlarge
a1.metal
c4.4xlarge
c4.8xlarge
c5.12xlarge
c5.18xlarge
c5.24xlarge
c5.2xlarge
c5.4xlarge
c5.9xlarge
c5.metal
```

API 서버 로드를 줄이기 위해 더 큰 노드 선호

사용할 인스턴스 유형을 결정할 때 실행 중인 kubelet 및 DaemonSets가 적기 때문에 더 적은 수의 대형 노드가 Kubernetes 컨트롤 플레인에 대한 부하를 줄입니다. 그러나 대형 노드는 소형 노드처럼 완전히 활용되지 않을 수 있습니다. 노드 크기는 워크로드 가용성 및 규모 요구 사항에 따라 평가해야 합니다.

3개의 u-24tb1.metal 인스턴스(24TB 메모리 및 448 코어)가 있는 클러스터에는 3개의 kubelet이 있으며 기본적으로 노드당 110개의 포드로 제한됩니다. 포드가 각각 4개의 코어를 사용하는 경우 이는 예상될 수 있습니다(4개의 코어 x 110 = 440개의 코어/노드). 3노드 클러스터의 경우 인스턴스 중단 1개가 클러스터의 1/3에 영향을 미칠 수 있으므로 인스턴스 인시던트를 처리하는 기능이 낮아집니다. Kubernetes 스케줄러가 워크로드를 올바르게 배치할 수 있도록 워크로드에 노드 요구 사항과 포드 분산을 지정해야 합니다.

워크로드는 테인트, 허용 오차 및 PodTopologySpread를 통해 필요한 리소스와 필요한 가용성을 정의해야 합니다. 컨트롤 플레인 로드를 줄이고, 운영을 줄이고, 비용을 절감하려면 최대한 활용할 수 있는 가장 큰 노드를 선호하고 가용성 목표를 충족해야 합니다.

Kubernetes 스케줄러는 리소스를 사용할 수 있는 경우 가용 영역 및 호스트에 워크로드를 자동으로 분산하려고 시도합니다. 사용 가능한 용량이 없는 경우 Kubernetes Cluster Autoscaler는 각 가용 영역에 노드를 균등하게 추가하려고 시도합니다. Karpenter는 워크로드가 다른 요구 사항을 지정하지 않는 한 노드를 최대한 빠르고 저렴하게 추가하려고 시도합니다.

워크로드가 스케줄러와 새 노드로 분산되도록 하려면 topologySpreadConstraints를 사용해야 합니다.

```
spec:
  topologySpreadConstraints:
    - maxSkew: 3
      topologyKey: "topology.kubernetes.io/zone"
      whenUnsatisfiable: ScheduleAnyway
      labelSelector:
        matchLabels:
          dev: my-deployment
    - maxSkew: 2
      topologyKey: "kubernetes.io/hostname"
      whenUnsatisfiable: ScheduleAnyway
      labelSelector:
        matchLabels:
          dev: my-deployment
```

일관된 워크로드 성능을 위해 유사한 노드 크기 사용

워크로드는 일관된 성능과 예측 가능한 규모 조정을 위해 실행해야 하는 노드 크기를 정의해야 합니다. 500m CPU를 요청하는 워크로드는 코어가 4개인 인스턴스와 코어가 16개인 인스턴스에서 다르게 작동합니다. T 시리즈 인스턴스와 같이 버스트 가능한 CPUs.

워크로드의 일관된 성능을 보장하기 위해 워크로드는 지원되는 Karpenter 레이블을 사용하여 특정 인스턴스 크기를 대상으로 지정할 수 있습니다.

```
kind: deployment
...
spec:
  template:
    spec:
    containers:
    nodeSelector:
      karpenter.k8s.aws/instance-size: 8xlarge
```

Kubernetes Cluster Autoscaler를 사용하여 클러스터에서 예약되는 워크로드는 레이블 일치를 기반으로 노드 선택기와 노드 그룹과 일치해야 합니다.

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
```

```
        nodeSelectorTerms:
        - matchExpressions:
          - key: eks.amazonaws.com/nodegroup
            operator: In
            values:
            - 8-core-node-group    # match your node group name
```

컴퓨팅 리소스를 효율적으로 사용

컴퓨팅 리소스에는 EC2 인스턴스 및 가용 영역이 포함됩니다. 컴퓨팅 리소스를 효과적으로 사용하면 확장성, 가용성, 성능이 향상되고 총 비용이 절감됩니다. 여러 애플리케이션이 있는 Auto Scaling 환경에서는 효율적인 리소스 사용량을 예측하기가 매우 어렵습니다. Karpenter는 사용률과 유연성을 극대화하기 위해 워크로드 요구 사항에 따라 온디맨드로 인스턴스를 프로비저닝하도록 생성되었습니다.

Karpenter를 사용하면 워크로드가 먼저 노드 그룹을 생성하거나 특정 노드에 대한 레이블 테인트를 구성하지 않고도 필요한 컴퓨팅 리소스 유형을 선언할 수 있습니다. 자세한 내용은 Karpenter 모범 사례를 참조하세요. Karpenter 프로비저너에서 통합을 활성화하여 사용 중인 노드를 대체하는 것이 좋습니다.

Amazon Machine Image(AMI) 업데이트 자동화

작업자 노드 구성 요소를 최신 상태로 유지하면 최신 보안 패치와 Kubernetes API와 호환되는 기능이 있는지 확인할 수 있습니다. kubelet을 업데이트하는 것은 Kubernetes 기능에 가장 중요한 구성 요소이지만 OS, 커널 및 로컬로 설치된 애플리케이션 패치를 자동화하면 확장 시 유지 관리를 줄일 수 있습니다.

노드 이미지에 최신 Amazon EKS 최적화 Amazon Linux 2 또는 Amazon EKS 최적화 Bottlerocket AMI를 사용하는 것이 좋습니다. Karpenter는 사용 가능한 최신 AMI를 자동으로 사용하여 클러스터에 새 노드를 프로비저닝합니다. 관리형 노드 그룹은 노드 그룹 업데이트 중에 AMI를 업데이트하지만 노드 프로비저닝 시간에는 AMI ID를 업데이트하지 않습니다.

관리형 노드 그룹의 경우 패치 릴리스에 사용할 수 있는 경우 새 AMI IDs로 Auto Scaling 그룹(ASG) 시작 템플릿을 업데이트해야 합니다. AMI 마이너 버전(예: 1.23.5~1.24.3)은 EKS 콘솔 및 API에서 노드 그룹의 업그레이드로 사용할 수 있습니다. 패치 릴리스 버전(예: 1.23.5~1.23.6)은 노드 그룹에 대한 업그레이드로 제공되지 않습니다. AMI 패치 릴리스로 노드 그룹을 최신 상태로 유지하려면 새 시작 템플릿 버전을 생성하고 노드 그룹이 인스턴스를 새 AMI 릴리스로 대체하도록 해야 합니다.

이 페이지에서 사용 가능한 최신 AMI를 찾거나 AWS CLI를 사용할 수 있습니다.

```
aws ssm get-parameter \
  --name /aws/service/eks/optimized-ami/1.24/amazon-linux-2/recommended/image_id \
```

```
  --query "Parameter.Value" \
  --output text
```

컨테이너에 여러 EBS 볼륨 사용

EBS 볼륨에는 볼륨 유형(예: gp3) 및 디스크 크기에 따라 입출력(I/O) 할당량이 있습니다. 애플리케이션이 호스트와 단일 EBS 루트 볼륨을 공유하는 경우 전체 호스트의 디스크 할당량이 소진되어 다른 애플리케이션이 사용 가능한 용량을 기다릴 수 있습니다. 애플리케이션은 오버레이 파티션에 파일을 쓰고, 호스트에서 로컬 볼륨을 탑재하고, 사용된 로깅 에이전트에 따라 표준 출력(STDOUT)에 로그인할 때 디스크에 씁니다.

디스크 I/O 소진을 방지하려면 컨테이너 상태 폴더(예: /run/containerd)에 두 번째 볼륨을 탑재하고 워크로드 스토리지에 별도의 EBS 볼륨을 사용하고 불필요한 로컬 로깅을 비활성화해야 합니다.

eksctl을 사용하여 EC2 인스턴스에 두 번째 볼륨을 탑재하려면 다음 구성과 함께 노드 그룹을 사용할 수 있습니다.

```
managedNodeGroups:
  - name: al2-workers
    amiFamily: AmazonLinux2
    desiredCapacity: 2
    volumeSize: 80
    additionalVolumes:
      - volumeName: '/dev/sdz'
        volumeSize: 100
    preBootstrapCommands:
    - |
      "systemctl stop containerd"
      "mkfs -t ext4 /dev/nvme1n1"
      "rm -rf /var/lib/containerd/*"
      "mount /dev/nvme1n1 /var/lib/containerd/"
      "systemctl start containerd"
```

terraform을 사용하여 노드 그룹을 프로비저닝하는 경우 EKS Blueprints for terraform의 예제를 참조하세요. Karpenter를 사용하여 노드를 프로비저닝하는 경우 노드 사용자 데이터와 `blockDeviceMappings` 함께를 사용하여 볼륨을 추가할 수 있습니다.

EBS 볼륨을 포드에 직접 탑재하려면 AWS EBS CSI 드라이버를 사용하고 스토리지 클래스와 함께 볼륨을 사용해야 합니다.

```
---
```

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ebs-sc
provisioner: ebs.csi.aws.com
volumeBindingMode: WaitForFirstConsumer
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ebs-claim
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: ebs-sc
  resources:
    requests:
      storage: 4Gi
---
apiVersion: v1
kind: Pod
metadata:
  name: app
spec:
  containers:
  - name: app
    image: public.ecr.aws/docker/library/nginx
    volumeMounts:
    - name: persistent-storage
      mountPath: /data
  volumes:
  - name: persistent-storage
    persistentVolumeClaim:
      claimName: ebs-claim
```

워크로드가 EBS 볼륨을 사용하는 경우 EBS 연결 제한이 낮은 인스턴스 방지

EBS는 워크로드가 영구 스토리지를 보유하는 가장 쉬운 방법 중 하나이지만 확장성 제한도 함께 제공합니다. 각 인스턴스 유형에는 연결할 수 있는 최대 EBS 볼륨 수가 있습니다. 워크로드는 실행해야 하는 인스턴스 유형을 선언하고 Kubernetes 테인트가 있는 단일 인스턴스의 복제본 수를 제한해야 합니다.

디스크에 대한 불필요한 로깅 비활성화

프로덕션 환경에서 디버그 로깅으로 애플리케이션을 실행하지 않고 디스크에 자주 읽고 쓰는 로깅을 비활성화하여 불필요한 로컬 로깅을 방지합니다. 저널링은 로그 버퍼를 메모리에 유지하고 주기적으로 디스크로 플러시하는 로컬 로깅 서비스입니다. 저널링은 모든 줄을 디스크에 즉시 로깅하는 syslog보다 선호됩니다. 또한 syslog를 비활성화하면 필요한 총 스토리지 양이 줄어들고 복잡한 로그 교체 규칙이 필요하지 않습니다. syslog를 비활성화하려면 cloud-init 구성에 다음 코드 조각을 추가할 수 있습니다.

```
runcmd:
  - [ systemctl, disable, --now, syslog.service ]
```

OS 업데이트 속도가 필요한 경우 패치 인스턴스 적용

> **Important**
>
> 패치 적용 인스턴스는 필요한 경우에만 수행해야 합니다. Amazon은 애플리케이션을 사용하는 것과 동일한 방식으로 하위 환경을 통해 승격되는 업데이트를 변경 불가능하고 철저하게 테스트하는 것으로 인프라를 취급할 것을 권장합니다. 이 섹션은 불가능할 때 적용됩니다.

컨테이너화된 워크로드를 중단하지 않고 기존 Linux 호스트에 패키지를 설치하는 데 몇 초가 걸립니다. 인스턴스를 연결, 드레이닝 또는 교체하지 않고도 패키지를 설치하고 검증할 수 있습니다.

인스턴스를 교체하려면 먼저 새 AMIs. 인스턴스는 교체를 생성해야 하며 이전 인스턴스는 연결 및 드레이닝해야 합니다. 그런 다음 새 인스턴스에서 워크로드를 생성하고 패치를 적용해야 하는 모든 인스턴스에 대해 확인 및 반복해야 합니다. 워크로드를 중단하지 않고 인스턴스를 안전하게 교체하는 데 몇 시간, 며칠 또는 몇 주가 걸립니다.

Amazon은 자동화된 선언적 시스템에서 빌드, 테스트 및 승격된 변경 불가능한 인프라를 사용할 것을 권장하지만, 시스템을 신속하게 패치해야 하는 경우 새 AMIs 제공되면 시스템을 패치하고 교체해야 합니다. 패치 적용과 시스템 교체 간의 시간 차이가 크기 때문에 필요한 경우 AWS Systems Manager 패치 관리자를 사용하여 패치 적용 노드를 자동화하는 것이 좋습니다.

패치 노드를 사용하면 AMI가 업데이트된 후 정기적으로 보안 업데이트를 신속하게 롤아웃하고 인스턴스를 교체할 수 있습니다. Flatcar Container Linux 또는 Bottlerocket OS와 같은 읽기 전용 루트 파일 시스템이 있는 운영 체제를 사용하는 경우 해당 운영 체제에서 작동하는 업데이트 연산자를 사용하는 것이 좋습니다. Flatcar Linux 업데이트 운영자와 Bottlerocket 업데이트 운영자는 노드를 자동으로 최신 상태로 유지하기 위해 인스턴스를 재부팅합니다.

클러스터 서비스

클러스터 서비스는 EKS 클러스터 내에서 실행되지만 사용자 워크로드는 아닙니다. Linux 서버가 있는 경우 워크로드를 지원하기 위해 NTP, syslog 및 컨테이너 런타임과 같은 서비스를 실행해야 하는 경우가 많습니다. 클러스터 서비스는 비슷하며 클러스터를 자동화하고 운영하는 데 도움이 되는 지원 서비스입니다. Kubernetes에서는 일반적으로 kube 시스템 네임스페이스에서 실행되며 일부는 DaemonSets로 실행됩니다.

클러스터 서비스는 가동 시간이 길 것으로 예상되며 중단 및 문제 해결을 위해 중요한 경우가 많습니다. 코어 클러스터 서비스를 사용할 수 없는 경우 데이터 액세스 권한을 잃어 중단을 복구하거나 방지하는 데 도움이 될 수 있습니다(예: 높은 디스크 사용률). 별도의 노드 그룹 또는 AWS Fargate와 같은 전용 컴퓨팅 인스턴스에서 실행해야 합니다. 이렇게 하면 클러스터 서비스가 확장되거나 더 많은 리소스를 사용할 수 있는 워크로드의 공유 인스턴스에 영향을 받지 않습니다.

CoreDNS 규모 조정

CoreDNS 조정에는 두 가지 기본 메커니즘이 있습니다. CoreDNS 서비스에 대한 호출 수를 줄이고 복제본 수를 늘립니다.

ndots를 줄여 외부 쿼리 감소

ndots 설정은 DNS 쿼리를 방지하기에 충분한 것으로 간주되는 도메인 이름의 마침표(일명 "점") 수를 지정합니다. 애플리케이션의 ndots 설정이 5(기본값)이고 api.example.com(점 2개)과 같은 외부 도메인에서 리소스를 요청하는 경우 더 구체적인 도메인에 대해 /etc/resolv.conf에 정의된 각 검색 도메인에 대해 CoreDNS가 쿼리됩니다. 기본적으로 외부 요청을 하기 전에 다음 도메인이 검색됩니다.

```
api.example.<namespace>.svc.cluster.local
api.example.svc.cluster.local
api.example.cluster.local
api.example.<region>.compute.internal
```

`namespace` 및 `region` 값은 워크로드 네임스페이스 및 컴퓨팅 리전으로 대체됩니다. 클러스터 설정에 따라 추가 검색 도메인이 있을 수 있습니다.

워크로드의 ndots 옵션을 낮추거나 후행를 포함하여 도메인 요청을 완전히 검증하여 CoreDNS에 대한 요청 수를 줄일 수 있습니다(예: `api.example.com.`). 워크로드가 DNS를 통해 외부 서비스에 연결하는 경우 워크로드가 불필요한 클러스터 내 DNS 쿼리를 생성하지 않도록 ndot를 2로 설정하는 것이 좋습니다. 워크로드에 클러스터 내의 서비스에 대한 액세스가 필요하지 않은 경우 다른 DNS 서버 및 검색 도메인을 설정할 수 있습니다.

```
spec:
  dnsPolicy: "None"
  dnsConfig:
    options:
      - name: ndots
        value: "2"
      - name: edns0
```

ndots를 너무 낮은 값으로 낮추거나 연결하려는 도메인에 충분한 특이도(후행 포함)가 포함되어 있지 않으면 DNS 조회가 실패할 수 있습니다. 이 설정이 워크로드에 미치는 영향을 테스트해야 합니다.

CoreDNS 수평 확장

CoreDNS 인스턴스는 배포에 복제본을 추가하여 확장할 수 있습니다. NodeLocal DNS 또는 클러스터 비례 오토스케일러를 사용하여 CoreDNS를 조정하는 것이 좋습니다.

NodeLocal DNS는 클러스터에 더 많은 컴퓨팅 리소스가 필요한 DaemonSet로 노드당 하나의 인스턴스를 실행해야 하지만 실패한 DNS 요청을 방지하고 클러스터의 DNS 쿼리에 대한 응답 시간을 줄입니다. 클러스터 비례 오토스케일러는 클러스터의 노드 또는 코어 수에 따라 CoreDNS를 조정합니다. 이는 요청 쿼리와 직접적인 상관 관계가 아니지만 워크로드 및 클러스터 크기에 따라 유용할 수 있습니다. 기본 비례 규모는 클러스터의 코어 256개 또는 노드 16개 중 먼저 발생하는 것에 대해 추가 복제본을 추가하는 것입니다.

CoreDNS EKS 추가 기능을 사용하는 경우 Auto Scaling 옵션을 활성화하는 것이 좋습니다. CoreDNS 오토스케일러는 최대 (노드÷16) 또는 (CPU 코어÷256)를 차지하는 공식을 사용하여 노드 수와 CPU 코어를 모니터링하여 CoreDNS 복제본 수를 동적으로 조정하고 필요한 경우 즉시 확장하고 점진적으로 축소하여 안정성을 유지합니다.

Kubernetes 지표 서버를 수직으로 확장

Kubernetes 지표 서버는 수평 및 수직 조정을 지원합니다. 지표 서버를 수평적으로 조정하면 가용성이 높아지지만 더 많은 클러스터 지표를 처리하기 위해 수평적으로 조정되지는 않습니다. 노드와 수집된 지표가 클러스터에 추가되면 권장 사항에 따라 지표 서버를 수직으로 조정해야 합니다.

지표 서버는 수집, 집계 및 서비스하는 데이터를 메모리에 보관합니다. 클러스터가 증가하면 Metrics Server가 저장하는 데이터의 양이 증가합니다. 대규모 클러스터에서 지표 서버는 기본 설치에 지정된 메모리 및 CPU 예약보다 더 많은 컴퓨팅 리소스를 필요로 합니다. Vertical Pod Autoscaler(VPA) 또는 Addon Resizer를 사용하여 지표 서버를 조정할 수 있습니다. Addon Resizer는 작업자 노드에 비례하여 수직으로 확장되며, VPA는 CPU 및 메모리 사용량에 따라 확장됩니다.

CoreDNS lameduck 기간

포드는 이름 확인에 kube-dns 서비스를 사용합니다. Kubernetes는 대상 NAT(DNAT)를 사용하여 노드에서 CoreDNS 백엔드 포드로 kube-dns 트래픽을 리디렉션합니다. CoreDNS 배포를 확장할 때는 노드의 iptables 규칙 및 체인을 kube-proxy 업데이트하여 DNS 트래픽을 CoreDNS 포드로 리디렉션합니다. 스케일 업 시 새 엔드포인트를 전파하고 CoreDNS 다운 시 규칙을 삭제하는 데 클러스터 크기에 따라 1~10초가 걸릴 수 있습니다.

이 전파 지연으로 인해 CoreDNS 포드가 종료되었지만 노드의 iptables 규칙이 업데이트되지 않은 경우 DNS 조회에 실패할 수 있습니다. 이 시나리오에서 노드는 종료된 CoreDNS 포드에 DNS 쿼리를 계속 전송할 수 있습니다.

CoreDNS 포드에서 lameduck 기간을 설정하여 DNS 조회 실패를 줄일 수 있습니다. lameduck 모드에서 CoreDNS는 진행 중인 요청에 계속 응답합니다. lameduck 기간을 설정하면 CoreDNS 종료 프로세스가 지연되어 노드가 iptables 규칙 및 체인을 업데이트하는 데 필요한 시간을 확보할 수 있습니다.

CoreDNS lameduck 기간을 30초로 설정하는 것이 좋습니다.

CoreDNS 준비 프로브

CoreDNS의 준비 프로브/health에는 /ready 대신를 사용하는 것이 좋습니다.

lameduck 기간을 30초로 설정하는 이전 권장 사항에 따라 포드 종료 전에 노드의 iptables 규칙을 업데이트할 수 있는 충분한 시간을 제공하고 CoreDNS 준비 프로브 /ready 대신 /health를 사용하면 시작 시 CoreDNS 포드가 DNS 요청에 즉시 응답할 수 있도록 완전히 준비될 수 있습니다.

```
readinessProbe:
  httpGet:
    path: /ready
    port: 8181
    scheme: HTTP
```

CoreDNS Ready 플러그인에 대한 자세한 내용은 https://coredns.io/plugins/ready/ 참조하십시오.

에이전트 로깅 및 모니터링

로깅 및 모니터링 에이전트는 API 서버를 쿼리하여 워크로드 메타데이터로 로그 및 지표를 보강하기 때문에 클러스터 컨트롤 플레인에 상당한 부하를 추가할 수 있습니다. 노드의 에이전트는 로컬 노드 리소스에만 액세스하여 컨테이너 및 프로세스 이름과 같은 것을 볼 수 있습니다. API 서버를 쿼리하면 Kubernetes 배포 이름 및 레이블과 같은 세부 정보를 추가할 수 있습니다. 이는 문제 해결에 매우 유용할 수 있지만 규모 조정에는 해로울 수 있습니다.

로깅 및 모니터링에는 매우 다양한 옵션이 있기 때문에 모든 공급자에 대한 예를 표시할 수는 없습니다. fluentbit에서는 Use_Kubelet을 활성화하여 Kubernetes API 서버 대신 로컬 kubelet에서 메타데이터를 가져오고 데이터를 캐시할 수 있을 때 반복 호출을 줄이는 숫자Kube_Meta_Cache_TTL로 설정하는 것이 좋습니다(예: 60).

조정 모니터링 및 로깅에는 두 가지 일반 옵션이 있습니다.

- 통합 비활성화
- 샘플링 및 필터링

로그 메타데이터가 손실되기 때문에 통합을 비활성화하는 것은 종종 옵션이 아닙니다. 이렇게 하면 API 조정 문제가 제거되지만 필요할 때 필요한 메타데이터가 없어 다른 문제가 발생합니다.

샘플링 및 필터링은 수집되는 지표 및 로그 수를 줄입니다. 이렇게 하면 Kubernetes API에 대한 요청 양이 줄어들고 수집된 지표 및 로그에 필요한 스토리지 양이 줄어듭니다. 스토리지 비용을 줄이면 전체 시스템의 비용이 절감됩니다.

샘플링을 구성하는 기능은 에이전트 소프트웨어에 따라 달라지며 다양한 수집 지점에서 구현할 수 있습니다. API 서버 호출이 발생할 가능성이 있으므로 에이전트에 최대한 가깝게 샘플링을 추가하는 것이 중요합니다. 샘플링 지원에 대해 자세히 알아보려면 공급자에게 문의하세요.

CloudWatch 및 CloudWatch Logs를 사용하는 경우 설명서에 설명된 패턴을 사용하여 에이전트 필터링을 추가할 수 있습니다.

로그 및 지표 손실을 방지하려면 수신 엔드포인트에서 중단이 발생할 경우 데이터를 버퍼링할 수 있는 시스템으로 데이터를 전송해야 합니다. fluentbit을 사용하면 Amazon Kinesis Data Firehose를 사용하여 데이터를 일시적으로 유지할 수 있으므로 최종 데이터 스토리지 위치에 과부하가 걸릴 가능성을 줄일 수 있습니다.

워크로드

워크로드는 클러스터의 규모 조정에 영향을 미칩니다. Kubernetes APIs를 많이 사용하는 워크로드는 단일 클러스터에서 보유할 수 있는 총 워크로드 양을 제한하지만 부하를 줄이기 위해 변경할 수 있는 몇 가지 기본값이 있습니다.

Kubernetes 클러스터의 워크로드는 Kubernetes API와 통합되는 기능(예: Secrets 및 ServiceAccounts)에 액세스할 수 있지만 이러한 기능이 항상 필요한 것은 아니며 사용되지 않는 경우 비활성화해야 합니다. Kubernetes 컨트롤 플레인에 대한 워크로드 액세스 및 종속성을 제한하면 클러

스터에서 실행할 수 있는 워크로드 수가 증가하고 워크로드에 대한 불필요한 액세스를 제거하고 최소 권한 사례를 구현하여 클러스터의 보안이 향상됩니다. 자세한 내용은 보안 모범 사례를 참조하십시오.

포드 네트워킹에 IPv6 사용

VPC를 IPv4에서 IPv6로 전환할 수 없으므로 클러스터를 프로비저닝하기 전에 IPv6를 활성화하는 것이 중요합니다. VPC에서 IPv6를 활성화한다고 해서 IPv6를 사용해야 하는 것은 아니며 포드 및 서비스에서 IPv6를 사용하는 경우에도 IPv4 주소로 트래픽을 라우팅할 수 있습니다. 자세한 내용은 EKS 네트워킹 모범 사례를 참조하세요.

클러스터에서 IPv6를 사용하면 가장 일반적인 클러스터 및 워크로드 조정 제한 중 일부를 피할 수 있습니다. IPv6는 IP 주소를 사용할 수 없으므로 포드와 노드를 생성할 수 없는 IP 주소 소진을 방지합니다. 또한 노드당 ENI 연결 수를 줄여 포드가 IP 주소를 더 빠르게 수신하기 때문에 노드당 성능이 향상되었습니다. VPC CNI에서 IPv4 접두사 모드를 사용하여 유사한 노드 성능을 달성할 수 있지만 VPC에서 사용 가능한 IP 주소가 충분한지 확인해야 합니다.

네임스페이스당 서비스 수 제한

네임스페이스의 최대 서비스 수는 5,000개이고 클러스터의 최대 서비스 수는 10,000개입니다. 워크로드 및 서비스를 구성하고, 성능을 높이고, 네임스페이스 범위 리소스에 대한 계단식 영향을 방지하려면 네임스페이스당 서비스 수를 500개로 제한하는 것이 좋습니다.

kube-proxy를 사용하여 노드당 생성되는 IP 테이블 규칙 수는 클러스터의 총 서비스 수에 따라 증가합니다. 수천 개의 IP 테이블 규칙을 생성하고 이러한 규칙을 통해 패킷을 라우팅하면 노드에 성능 영향을 미치고 네트워크 지연 시간이 추가됩니다.

네임스페이스당 서비스 수가 500개 미만인 한 단일 애플리케이션 환경을 포함하는 Kubernetes 네임스페이스를 생성합니다. 이렇게 하면 서비스 검색 제한을 피할 수 있을 만큼 서비스 검색이 작게 유지되며 서비스 이름 충돌을 방지하는 데도 도움이 될 수 있습니다. 애플리케이션 환경(예: 개발, 테스트, 생산)은 네임스페이스 대신 별도의 EKS 클러스터를 사용해야 합니다.

Elastic Load Balancer 할당량 이해

서비스를 생성할 때 사용할 로드 밸런싱 유형(예: Network Load Balancer(NLB) 또는 Application Load Balancer(ALB))을 고려합니다. 각 로드 밸런서 유형은 서로 다른 기능을 제공하며 할당량이 다릅니다. 일부 기본 할당량은 조정할 수 있지만 변경할 수 없는 할당량 최대값이 있습니다. 계정 할당량 및 사용량을 보려면 AWS 콘솔에서 Service Quotas 대시보드를 확인합니다.

예를 들어 기본 ALB 대상은 1000입니다. 엔드포인트가 1,000개 이상인 서비스가 있는 경우 할당량을 늘리거나 서비스를 여러 ALBs로 분할하거나 Kubernetes Ingress를 사용해야 합니다. 기본 NLB 대상

은 3000이지만 AZ당 500개의 대상으로 제한됩니다. 클러스터가 NLB 서비스에 대해 500개 이상의 포드를 실행하는 경우 여러 AZs 사용하거나 할당량 한도 증가를 요청해야 합니다.

서비스에 연결된 로드 밸런서를 사용하는 대신 수신 컨트롤러를 사용하는 것이 좋습니다. AWS Load Balancer 컨트롤러는 수신 리소스에 대한 ALBs를 생성할 수 있지만 클러스터에서 전용 컨트롤러를 실행하는 것을 고려할 수 있습니다. 클러스터 내 수신 컨트롤러를 사용하면 클러스터 내에서 역방향 프록시를 실행하여 단일 로드 밸런서에서 여러 Kubernetes 서비스를 노출할 수 있습니다. 컨트롤러에는 Gateway API 지원과 같은 다양한 기능이 있으며, 이는 워크로드의 수와 크기에 따라 이점이 있을 수 있습니다.

Route 53, Global Accelerator 또는 CloudFront 사용

여러 로드 밸런서를 사용하는 서비스를 단일 엔드포인트로 사용하려면 Amazon CloudFront, AWS Global Accelerator 또는 Amazon Route 53를 사용하여 모든 로드 밸런서를 단일 고객 대상 엔드포인트로 노출해야 합니다. 각 옵션은 이점이 다르며 필요에 따라 개별적으로 또는 함께 사용할 수 있습니다.

Route 53는 여러 로드 밸런서를 공통 이름으로 노출하고 할당된 가중치에 따라 각 로드 밸런서에 트래픽을 전송할 수 있습니다. 설명서에서 DNS 가중치에 대한 자세한 내용을 읽고 AWS Load Balancer Controller 설명서에서 Kubernetes 외부 DNS 컨트롤러를 사용하여 이를 구현하는 방법을 읽을 수 있습니다.

Global Accelerator는 요청 IP 주소를 기반으로 워크로드를 가장 가까운 리전으로 라우팅할 수 있습니다. 이는 여러 리전에 배포된 워크로드에 유용할 수 있지만 단일 리전의 단일 클러스터로의 라우팅을 개선하지는 않습니다. Route 53를 Global Accelerator와 함께 사용하면 AZ를 사용할 수 없는 경우 상태 확인 및 자동 장애 조치와 같은 추가 이점이 있습니다. 이 블로그 게시물에서 Route 53과 함께 Global Accelerator를 사용하는 예를 볼 수 있습니다.

CloudFront는 Route 53 및 Global Accelerator와 함께 사용하거나 단독으로 트래픽을 여러 대상으로 라우팅하는 데 사용할 수 있습니다. CloudFront는 오리진 소스에서 제공되는 자산을 캐싱하므로 서비스 대상에 따라 대역폭 요구 사항이 줄어들 수 있습니다.

엔드포인트 대신 EndpointSlices 사용

서비스 레이블과 일치하는 포드를 검색할 때는 엔드포인트 대신 EndpointSlices를 사용해야 합니다. 엔드포인트는 소규모로 서비스를 노출하는 간단한 방법이지만 자동으로 확장되거나 업데이트가 있는 대규모 서비스는 Kubernetes 컨트롤 플레인에서 많은 트래픽을 발생시킵니다. EndpointSlices에는 토폴로지 인식 힌트와 같은 항목을 활성화하는 자동 그룹화가 있습니다.

모든 컨트롤러가 기본적으로 EndpointSlices를 사용하는 것은 아닙니다. 컨트롤러 설정을 확인하고 필요한 경우 활성화해야 합니다. AWS Load Balancer Controller의 경우 EndpointSlices를 사용하려면 `--enable-endpoint-slices` 선택적 플래그를 활성화해야 합니다.

가능하면 변경 불가능한 보안 암호 및 외부 보안 암호 사용

kubelet은 해당 노드의 포드 볼륨에 사용되는 보안 암호에 대한 현재 키 및 값의 캐시를 유지합니다. kubelet은 보안 암호에 대한 감시를 설정하여 변경 사항을 감지합니다. 클러스터가 확장됨에 따라 감시 수가 증가하면 API 서버 성능에 부정적인 영향을 미칠 수 있습니다.

Secrets의 시계 수를 줄이는 두 가지 전략이 있습니다.

- Kubernetes 리소스에 액세스할 필요가 없는 애플리케이션의 경우 automountServiceAccountToken: false를 설정하여 자동 탑재 서비스 계정 보안 암호를 비활성화할 수 있습니다.
- 애플리케이션의 보안 암호가 정적이고 향후 수정되지 않을 경우 보안 암호를 변경 불가능으로 표시합니다. kubelet은 변경할 수 없는 보안 암호에 대한 API 감시를 유지하지 않습니다.

포드에 서비스 계정 자동 탑재를 비활성화하려면 워크로드에서 다음 설정을 사용할 수 있습니다. 특정 워크로드에 서비스 계정이 필요한 경우 이러한 설정을 재정의할 수 있습니다.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app
automountServiceAccountToken: true
```

10,000개 제한을 초과하기 전에 클러스터의 보안 암호 수를 모니터링합니다. 다음 명령을 사용하여 클러스터의 총 보안 암호 수를 확인할 수 있습니다. 클러스터 모니터링 도구를 통해이 제한을 모니터링해야 합니다.

```
kubectl get secrets -A | wc -l
```

이 한도에 도달하기 전에 클러스터 관리자에게 알리도록 모니터링을 설정해야 합니다. Secrets Store CSI 드라이버와 함께 AWS Key Management Service(AWS KMS) 또는 Hashicorp Vault와 같은 외부 보안 암호 관리 옵션을 사용하는 것이 좋습니다. https://secrets-store-csi-driver.sigs.k8s.io/

배포 기록 제한

이전 객체는 여전히 클러스터에서 추적되므로 생성, 업데이트 또는 삭제 시 포드 속도가 느려질 수 있습니다. 배포revisionHistoryLimit의를 줄여 이전 ReplicaSets를 정리하면 Kubernetes Controller Manager에서 추적하는 총 객체 수가 줄어듭니다. 10의 배포에 대한 기본 기록 제한입니다.

클러스터가 CronJobs 또는 기타 메커니즘을 통해 많은 작업 객체를 생성하는 경우 ttlSecondsAfterFinished 설정을 사용하여 클러스터에서 이전 포드를 자동으로 정리해야 합니다. 이렇게 하면 지정된 시간 후에 작업 기록에서 성공적으로 실행된 작업이 제거됩니다.

기본적으로 enableServiceLinks 비활성화

포드가 노드에서 실행되면 kubelet은 각 활성 서비스에 대한 환경 변수 세트를 추가합니다. Linux 프로세스의 최대 환경 크기는 네임스페이스에 서비스가 너무 많으면 도달할 수 있습니다. 네임스페이스당 서비스 수는 5,000개를 초과할 수 없습니다. 그런 다음 서비스 환경 변수 수가 쉘 제한을 초과하여 시작 시 포드가 충돌합니다.

포드가 서비스 검색에 서비스 환경 변수를 사용해서는 안 되는 다른 이유가 있습니다. 환경 변수 이름 충돌, 누출되는 서비스 이름 및 총 환경 크기는 몇 가지입니다. 서비스 엔드포인트를 검색하려면 CoreDNS를 사용해야 합니다.

리소스당 동적 승인 웹후크 제한

동적 승인 웹후크에는 승인 웹후크와 변형 웹후크가 포함됩니다. 리소스가 Kubernetes API로 전송될 때 순차적으로 호출되는 Kubernetes 제어 플레인의 일부가 아닌 API 엔드포인트입니다. 각 웹후크의 기본 제한 시간은 10초이며 여러 웹후크 또는 제한 시간이 있는 경우 API 요청이 걸리는 시간을 늘릴 수 있습니다.

특히 AZ 인시던트 발생 시 웹후크의 가용성이 높고 리소스를 거부하거나 실패를 무시하도록 failurePolicy가 올바르게 설정되어 있는지 확인합니다. 필요하지 않은 경우 --dry-run kubectl 명령이 웹후크를 우회하도록 허용하여 웹후크를 호출하지 마십시오.

```
apiVersion: admission.k8s.io/v1
kind: AdmissionReview
request:
  dryRun: False
```

웹후크를 변경하면 리소스를 연속해서 수정할 수 있습니다. 변형 웹후크가 5개 있고 50개 리소스 등을 배포하면 압축이 실행될 때까지 각 리소스의 모든 버전이 5분마다 저장되어 수정된 리소스의 이전 버

전을 제거합니다. etcd가 대체된 리소스를 제거하는이 시나리오에서는 etcd에서 200개의 리소스 버전이 제거되며 리소스 크기에 따라 조각 모음이 15분마다 실행될 때까지 etcd 호스트에서 상당한 공간을 사용할 수 있습니다.

이 조각 모음으로 인해 etcd가 일시 중지되어 Kubernetes API 및 컨트롤러에 다른 영향을 미칠 수 있습니다. 대규모 리소스를 자주 수정하거나 수백 개의 리소스를 빠르게 연속해서 수정하지 않아야 합니다.

여러 클러스터의 워크로드 비교

성능이 비슷해야 하지만 그렇지 않은 클러스터가 두 개 있는 경우 지표를 비교하여 이유를 식별해 보세요.

예를 들어 클러스터 지연 시간을 비교하는 것은 일반적인 문제입니다. 이는 일반적으로 API 요청 볼륨의 차이로 인해 발생합니다. 다음 CloudWatch LogInsight 쿼리를 실행하여 차이를 이해할 수 있습니다.

```
filter @logStream like "kube-apiserver-audit"
| stats count(*) as cnt by objectRef.apiGroup, objectRef.apiVersion,
 objectRef.resource, userAgent, verb, responseStatus.code
| sort cnt desc
| limit 1000
```

필터를 추가하여 범위를 좁힐 수 있습니다. 예를 들어의 모든 목록 요청에 초점을 맞출 수 있습니다`foo`.

```
filter @logStream like "kube-apiserver-audit"
| filter verb = "list"
| filter user.username like "foo"
| stats count(*) as cnt by objectRef.apiGroup, objectRef.apiVersion,
 objectRef.resource, responseStatus.code
| sort cnt desc
| limit 1000
```

Kubernetes 조정 이론

노드와 이탈률 비교

종종 Kubernetes의 확장성에 대해 논의할 때 단일 클러스터에 있는 노드 수를 기준으로 이를 수행합니다. 흥미롭게도 확장성을 이해하는 데 가장 유용한 지표는 거의 없습니다. 예를 들어 포드 수가 많지만 고정된 5,000개의 노드 클러스터는 초기 설정 후 컨트롤 플레인에 많은 스트레스를 주지 않습니다. 그

러나 1,000개의 노드 클러스터를 가져와서 1분 이내에 10,000개의 단기 작업을 생성하려고 하면 컨트롤 플레인에 상당한 지속적인 부담을 줄 수 있습니다.

노드 수를 사용하여 조정을 이해하는 것만으로도 오해의 소지가 있습니다. 특정 기간 내에 발생하는 변화율을 고려하는 것이 좋습니다(이 논의에는 5분 간격을 사용하겠습니다. 이것이 Prometheus 쿼리가 일반적으로 사용하는 것이기 때문입니다). 변경 속도 측면에서 문제를 프레이밍하면 원하는 규모를 달성하기 위해 조정해야 할 사항을 더 잘 파악할 수 있는 이유를 살펴보겠습니다.

초당 쿼리로 생각하기

Kubernetes에는 Kubernetes 체인의 다음 링크를 압도하지 않도록 Kubelet, Scheduler, Kube Controller Manager, API 서버 등 각 구성 요소에 대한 여러 보호 메커니즘이 있습니다. 예를 들어 Kubelet에는 특정 속도로 API 서버에 대한 호출을 제한하는 플래그가 있습니다. 이러한 보호 메커니즘은 일반적으로 초당 또는 QPS로 허용되는 쿼리로 표현되지만 항상 표현되는 것은 아닙니다.

이러한 QPS 설정을 변경할 때는 각별히 주의해야 합니다. Kubelet에서 초당 쿼리와 같은 하나의 병목 현상을 제거하면 다른 다운스트림 구성 요소에 영향을 미칩니다. 이는 시스템을 특정 속도 이상으로 압도할 수 있으므로 Kubernetes에서 워크로드를 성공적으로 조정하려면 서비스 체인의 각 부분을 이해하고 모니터링하는 것이 중요합니다.

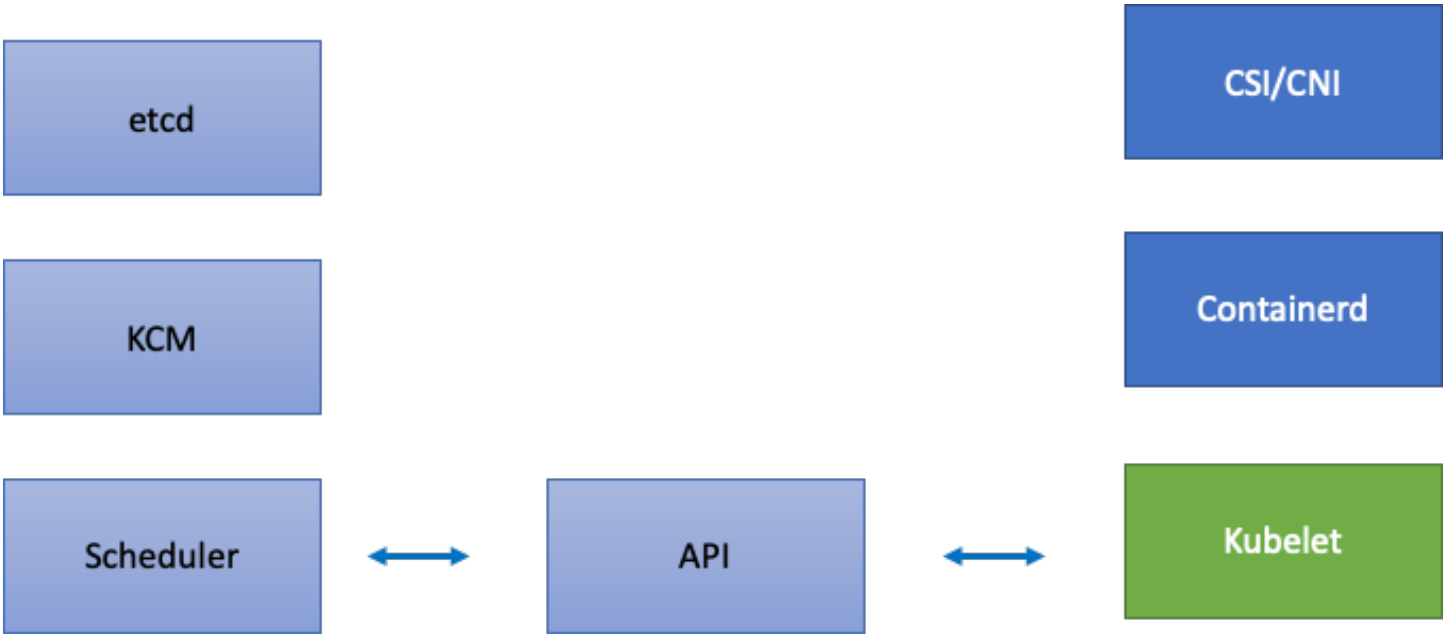
> **Note**
>
> API 서버는 API 우선 순위와 공정성을 도입하여 더 복잡한 시스템을 갖추고 있으며, 이에 대해서는 별도로 설명하겠습니다.

> **Note**
>
> 주의, 일부 지표는 적합한 것 같지만 실제로 다른 지표를 측정하고 있습니다. 예를 들어는 Kubelet에서 apiserver로의 요청 수가 아니라 Kubelet의 지표 서버와만 `kubelet_http_inflight_requests` 관련이 있습니다. 이로 인해 Kubelet에서 QPS 플래그를 잘못 구성할 수 있습니다. 특정 Kubelet에 대한 감사 로그 쿼리는 지표를 확인하는 보다 신뢰할 수 있는 방법입니다.

분산 구성 요소 조정

EKS는 관리형 서비스이므로 Kubernetes 구성 요소를 etcd, Kube Controller Manager, Scheduler(다이어그램의 왼쪽 부분)를 포함하는 AWS 관리형 구성 요소와 Kubelet, Container Runtime, 네트워킹 및

스토리지 드라이버(다이어그램의 오른쪽 부분)와 같은 AWS APIs를 호출하는 다양한 연산자와 같은 고객 구성 가능한 구성 요소의 두 가지 범주로 분할해 보겠습니다. 고객이 API 우선 순위 및 공정성 설정을 구성할 수 있으므로 AWS 관리형이더라도 API 서버를 중간에 둡니다.



업스트림 및 다운스트림 병목 현상

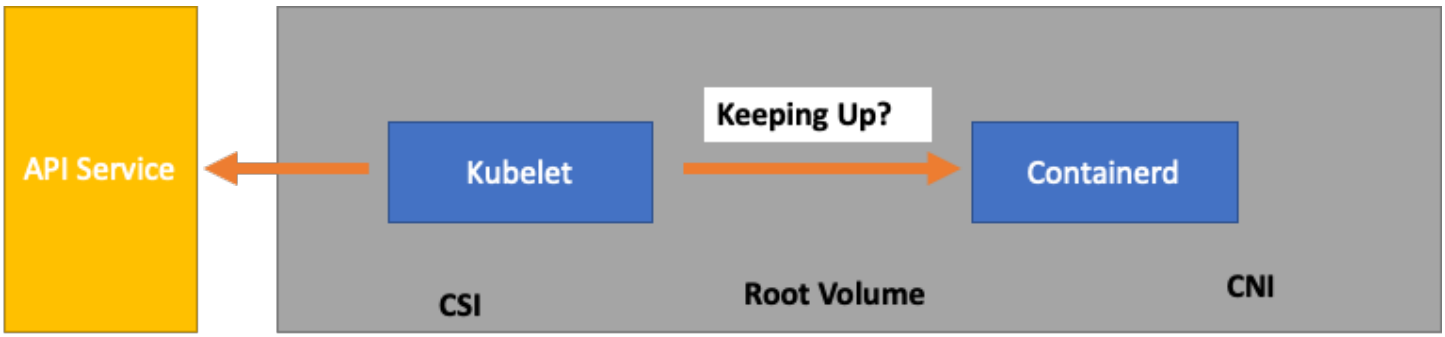
각 서비스를 모니터링할 때는 병목 현상을 찾기 위해 지표를 양방향으로 살펴보는 것이 중요합니다. Kubelet을 예로 사용하여 이를 수행하는 방법을 살펴보겠습니다. Kubelet은 API 서버와 컨테이너 런타임에 대해 모두 설명합니다. 두 구성 요소 중 하나에 문제가 있는지 여부를 감지하기 위해 모니터링해야 할 사항과 방법은 무엇입니까?

노드당 포드 수

노드에서 실행할 수 있는 포드 수와 같은 조정 수를 살펴보면 업스트림이 얼굴 값으로 지원하는 노드당 포드 110개를 가져올 수 있습니다.

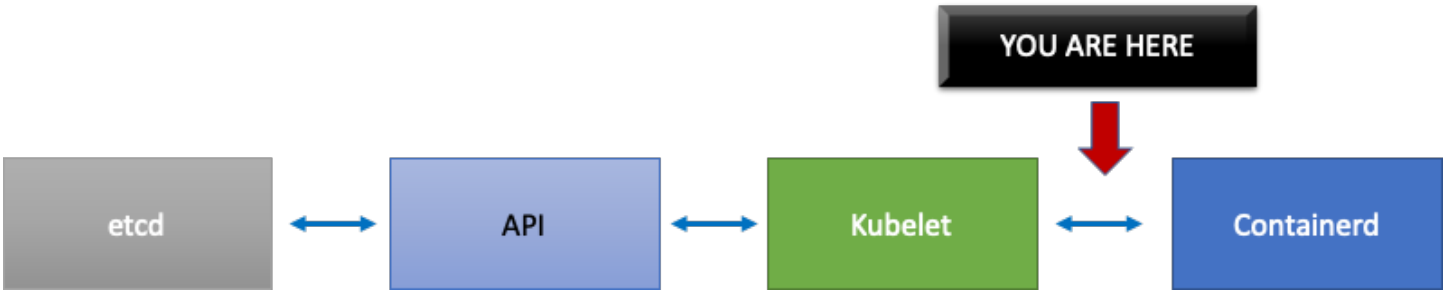
> **Note**
>
> https://kubernetes.io/docs/setup/best-practices/cluster-large/

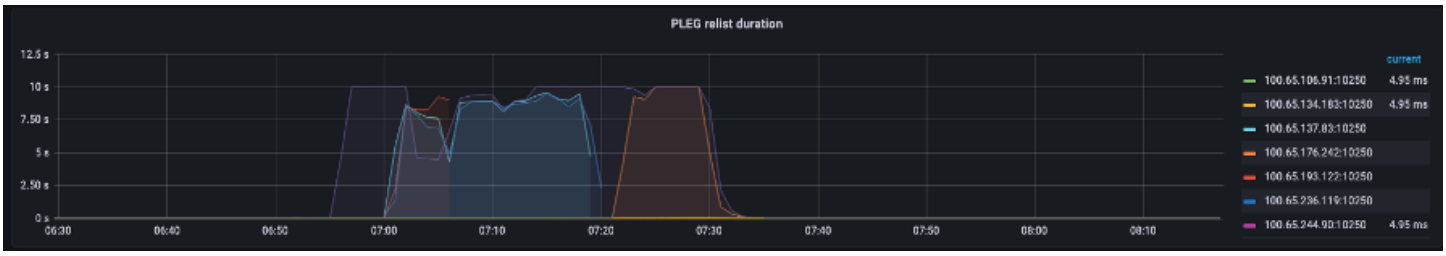
그러나 워크로드는 업스트림의 확장성 테스트에서 테스트한 것보다 더 복잡할 수 있습니다. 프로덕션 환경에서 실행하려는 포드 수를 서비스할 수 있도록 Kubelet이 Containerd 런타임과 함께 "유지" 중인지 확인해 보겠습니다.



간소화를 위해 Kubelet은 컨테이너 런타임(이 경우 Containerd)에서 포드의 상태를 가져옵니다. 상태를 너무 빨리 변경하는 포드가 너무 많으면 어떻게 해야 하나요? 변경 속도가 너무 높으면 [컨테이너 런타임에 대한] 요청 시간이 초과될 수 있습니다.

> **Note**
>
> Kubernetes는 지속적으로 진화하고 있으며,이 하위 시스템은 현재 변경 중입니다. https://github.com/kubernetes/enhancements/issues/3386





위 그래프에는 포드 수명 주기 이벤트 생성 기간 지표의 제한 시간 값에 도달했음을 나타내는 평면이 표시됩니다. 자체 클러스터에서 이를 보려면 다음 PromQL 구문을 사용할 수 있습니다.

```
increase(kubelet_pleg_relist_duration_seconds_bucket{instance="$instance"}
[$__rate_interval])
```

이 제한 시간 동작이 관찰되면 노드가 가능한 한도 이상으로 푸시되었음을 알 수 있습니다. 계속 진행하기 전에 제한 시간의 원인을 해결해야 합니다. 이는 노드당 포드 수를 줄이거나 재시도량이 많아질 수 있는 오류(즉 이탈률에 영향을 미치는 오류)를 찾아낼 수 있습니다. 중요한 점은 지표가 노드가 할당된 포드의 이탈률을 처리할 수 있는지와 고정된 수를 사용할 수 있는지를 이해하는 가장 좋은 방법이라는 것입니다.

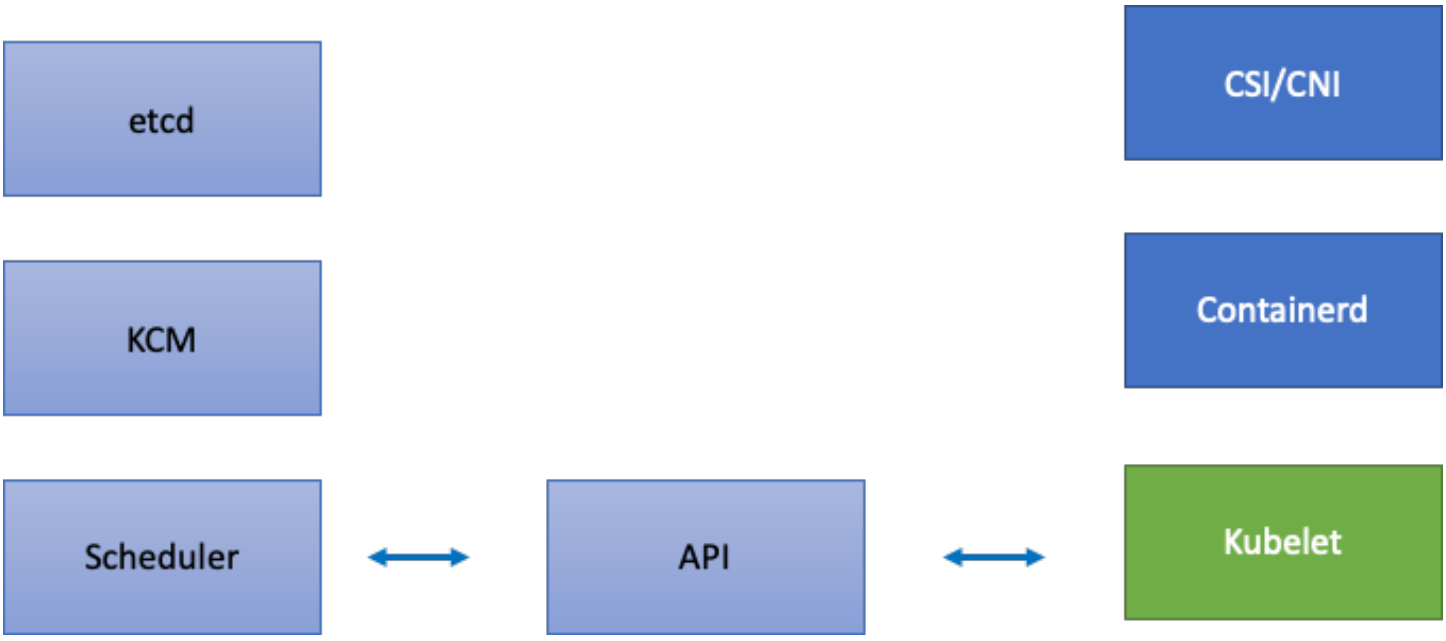
지표별 규모 조정

지표를 사용하여 시스템을 최적화하는 개념은 오래된 개념이지만 사람들이 Kubernetes 여정을 시작할 때 간과되는 경우가 많습니다. 특정 숫자(예: 노드당 포드 110개)에 초점을 맞추는 대신 시스템에서 병목 현상을 찾는 데 도움이 되는 지표를 찾는 데 노력을 집중합니다. 이러한 지표에 적합한 임계값을 이해하면 시스템이 최적으로 구성되어 있다는 높은 신뢰도를 얻을 수 있습니다.

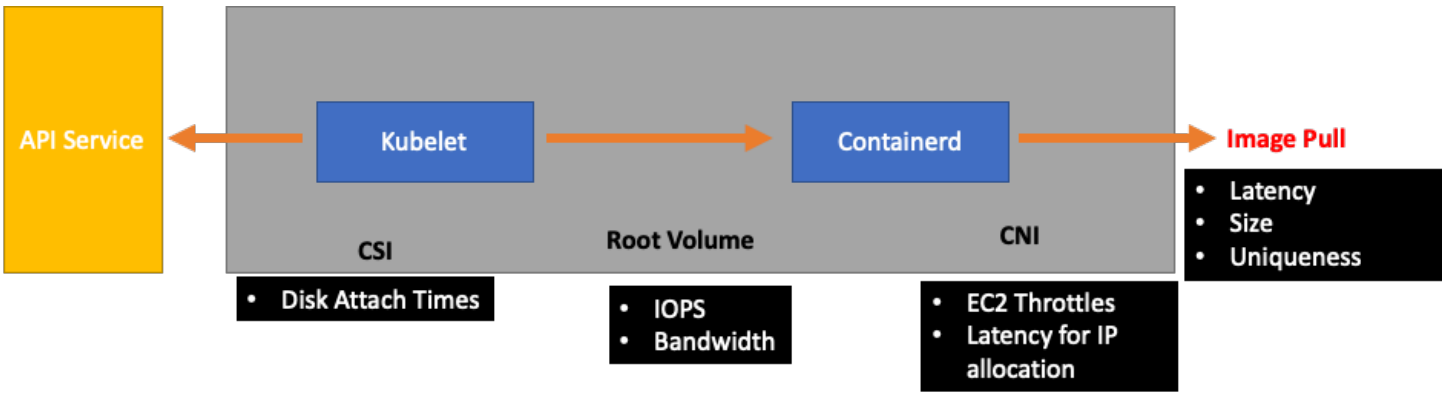
변경 사항의 영향

문제를 일으킬 수 있는 일반적인 패턴은 의심되는 첫 번째 지표 또는 로그 오류에 초점을 맞추는 것입니다. Kubelet이 이전에 시간 초과된 것을 확인했을 때 Kubelet이 전송할 수 있는 초당 속도 증가 등과 같은 무작위 작업을 시도할 수 있습니다. 하지만 먼저 발견된 오류의 다운스트림에 있는 모든 것을 전체적으로 살펴보는 것이 좋습니다. 용도에 따라 데이터를 기반으로 각 변경을 수행합니다.

Kubelet의 다운스트림은 컨테이너식 런타임(포드 오류), 스토리지 드라이버(CSI)와 같은 DaemonSets 및 EC2 API와 통신하는 네트워크 드라이버(CNI) 등이 됩니다.



런타임을 따라가지 않는 Kubelet의 이전 예를 계속 살펴보겠습니다. 노드를 너무 밀집하여 오류를 트리거할 수 있는 여러 지점이 있습니다.



워크로드에 적합한 노드 크기를 설계할 때 이러한 신호는 시스템에 불필요한 부담을 줄 수 있는 easy-to-overlook 신호이므로 규모와 성능이 모두 제한됩니다.

불필요한 오류의 비용

Kubernetes 컨트롤러는 오류 조건이 발생할 때 재시도하는 데 뛰어나지만 비용이 발생합니다. 이러한 재시도는 Kube Controller Manager와 같은 구성 요소에 대한 부담을 높일 수 있습니다. 이러한 오류를 모니터링하는 것은 규모 테스트의 중요한 테넌트입니다.

오류가 더 적게 발생하면 시스템에서 스팟 문제가 더 쉬워집니다. 주요 작업(예: 업그레이드) 전에 클러스터에 오류가 없는지 주기적으로 확인하면 예상치 못한 이벤트가 발생할 때 로그 문제 해결을 간소화할 수 있습니다.

보기 확장

수천 개의 노드가 있는 대규모 클러스터에서는 병목 현상을 개별적으로 찾고 싶지 않습니다. PromQL에서는 topk라는 함수를 사용하여 데이터 세트에서 가장 높은 값을 찾을 수 있습니다. K는 변수이므로 원하는 항목 수를 배치합니다. 여기서는 세 개의 노드를 사용하여 클러스터의 모든 Kubelets가 포화 상태인지 여부를 확인합니다. 지금까지 지연 시간을 살펴보았습니다. 이제 Kubelet에서 이벤트를 삭제하는지 살펴보겠습니다.

```
topk(3, increase(kubelet_pleg_discard_events{}[$__rate_interval]))
```

이 문을 분석합니다.

- Grafana 변수를 사용하여 필요한 4개의 샘플을 `$__rate_interval` 가져오도록 합니다. 이렇게 하면 간단한 변수를 사용한 모니터링에서 복잡한 주제를 우회할 수 있습니다.
- `topk`는 최상위 결과만 제공하고 숫자 3은 해당 결과를 3으로 제한합니다. 이는 클러스터 전체 지표에 유용한 함수입니다.

- {} 필터가 없다고 알려주세요. 일반적으로 스크레이핑 규칙의 작업 이름을 입력하지만 이러한 이름은 다르기 때문에 비워 둡니다.

문제를 절반으로 분할

시스템의 병목 현상을 해결하기 위해 업스트림 또는 다운스트림에 문제가 있음을 보여주는 지표를 찾는 접근 방식을 취합니다. 이렇게 하면 문제를 절반으로 분할할 수 있기 때문입니다. 또한 지표 데이터를 표시하는 방식의 핵심 원칙이기도 합니다.

이 프로세스로 시작하는 좋은 방법은 API 서버입니다. 클라이언트 애플리케이션 또는 컨트롤 플레인에 문제가 있는지 확인할 수 있기 때문입니다.

컨트롤 플레인 모니터링

API 서버

API 서버를 볼 때는 제어 영역의 과부하를 방지하기 위해 인바운드 요청을 제한하는 기능 중 하나라는 점을 기억해야 합니다. API 서버 수준에서 병목 현상처럼 보일 수 있는 것은 실제로 더 심각한 문제로부터 병목 현상을 보호하는 것일 수 있습니다. 시스템을 통해 이동하는 요청의 양을 늘리는 장단점을 고려해야 합니다. API 서버 값을 늘려야 하는지 여부를 결정하려면 다음 사항에 유의해야 합니다.

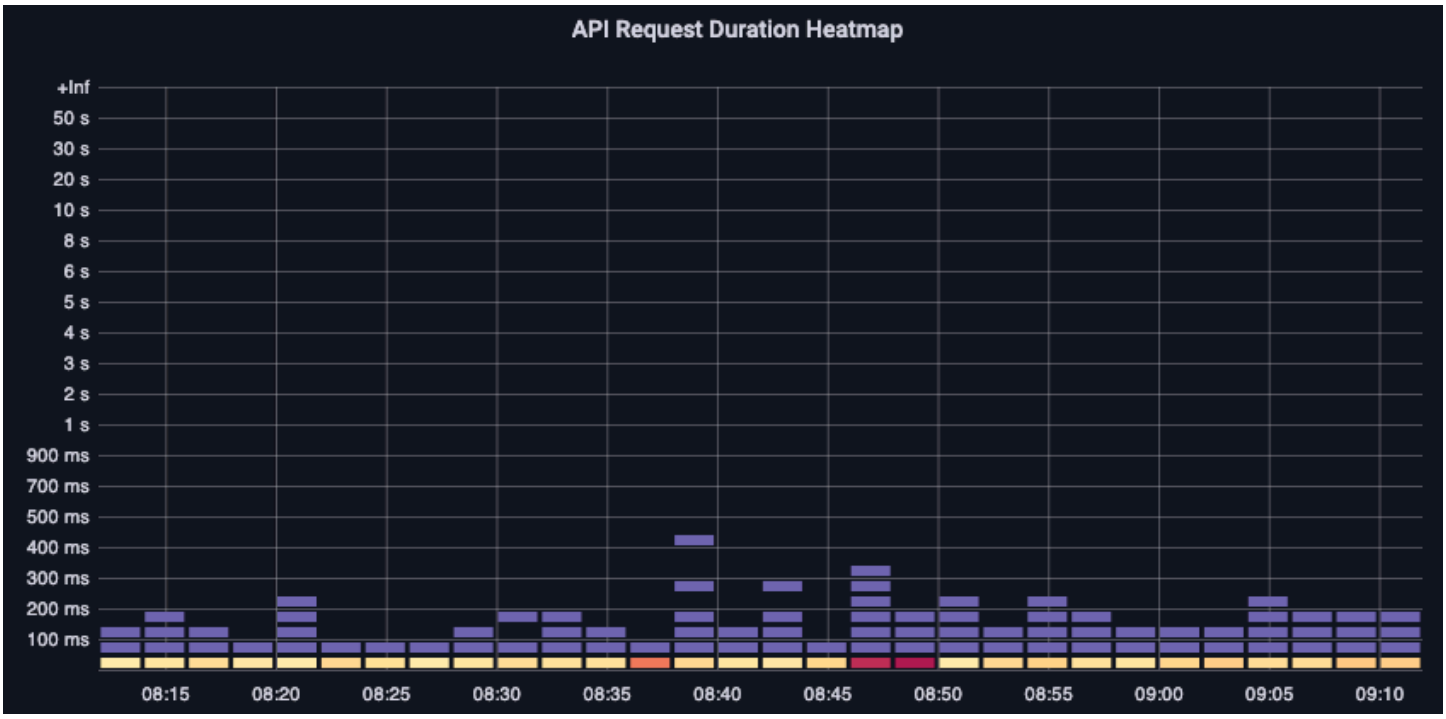
1. 시스템을 통해 이동하는 요청의 지연 시간은 어떻게 됩니까?
2. 이 지연 시간은 API 서버 자체입니까, 아니면 etcd와 같은 "다운스트림"입니까?
3. API 서버 대기열 깊이가이 지연 시간의 요인입니까?
4. 원하는 API 호출 패턴에 대해 API 우선 순위 및 공정성(APF) 대기열이 올바르게 설정되어 있습니까?

문제는 어디에 있나요?

먼저 API 지연 시간에 대한 지표를 사용하여 API 서버가 요청을 처리하는 데 걸리는 시간을 파악할 수 있습니다. 아래 PromQL 및 Grafana 히트맵을 사용하여이 데이터를 표시해 보겠습니다.

```
max(increase(apiserver_request_duration_seconds_bucket{subresource!
="status",subresource!="token",subresource!="scale",subresource!="/
healthz",subresource!="binding",subresource!="proxy",verb!="WATCH"}[$__rate_interval]))
 by (le)
```

> **Note**
>
> 이 문서에 사용된 API 대시보드를 사용하여 API 서버를 모니터링하는 방법에 대한 자세한 내용은 다음 블로그를 참조하세요.



이러한 요청은 모두 1초 표시 아래에 있으며, 이는 컨트롤 플레인이 적시에 요청을 처리하고 있음을 나타내는 좋은 표시입니다. 하지만 그렇지 않다면 어떻게 해야 할까요?

위의 API 요청 기간에서 사용하는 형식은 히트맵입니다. 히트맵 형식의 장점은 기본적으로 API의 제한 시간 값(60초)을 알려주는 것입니다. 그러나 진정으로 알아야 할 것은 제한 시간 임계값에 도달하기 전에이 값이 우려해야 하는 임계값입니다. 허용되는 임계값에 대한 대략적인 지침을 보려면 여기에서 찾을 수 있는 업스트림 Kubernetes SLO를 사용할 수 있습니다.

> **Note**
>
> 이 문에서 최대 함수를 확인하시겠습니까? 여러 서버(기본적으로 EKS의 API 서버 2개)를 집계하는 지표를 사용하는 경우 해당 서버의 평균을 구하지 않는 것이 중요합니다.

비대칭 트래픽 패턴

한 API 서버[포드]가 약간 로드되고 다른 API 서버는 많이 로드되면 어떻게 되나요? 이 두 숫자의 평균을 구하면 어떤 일이 발생했는지 잘못 해석할 수 있습니다. 예를 들어 여기서는 3개의 API 서버가 있지만 모든 로드는 이러한 API 서버 중 하나에 있습니다. 일반적으로 규모 및 성능 문제를 투자할 때는 etcd 및 API 서버와 같이 여러 서버가 있는 모든 것을 구분해야 합니다.



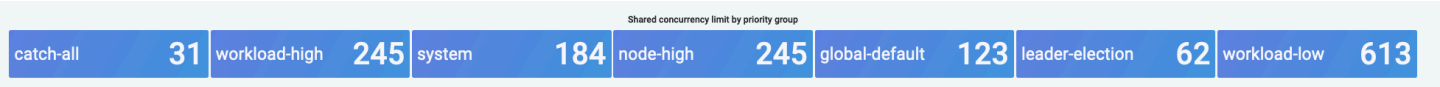
API 우선 순위 및 공정성으로 이동하면 시스템의 총 요청 수는 API 서버가 초과 구독되었는지 확인하는 한 가지 요소일 뿐입니다. 이제 시스템이 일련의 대기열에서 작동하므로 이러한 대기열이 가득 찼는지, 해당 대기열의 트래픽이 삭제되고 있는지 확인해야 합니다.

다음 쿼리를 사용하여 이러한 대기열을 살펴보겠습니다.

```
max without(instance)(apiserver_flowcontrol_nominal_limit_seats{})
```

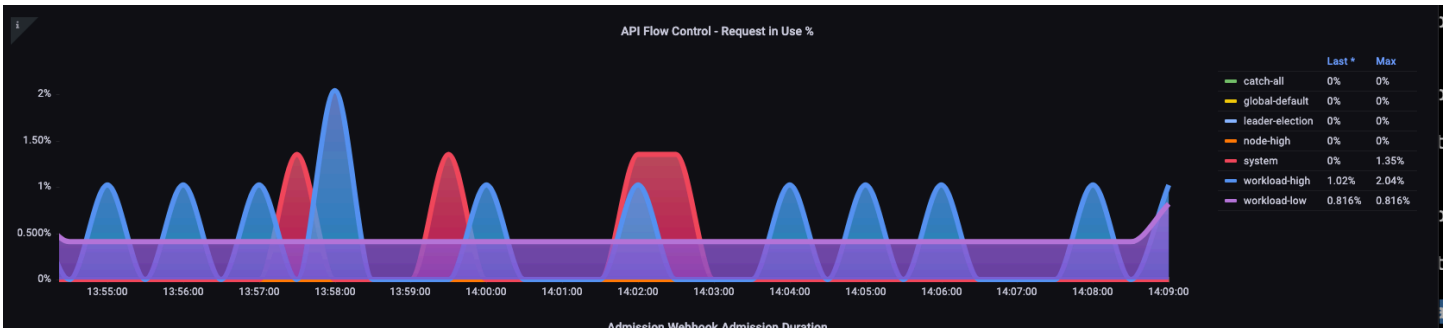
> **Note**
>
> API A&F 작동 방식에 대한 자세한 내용은 다음 모범 사례 가이드를 참조하세요.

다음은 클러스터에서 기본적으로 제공되는 7개의 서로 다른 우선 순위 그룹입니다.



그런 다음 특정 우선 순위 수준이 포화되고 있는지 이해할 수 있도록 해당 우선 순위 그룹의 몇 퍼센트가 사용되고 있는지 확인하고자 합니다. 워크로드 낮음 수준에서 요청을 제한하는 것이 바람직하지만 리더 선출 수준에서는 삭제되지 않습니다.

API 우선순위 및 공정성(APF) 시스템에는 여러 가지 복잡한 옵션이 있으며, 이러한 옵션 중 일부는 의도하지 않은 결과를 초래할 수 있습니다. 필드에 표시되는 일반적인 문제는 대기열 깊이를 불필요한 지연 시간을 추가하기 시작하는 지점까지 늘리는 것입니다. `apiserver_flowcontrol_current_inqueue_request` 지표를 사용하여이 문제를 모니터링할 수 있습니다. 를 사용하여 드롭을 확인할 수 있습니다`apiserver_flowcontrol_rejected_requests_total`. 버킷이 동시성을 초과하는 경우 이러한 지표는 0이 아닌 값이 됩니다.



대기열 깊이를 늘리면 API 서버가 상당한 지연 시간 소스가 될 수 있으므로 주의해서 수행해야 합니다. 생성된 대기열 수에 신중을 기하는 것이 좋습니다. 예를 들어 EKS 시스템의 공유 수는 600개입니다. 대기열을 너무 많이 생성하면 리더-선택 대기열 또는 시스템 대기열과 같이 처리량이 필요한 중요한 대기열의 공유를 줄일 수 있습니다. 대기열을 너무 많이 생성하면 이러한 대기열의 크기를 올바르게 조정하기가 더 어려울 수 있습니다.

APF에서 수행할 수 있는 단순하고 영향력 있는 변경에 초점을 맞추기 위해 활용도가 낮은 버킷에서 공유를 가져와 최대 사용량에 있는 버킷의 크기를 늘리기만 하면 됩니다. 이러한 버킷 간에 공유를 지능적으로 재배포하면 드롭의 가능성을 줄일 수 있습니다.

자세한 내용은 EKS 모범 사례 안내서의 API 우선순위 및 공정성 설정을 참조하세요.

API 및 etcd 지연 시간

API 서버의 지표/로그를 사용하여 API 서버에 문제가 있는지, API 서버의 업스트림/다운스트림에 문제가 있는지 또는 둘의 조합이 있는지 확인하는 방법. 이를 더 잘 이해하기 위해 API Server와 etcd의 관련성과 잘못된 시스템 문제를 해결하는 것이 얼마나 쉬운지 살펴보겠습니다.

아래 차트에서는 API 서버 지연 시간을 볼 수 있지만, 그래프의 막대가 etcd 수준에서 대부분의 지연 시간을 보여주기 때문에 이러한 지연 시간의 대부분은 etcd 서버와 상관관계가 있습니다. 15초의 etcd 지연 시간이 동시에 20초의 API 서버 지연 시간이 있는 경우 대부분의 지연 시간은 실제로 etcd 수준에 있습니다.

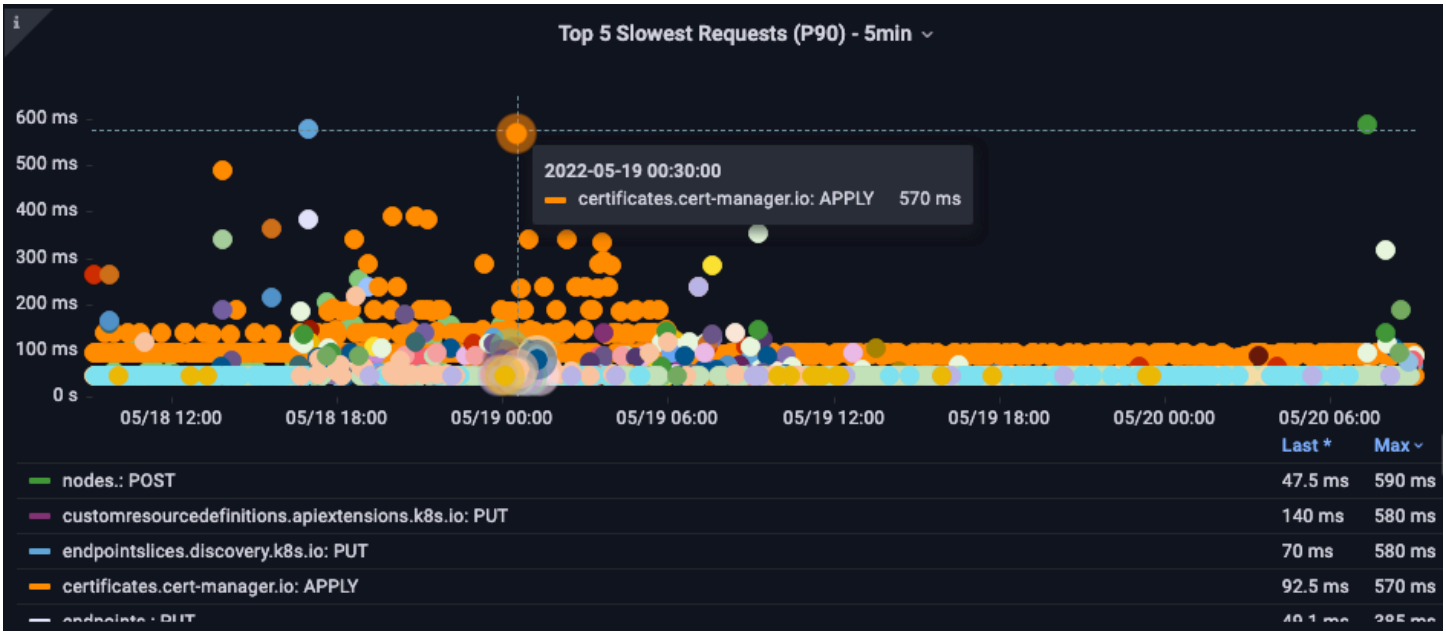
전체 흐름을 살펴보면 API 서버에만 집중하지 않고 etcd가 부족하다는 신호(예: 느린 적용 카운터 증가)도 찾는 것이 좋습니다. 대시보드를 강력하게 만들려면 한 눈에 적절한 문제 영역으로 빠르게 이동할 수 있어야 합니다.

> **Note**
>
> 섹션의 대시보드는 https://github.com/RiskyAdventure/Troubleshooting-Dashboards/blob/main/api-troubleshooter.json 찾을 수 있습니다.



컨트롤 플레인과 클라이언트 측 문제 비교

이 차트에서는 해당 기간 동안 완료하는 데 가장 많은 시간이 걸린 API 직접 호출을 찾고 있습니다. 이 경우 사용자 지정 리소스(CRD)가 05:40 기간 동안 가장 지연된 호출인 APPLY 함수를 호출하는 것을 볼 수 있습니다.



이 데이터로 무장하면 Ad-Hoc PromQL 또는 CloudWatch Insights 쿼리를 사용하여 해당 기간 동안 감사 로그에서 LIST 요청을 가져와 어떤 애플리케이션인지 확인할 수 있습니다.

CloudWatch를 사용하여 소스 찾기

지표는 살펴보려는 문제 영역을 찾고 문제의 기간과 검색 파라미터를 모두 좁히는 데 가장 적합합니다. 이 데이터가 확보되면 더 자세한 시간과 오류를 위해 로그로 전환하려고 합니다. 이를 위해 CloudWatch Logs Insights를 사용하여 로그를 지표로 변환합니다.

예를 들어 위의 문제를 조사하기 위해 다음 CloudWatch Logs Insights 쿼리를 사용하여 userAgent 및 requestURI를 가져와서 이러한 지연 시간을 유발하는 애플리케이션을 고정할 수 있습니다.

> **Note**
>
> 시계에서 정상적인 목록/재동기 동작을 가져오지 않으려면 적절한 개수를 로 사용해야 합니다.

```
fields @timestamp, @message
| filter @logStream like "kube-apiserver-audit"
| filter ispresent(requestURI)
| filter verb = "list"
| parse requestReceivedTimestamp /\d+-\d+-(?<StartDay>\d+)T(?<StartHour>\d+):(?
<StartMinute>\d+):(?<StartSec>\d+).(?<StartMsec>\d+)Z/
| parse stageTimestamp /\d+-\d+-(?<EndDay>\d+)T(?<EndHour>\d+):(?<EndMinute>\d+):(?
<EndSec>\d+).(?<EndMsec>\d+)Z/
| fields (StartHour * 3600 + StartMinute * 60 + StartSec + StartMsec / 1000000) as
 StartTime, (EndHour * 3600 + EndMinute * 60 + EndSec + EndMsec / 1000000) as EndTime,
 (EndTime - StartTime) as DeltaTime
| stats avg(DeltaTime) as AverageDeltaTime, count(*) as CountTime by requestURI,
 userAgent
| filter CountTime >=50
| sort AverageDeltaTime desc
```

이 쿼리를 사용하여 많은 수의 지연 시간이 긴 목록 작업을 실행하는 두 개의 에이전트를 찾았습니다. Splunk 및 CloudWatch 에이전트. 데이터를 사용하여이 컨트롤러를 제거, 업데이트 또는 다른 프로젝트로 대체하기로 결정할 수 있습니다.

requestURI	AverageDeltaTime	AverageD	CountTime
/api/v1/pods	rest-client/2.1.0 (linux-gnu x86_64) ruby/2.5.5p157	9.1395	87
/api/v1/pods?limit=500&resourceVersion=0	amazon-cloudwatch-agent/v0.0.0 (linux/amd64) kubernetes/$Format	5.3976	929

> **Note**
>
> 이 주제에 대한 자세한 내용은 다음 블로그를 참조하세요.

스케줄러

EKS 컨트롤 플레인 인스턴스는 별도의 AWS 계정에서 실행되므로 지표에 대해 해당 구성 요소를 스크레이프할 수 없습니다(API 서버는 예외임). 그러나 이러한 구성 요소에 대한 감사 로그에 액세스할 수 있으므로 이러한 로그를 지표로 전환하여 하위 시스템 중 하나에서 조정 병목 현상이 발생하는지 확인할 수 있습니다. CloudWatch Logs Insights를 사용하여 스케줄러 대기열에 있는 예약되지 않은 포드 수를 확인해 보겠습니다.

스케줄러 로그의 예약되지 않은 포드

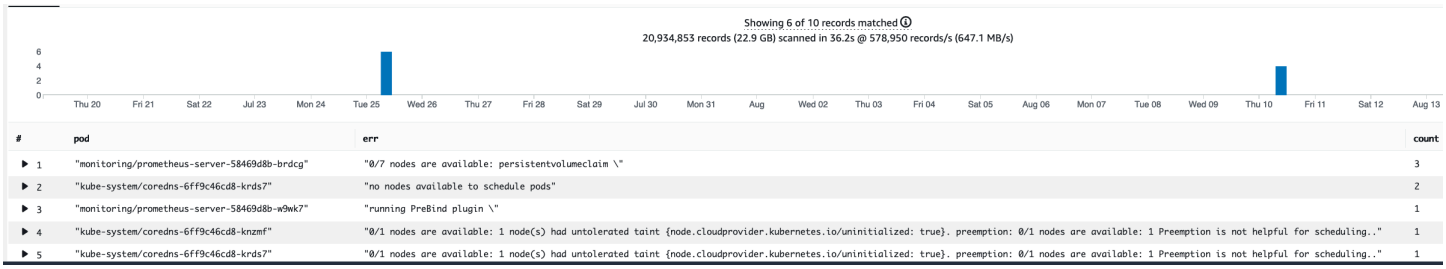
자체 관리형 Kubernetes(예: Kops)에서 스케줄러 지표를 직접 스크레이프할 수 있는 액세스 권한이 있는 경우 다음 PromQL을 사용하여 스케줄러 백로그를 이해합니다.

```
max without(instance)(scheduler_pending_pods)
```

EKS에서 위의 지표에 액세스할 수 없으므로 아래 CloudWatch Logs Insights 쿼리를 사용하여 특정 기간 동안 예약되지 않은 포드 수를 확인하여 백로그를 확인합니다. 그런 다음 피크 시간대에 메시지를 자세히 살펴보고 병목 현상의 특성을 이해할 수 있습니다. 예를 들어 노드가 충분히 빠르게 실행되지 않거나 스케줄러 자체의 속도 제한기입니다.

```
fields timestamp, pod, err, @message
| filter @logStream like "scheduler"
| filter @message like "Unable to schedule pod"
| parse @message  /^.(?<date>\d{4})\s+(?<timestamp>\d+:\d+:\d+\.\d+)\s+\S*\s+\S+\]\s
\"(.*?)\"\s+pod=(?<pod>\"(.*?)\")\s+err=(?<err>\"(.*?)\")/
| count(*) as count by pod, err
| sort count desc
```

여기서는 스토리지 PVC를 사용할 수 없어 포드가 배포되지 않았다는 스케줄러의 오류를 볼 수 있습니다.



> **Note**
>
> 이 함수를 활성화하려면 컨트롤 플레인에서 감사 로깅을 활성화해야 합니다. 또한 시간 경과에 따른 비용을 불필요하게 증가시키지 않도록 로그 보존을 제한하는 것이 모범 사례입니다. 아래 EKSCTL 도구를 사용하여 모든 로깅 함수를 켜는 예제입니다.

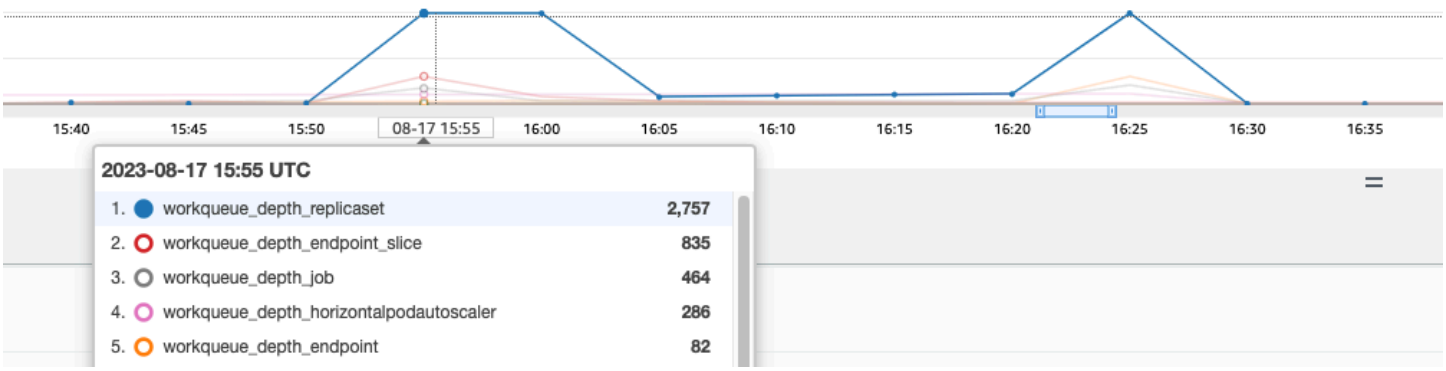
```
cloudWatch:
  clusterLogging:
    enableTypes: ["*"]
    logRetentionInDays: 10
```

Kube 컨트롤러 관리자

Kube Controller Manager는 다른 모든 컨트롤러와 마찬가지로 한 번에 수행할 수 있는 작업 수에 제한이 있습니다. 이러한 파라미터를 설정할 수 있는 KOPS 구성을 살펴보면서 이러한 플래그의 일부를 살펴보겠습니다.

```
  kubeControllerManager:
    concurrentEndpointSyncs: 5
    concurrentReplicasetSyncs: 5
    concurrentNamespaceSyncs: 10
    concurrentServiceaccountTokenSyncs: 5
    concurrentServiceSyncs: 5
    concurrentResourceQuotaSyncs: 5
    concurrentGcSyncs: 20
    kubeAPIBurst: 30
    kubeAPIQPS: "20"
```

이러한 컨트롤러에는 클러스터의 이탈률이 높은 시간 동안 채워지는 대기열이 있습니다. 이 경우 복제본 세트 컨트롤러의 대기열에 큰 백로그가 있는 것을 볼 수 있습니다.



이러한 상황을 해결하는 방법에는 두 가지가 있습니다. 자체 관리형을 실행하는 경우 동시 고루틴을 늘릴 수 있지만 KCM에서 더 많은 데이터를 처리하면 etcd에 영향을 미칠 수 있습니다. 다른 옵션은 배포.spec.revisionHistoryLimit에서를 사용하여 복제본 객체 수를 줄여 롤백할 수 있는 복제본 객체 수를 줄여이 컨트롤러에 대한 부담을 줄이는 것입니다.

```
spec:
  revisionHistoryLimit: 2
```

다른 Kubernetes 기능은 높은 이탈률 시스템의 압력을 줄이기 위해 조정하거나 끌 수 있습니다. 예를 들어 포드의 애플리케이션이 k8s API와 직접 통신할 필요가 없는 경우 프로젝션된 보안 암호를 해당 포드로 끄면 ServiceaccountTokenSyncs의 부하가 줄어듭니다. 이는 가능한 경우 이러한 문제를 해결하는 더 바람직한 방법입니다.

```
kind: Pod
spec:
  automountServiceAccountToken: false
```

지표에 액세스할 수 없는 시스템에서는 로그를 다시 살펴보고 경합을 감지할 수 있습니다. 컨트롤러당 또는 집계 수준에서 처리 중인 요청 수를 확인하려면 다음 CloudWatch Logs Insights 쿼리를 사용합니다.

KCM에서 처리한 총 볼륨

```
# Query to count API qps coming from kube-controller-manager, split by controller type.
# If you're seeing values close to 20/sec for any particular controller, it's most
 likely seeing client-side API throttling.
fields @timestamp, @logStream, @message
| filter @logStream like /kube-apiserver-audit/
```

```
| filter userAgent like /kube-controller-manager/
# Exclude lease-related calls (not counted under kcm qps)
| filter requestURI not like "apis/coordination.k8s.io/v1/namespaces/kube-system/
leases/kube-controller-manager"
# Exclude API discovery calls (not counted under kcm qps)
| filter requestURI not like "?timeout=32s"
# Exclude watch calls (not counted under kcm qps)
| filter verb != "watch"
# If you want to get counts of API calls coming from a specific controller, uncomment
 the appropriate line below:
# | filter user.username like "system:serviceaccount:kube-system:job-controller"
# | filter user.username like "system:serviceaccount:kube-system:cronjob-controller"
# | filter user.username like "system:serviceaccount:kube-system:deployment-controller"
# | filter user.username like "system:serviceaccount:kube-system:replicaset-controller"
# | filter user.username like "system:serviceaccount:kube-system:horizontal-pod-
autoscaler"
# | filter user.username like "system:serviceaccount:kube-system:persistent-volume-
binder"
# | filter user.username like "system:serviceaccount:kube-system:endpointslice-
controller"
# | filter user.username like "system:serviceaccount:kube-system:endpoint-controller"
# | filter user.username like "system:serviceaccount:kube-system:generic-garbage-
controller"
| stats count(*) as count by user.username
| sort count desc
```

여기서 중요한 점은 확장성 문제를 살펴볼 때 자세한 문제 해결 단계로 이동하기 전에 경로의 모든 단계(API, 스케줄러, KCM 등)를 살펴보는 것입니다. 프로덕션 환경에서 시스템이 가장 뛰어난 성능을 발휘할 수 있도록 Kubernetes의 여러 부분을 조정하는 경우가 많습니다. 병목이 훨씬 더 큰 증상(예: 노드 제한 시간)일 뿐인 문제를 실수로 쉽게 해결할 수 있습니다.

ETCD

etcd는 메모리 매핑된 파일을 사용하여 키 값 페어를 효율적으로 저장합니다. 일반적으로 2, 4, 8GB 제한으로 설정된이 메모리 공간의 크기를 설정하는 보호 메커니즘이 있습니다. 데이터베이스의 객체 수가 적을수록 객체가 업데이트되고 이전 버전을 정리해야 할 때 더 적은 정리 및 정리가 필요합니다. 이전 버전의 객체를 정리하는이 프로세스를 압축이라고 합니다. 여러 압축 작업 후 특정 임계값을 초과하거나 고정된 일정에 따라 발생하는 조각 모음이라고 하는 사용 가능한 공간 공간을 복구하는 후속 프로세스가 있습니다.

Kubernetes의 객체 수를 제한하여 압축 및 조각 제거 프로세스의 영향을 줄이기 위해 수행할 수 있는 몇 가지 사용자 관련 항목이 있습니다. 예를 들어 Helm은 높은를 유지합니

다`revisionHistoryLimit`. 이렇게 하면 시스템의 ReplicaSets와 같은 이전 객체가 롤백을 수행할 수 있습니다. 기록 제한을 2로 설정하면 객체(예: ReplicaSets) 수를 10개에서 2개로 줄여 시스템에 부하를 줄일 수 있습니다.

```
apiVersion: apps/v1
kind: Deployment
spec:
  revisionHistoryLimit: 2
```

모니터링 관점에서 시스템 지연 시간 스파이크가 시간 단위로 구분된 설정 패턴에서 발생하는 경우이 조각 모음 프로세스가 소스인지 확인하는 것이 도움이 될 수 있습니다. CloudWatch Logs를 사용하여 이를 확인할 수 있습니다.

조각 모음의 시작/종료 시간을 보려면 다음 쿼리를 사용합니다.

```
fields @timestamp, @message
| filter @logStream like /etcd-manager/
| filter @message like /defraging|defraged/
| sort @timestamp asc
```

@message	@logStream
Oct 6 19:19:52 ip-172-16-176-9 etcdnanny: {"level":"info","ts":"2022-10-06T19:19:52.826Z","caller":"defrag/defrag.go:312","msg":"defraging","…	etcd-manager-i-0101a20d88e74fa5b
Oct 6 19:20:28 ip-172-16-176-9 etcdnanny: {"level":"info","ts":"2022-10-06T19:20:28.836Z","caller":"defrag/defrag.go:322","msg":"defraged"}	etcd-manager-i-0101a20d88e74fa5b
Oct 6 19:25:57 ip-172-16-101-191 etcdnanny: {"level":"info","ts":"2022-10-06T19:25:57.650Z","caller":"defrag/defrag.go:312","msg":"defraging"…	etcd-manager-i-0e3aa5c2ae801af44
Oct 6 19:26:34 ip-172-16-101-191 etcdnanny: {"level":"info","ts":"2022-10-06T19:26:34.457Z","caller":"defrag/defrag.go:322","msg":"defraged"}	etcd-manager-i-0e3aa5c2ae801af44
Oct 6 19:32:26 ip-172-16-50-136 etcdnanny: {"level":"info","ts":"2022-10-06T19:32:26.592Z","caller":"defrag/defrag.go:312","msg":"defraging",…	etcd-manager-i-025ebb6b463ee0f21
Oct 6 19:33:09 ip-172-16-50-136 etcdnanny: {"level":"info","ts":"2022-10-06T19:33:09.575Z","caller":"defrag/defrag.go:322","msg":"defraged"}	etcd-manager-i-025ebb6b463ee0f21

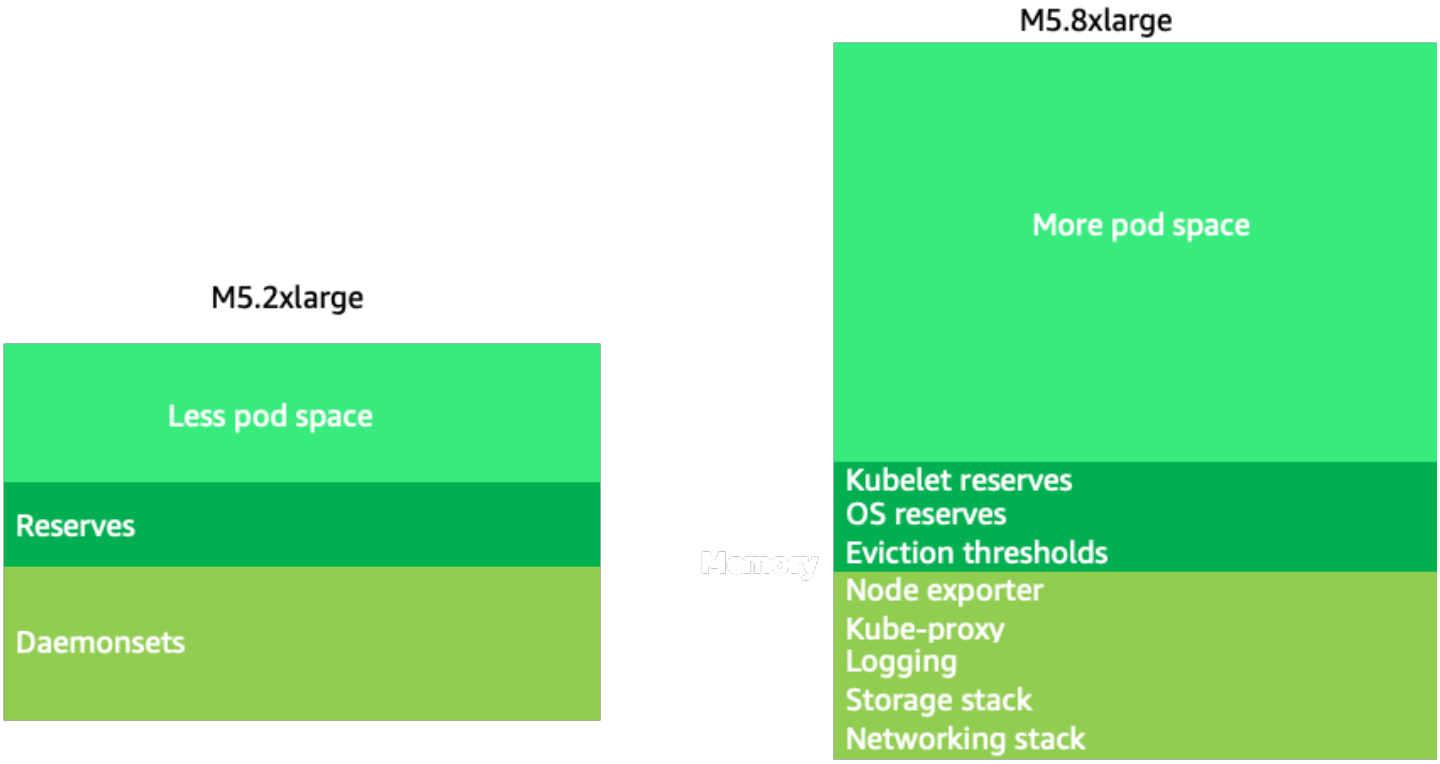
노드 및 워크로드 효율성

워크로드 및 노드를 효율적으로 사용하면 성능 및 규모를 높이는 동시에 복잡성/비용을 줄일 수 있습니다. 이러한 효율성을 계획할 때 고려해야 할 요소는 여러 가지가 있으며, 각 기능에 대한 모범 사례 설정 하나와 비교하여 절충 측면에서 가장 쉽게 생각할 수 있습니다. 다음 단원에서 이러한 장단점을 자세히 살펴보겠습니다.

노드 선택

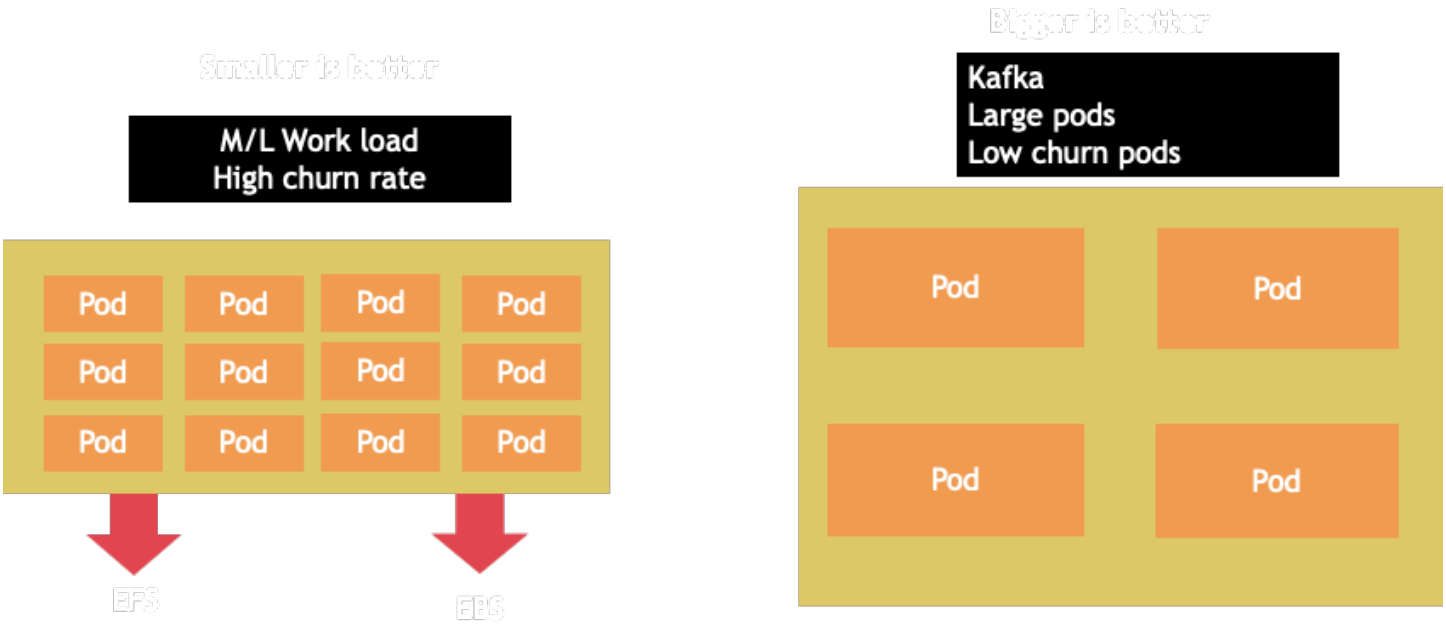
약간 더 큰(4~12xlarge) 노드 크기를 사용하면 DaemonSets 및 시스템 구성 요소에 대한 예약과 같은 "오버헤드"에 사용되는 노드의 백분율이 줄어들기 때문에 포드를 실행하는 데 사용할 수 있는 공간이 늘어납니다. 아래 다이어그램에서 2xlarge의 사용 가능한 공간과 중간 수의 DaemonSets.

> **Note**
>
> k8s는 일반 규칙으로 수평적으로 확장되므로 대부분의 애플리케이션에서 NUMA 크기 노드의 성능 영향을 받아들일 필요가 없으므로 해당 노드 크기보다 작은 범위의 권장 사항을 따르는 것이 좋습니다.



노드 크기가 크면 노드당 사용 가능한 공간의 비율이 높아질 수 있습니다. 그러나 노드를 너무 많은 포드로 패킹하여 오류를 일으키거나 노드를 포화시켜이 모델을 극단적으로 만들 수 있습니다. 노드 포화를 모니터링하는 것은 더 큰 노드 크기를 성공적으로 사용하는 데 중요합니다.

노드 선택이 one-size-fits-all 제안인 경우는 거의 없습니다. 매우 다른 이탈률로 워크로드를 서로 다른 노드 그룹으로 분할하는 것이 가장 좋은 경우가 많습니다. 이탈률이 높은 소규모 배치 워크로드는 4xlarge 인스턴스 패밀리에서 가장 잘 제공되는 반면, 8 vCPU를 사용하고 이탈률이 낮은 Kafka와 같은 대규모 애플리케이션은 12xlarge 패밀리에서 더 잘 제공됩니다.



> **Note**
>
> 매우 큰 노드 크기에서 고려해야 할 또 다른 요소는 CGROUPS가 컨테이너화된 애플리케이션에서 vCPU의 총 수를 숨기지 않기 때문입니다. 동적 런타임은 종종 의도하지 않은 수의 OS 스레드를 생성하여 문제를 해결하기 어려운 지연 시간을 생성할 수 있습니다. 이러한 애플리케이션의 경우 CPU 고정이 권장됩니다. 주제에 대한 자세한 내용은 다음 비디오 https://www.youtube.com/watch?v=NqtfDy_KAqg 참조하십시오.

노드 빈 패키징

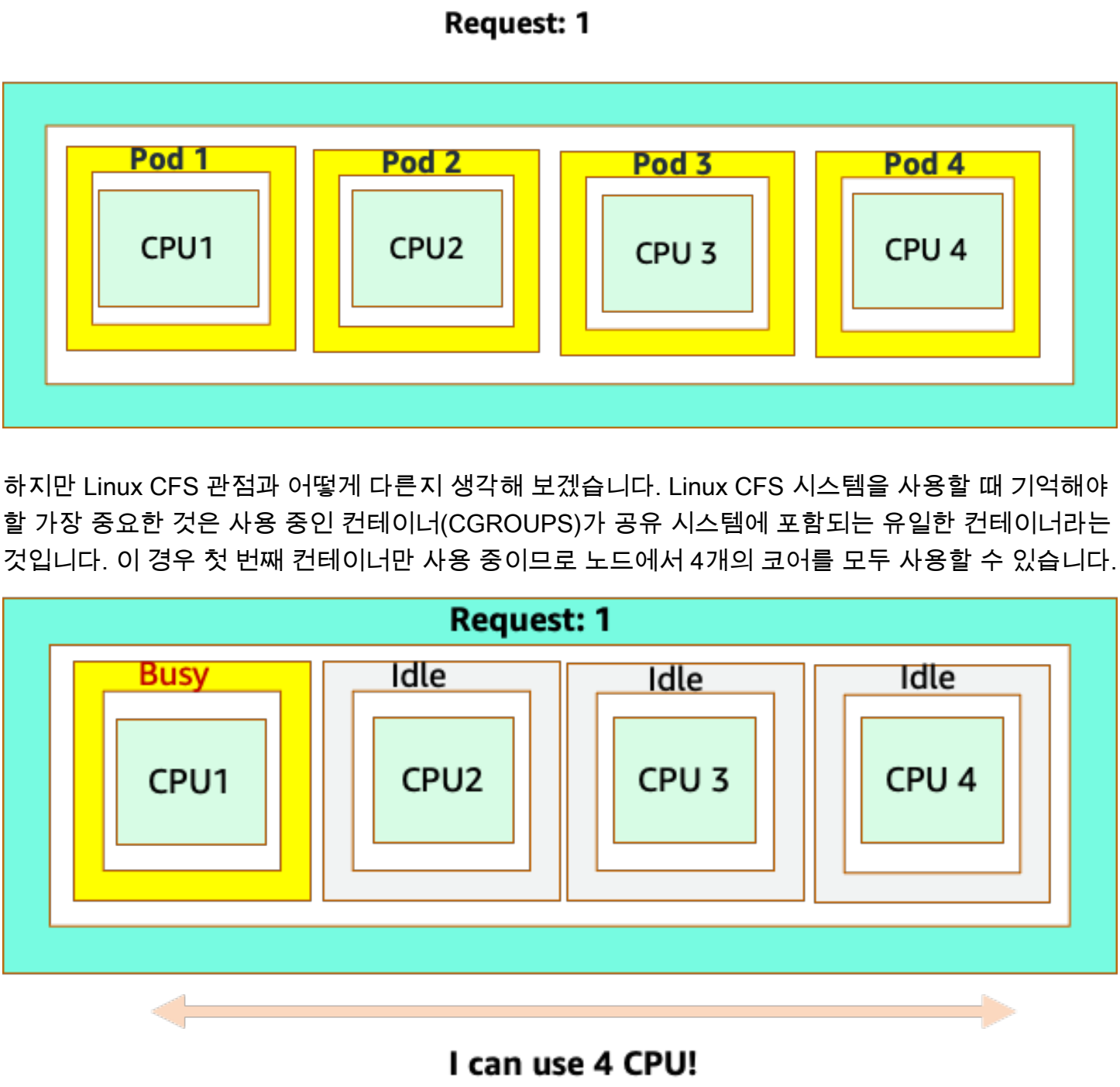
Kubernetes와 Linux 규칙 비교

Kubernetes에서 워크로드를 처리할 때 주의해야 할 두 가지 규칙 세트가 있습니다. 요청 값을 사용하여 노드에서 포드를 예약한 다음 포드가 예약된 후 발생하는 Kubernetes 스케줄러의 규칙으로, Kubernetes가 아닌 Linux의 영역입니다.

Kubernetes 스케줄러가 완료되면 새로운 규칙 세트인 Linux 완전 공정 스케줄러(CFS)가 인수됩니다. 핵심은 Linux CFS에 코어 개념이 없다는 것입니다. 코어를 고려하면 워크로드를 규모에 맞게 최적화하는 데 큰 문제가 발생할 수 있는 이유에 대해 살펴보겠습니다.

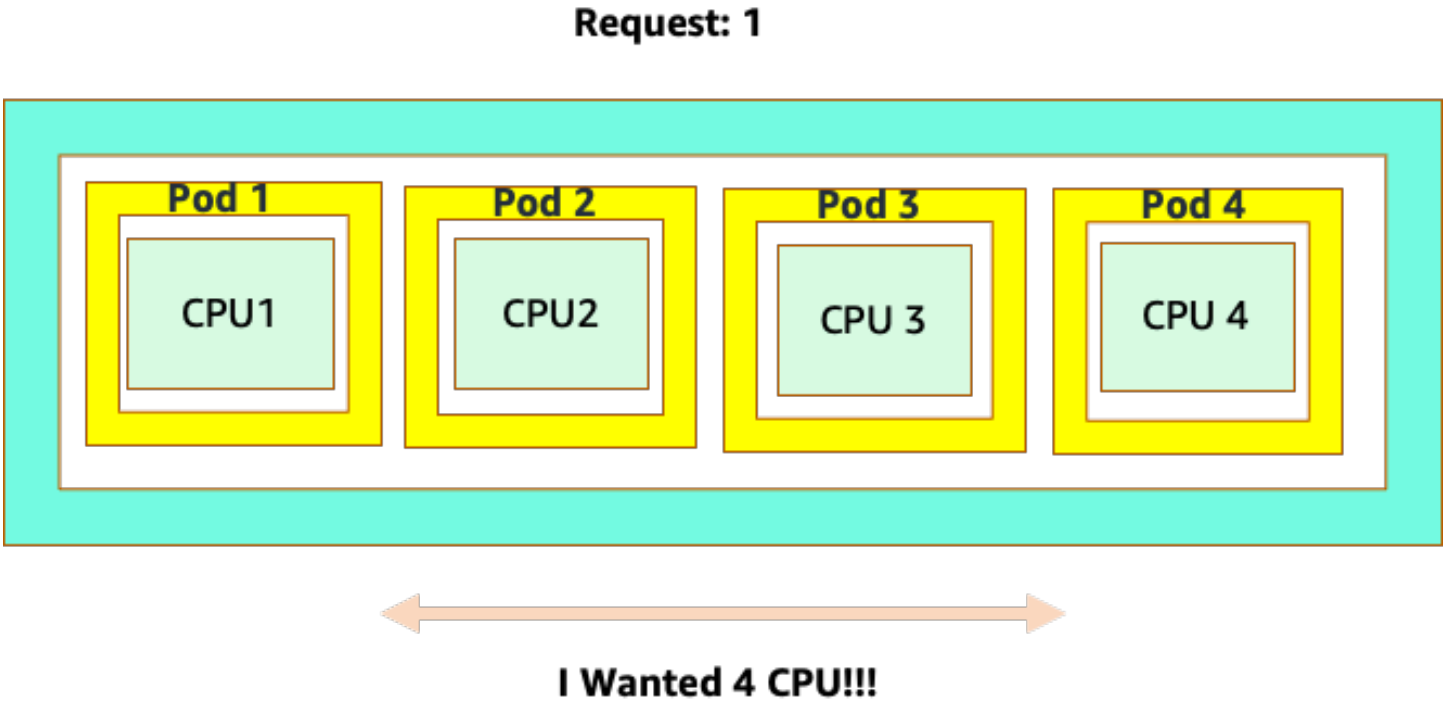
코어에서 생각하기

혼동은 Kubernetes 스케줄러에 코어 개념이 있기 때문에 시작됩니다. Kubernetes 스케줄러 관점에서 각각 하나의 코어 세트 요청이 있는 4개의 NGINX 포드가 있는 노드를 살펴보면 노드는 다음과 같습니다.



하지만 Linux CFS 관점과 어떻게 다른지 생각해 보겠습니다. Linux CFS 시스템을 사용할 때 기억해야 할 가장 중요한 것은 사용 중인 컨테이너(CGROUPS)가 공유 시스템에 포함되는 유일한 컨테이너라는 것입니다. 이 경우 첫 번째 컨테이너만 사용 중이므로 노드에서 4개의 코어를 모두 사용할 수 있습니다.

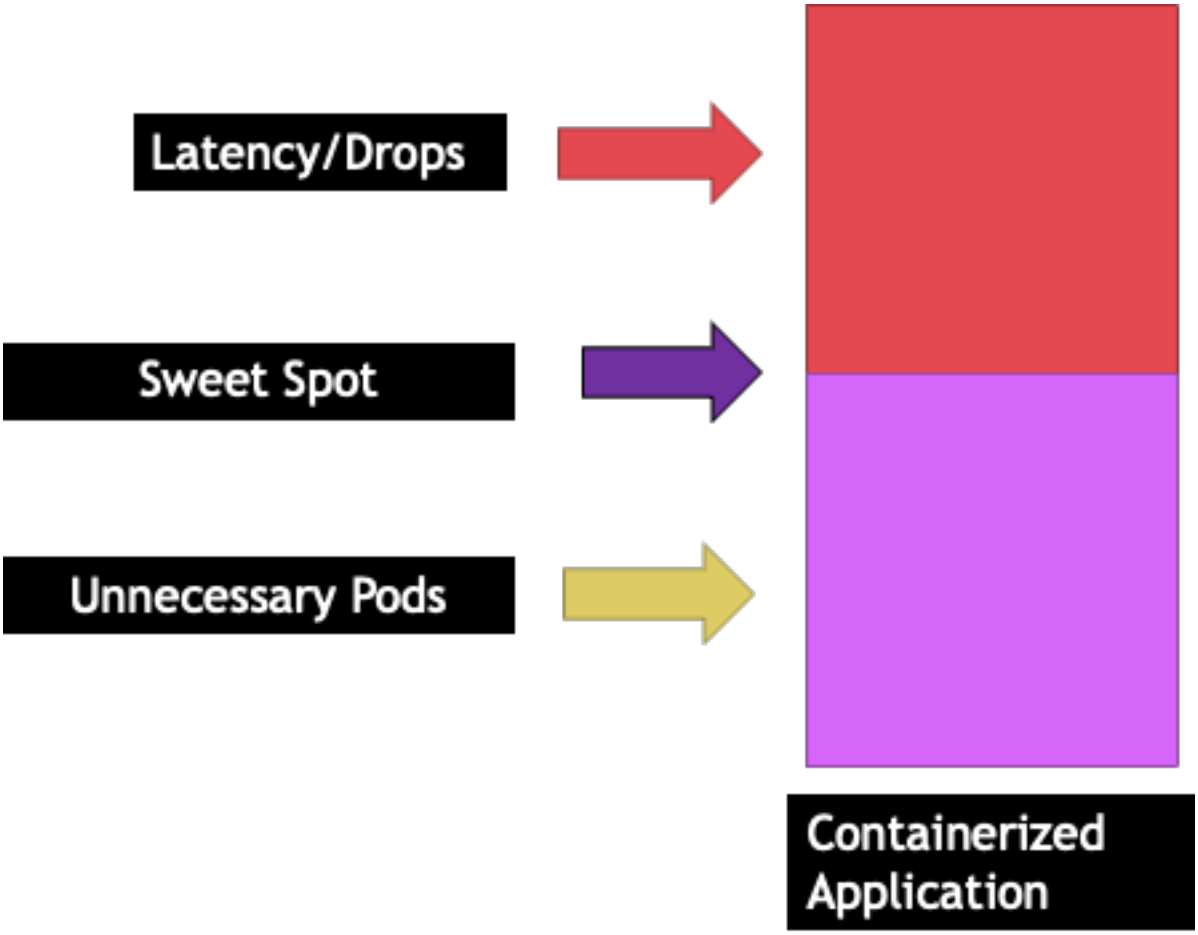
이것이 중요한 이유는 무엇입니까? NGINX 애플리케이션이 해당 노드에서 유일하게 사용 중인 컨테이너였던 개발 클러스터에서 성능 테스트를 실행했다고 가정해 보겠습니다. 앱을 프로덕션으로 이동하면 NGINX 애플리케이션에 4개의 vCPU 리소스가 필요하지만 노드의 다른 모든 포드가 사용 중이므로 앱 성능이 제한됩니다.



이 상황으로 인해 애플리케이션이 "`sweet spot"으로 확장되는 것을 허용하지 않았기 때문에 불필요한 컨테이너를 더 추가할 수 있습니다. 의이 중요한 개념"sweet spot"을 좀 더 자세히 살펴보겠습니다.

애플리케이션 오른쪽 크기 조정

각 애플리케이션에는 더 이상 트래픽을 가져올 수 없는 특정 지점이 있습니다. 이 지점을 초과하면 처리 시간이 늘어나고이 지점을 훨씬 초과하여 푸시할 때 트래픽이 떨어질 수도 있습니다. 이를 애플리케이션의 포화 지점이라고 합니다. 크기 조정 문제를 방지하려면 포화 지점에 도달하기 전에 애플리케이션의 크기 조정을 시도해야 합니다. 이 점을 스위트 스팟이라고 부르겠습니다.

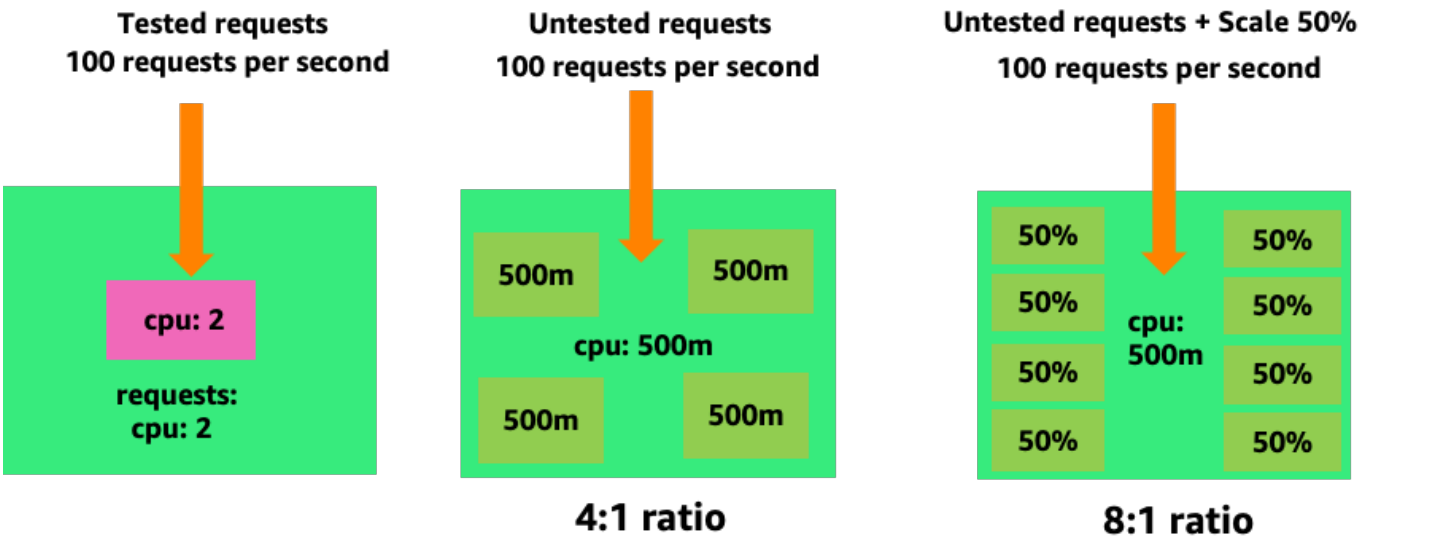


각 애플리케이션을 테스트하여 스위트 스팟을 이해해야 합니다. 애플리케이션마다 다르므로 여기에 범용 지침은 없습니다. 이 테스트에서는 애플리케이션 포화 지점을 보여주는 최상의 지표를 이해하려고 합니다. 사용률 지표를 사용하여 애플리케이션이 포화 상태임을 나타내는 경우가 많지만, 이로 인해 규모 조정 문제가 빠르게 발생할 수 있습니다(이 주제는 이후 단원에서 자세히 살펴보겠습니다). "`sweet spot""이 있으면 이를 사용하여 워크로드를 효율적으로 확장할 수 있습니다.

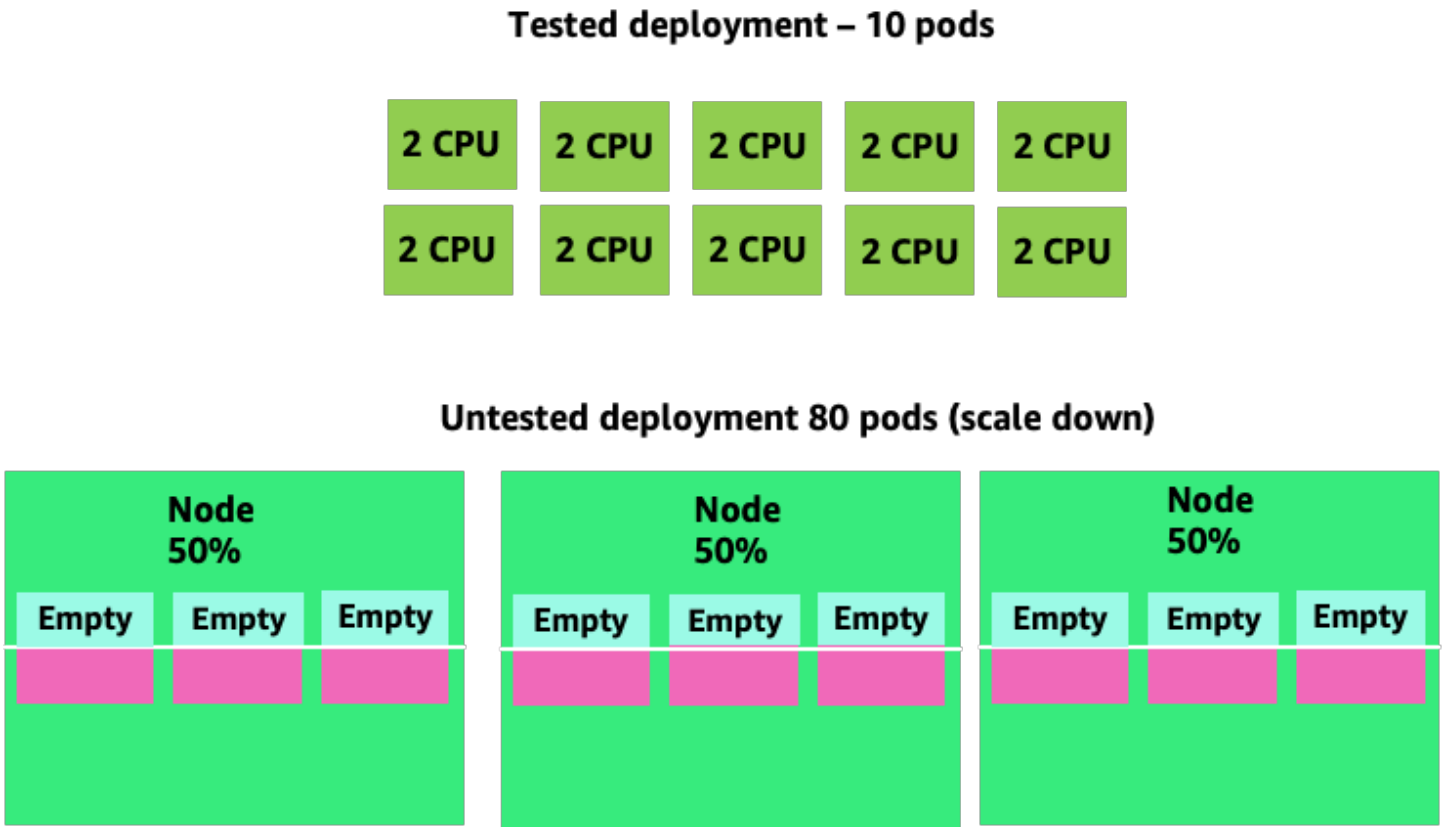
반대로 스위트 스팟 전에 스케일 업하여 불필요한 포드를 만들면 어떻게 될까요? 다음 단원에서 이를 살펴보겠습니다.

포드 스프롤링

불필요한 포드를 생성하는 방법을 빠르게 알아보려면 왼쪽의 첫 번째 예제를 살펴보겠습니다. 이 컨테이너의 올바른 수직 규모는 초당 100개의 요청을 처리할 때 약 2개의 vCPUs 사용률을 차지합니다. 그러나 요청을 코어의 절반으로 설정하여 요청 값을 과소 프로비저닝하려는 경우 이제 실제로 필요한 각 포드에 대해 포드 4개가 필요합니다. 이 문제를 더 강조하면 HPA가 기본값인 50% CPU로 설정된 경우 해당 포드는 절반만큼 비워져 8:1 비율을 생성합니다.

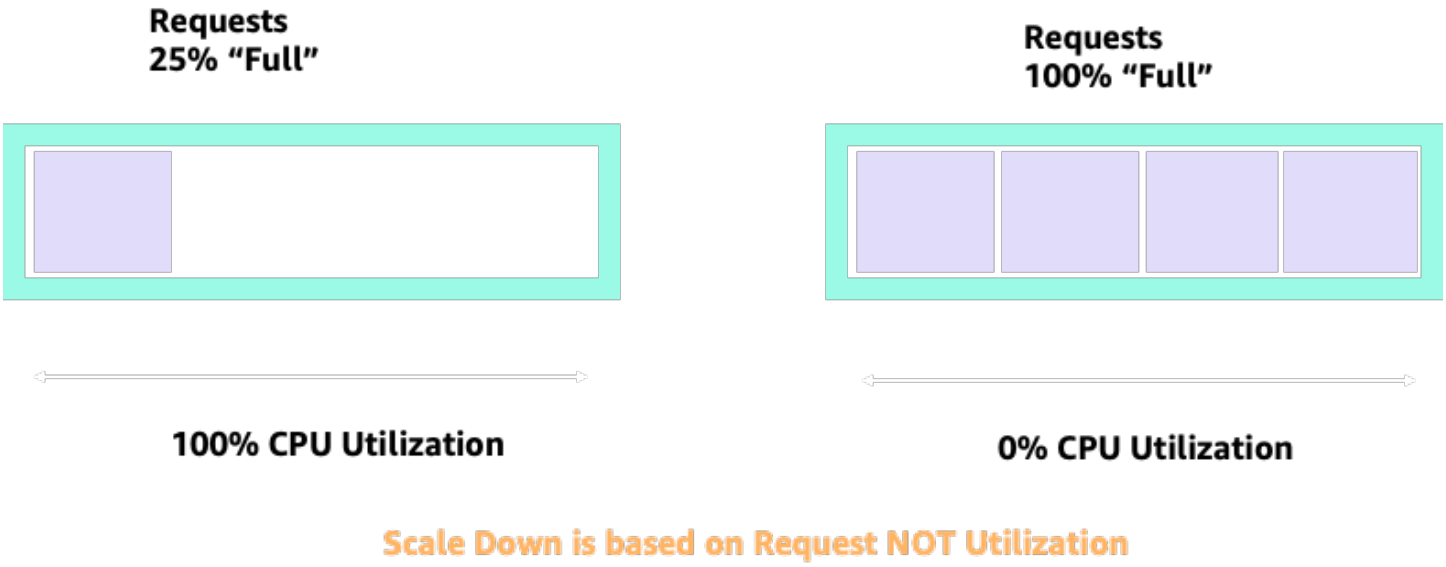


이 문제를 확장하면이 문제를 어떻게 해결할 수 있는지 빠르게 확인할 수 있습니다. 스위트 스팟이 잘못 설정된 10개의 포드를 배포하면 80개의 포드와 이를 실행하는 데 필요한 추가 인프라로 빠르게 스파이럴될 수 있습니다.



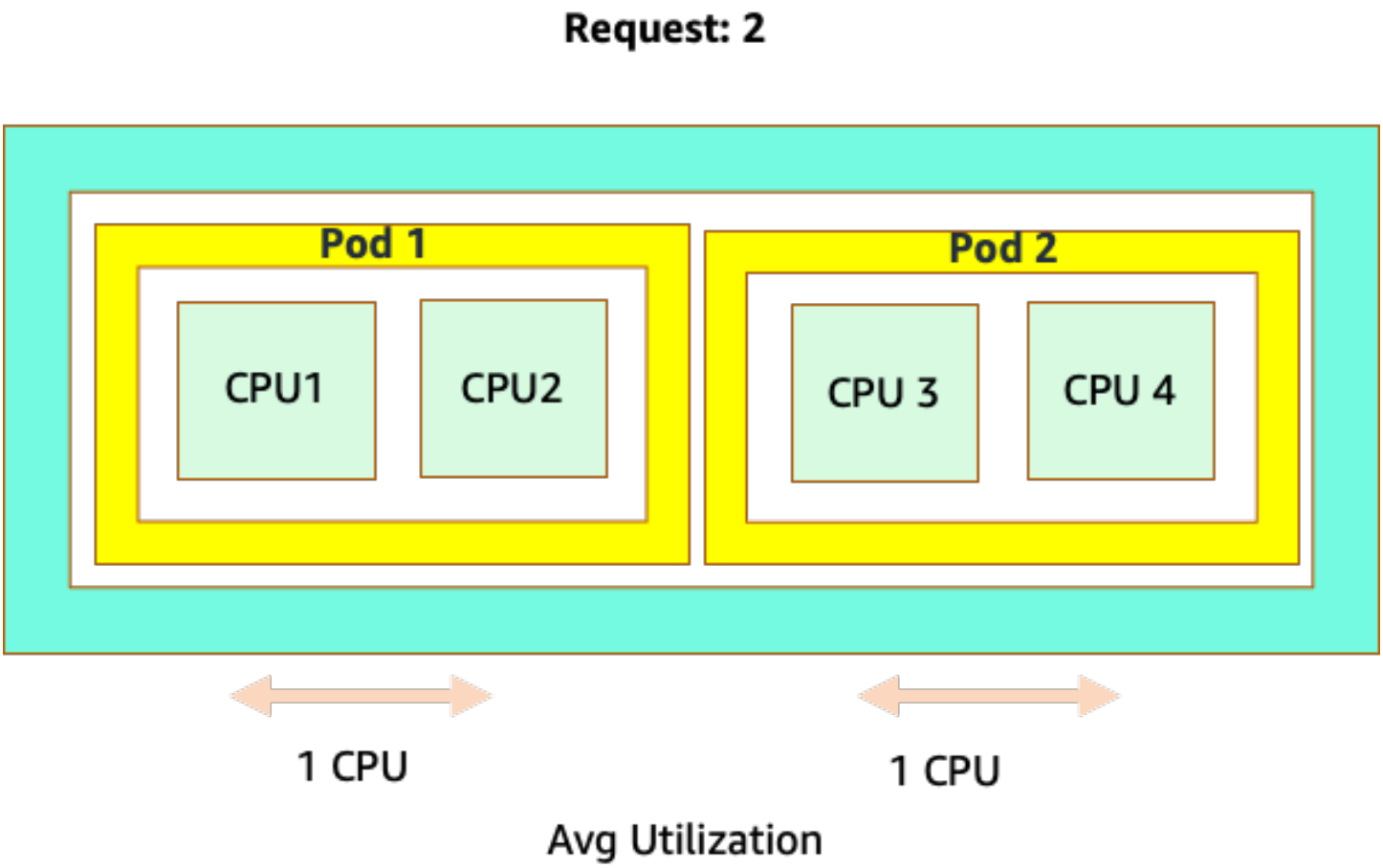
애플리케이션이 스위트 스팟에서 작동하도록 허용하지 않는 것이 미치는 영향을 이해했으므로 이제 노드 수준으로 돌아가 Kubernetes 스케줄러와 Linux CFS 간의 이러한 차이가 왜 중요한지 질문해 보겠습니다.

HPA로 스케일 업 및 스케일 다운할 때 더 많은 포드를 할당할 공간이 많은 시나리오가 있을 수 있습니다. 왼쪽에 표시된 노드의 CPU 사용률이 이미 100%이기 때문에 이는 잘못된 결정입니다. 비현실적이지만 이론적으로 가능한 시나리오에서는 노드가 완전히 채워졌지만 CPU 사용률이 0인 다른 극단적 상황이 발생할 수 있습니다.

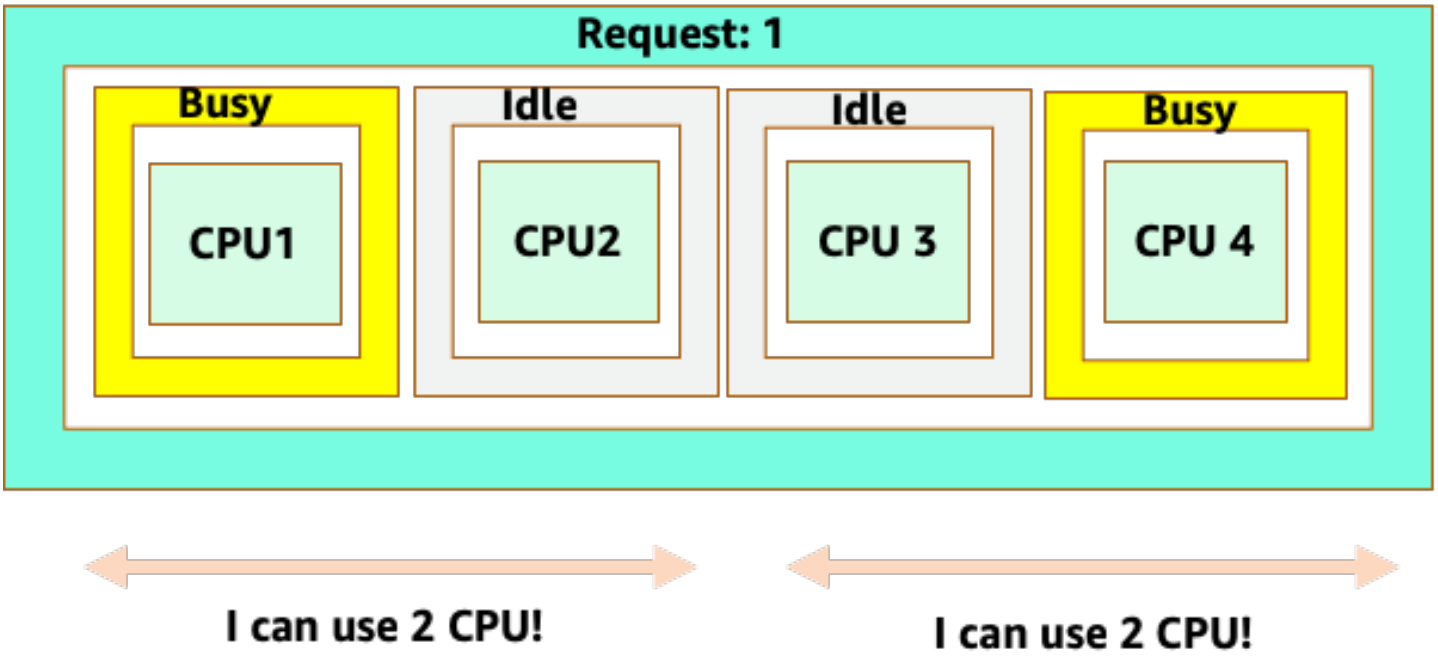


요청 설정

해당 애플리케이션의 "스윗 스팟" 값으로 요청을 설정하려고 하지만 아래 다이어그램과 같이 비효율성이 발생합니다. 여기서는 요청 값을 vCPU 2로 설정했지만 이러한 포드의 평균 사용률은 대부분 CPU 1개만 실행합니다. 이 설정을 사용하면 CPU 주기의 50%를 낭비할 수 있으며, 이는 허용되지 않습니다.



이렇게 하면 문제에 대한 복잡한 답변을 얻을 수 있습니다. 컨테이너 사용률은 vacuum에서 고려할 수 없으며 노드에서 실행되는 다른 애플리케이션을 고려해야 합니다. 다음 예제에서 급증하는 컨테이너는 메모리 제약이 있을 수 있는 낮은 CPU 사용률 컨테이너 2개와 혼합됩니다. 이렇게 하면 노드에 세금을 부과하지 않고 컨테이너가 스위트 스팟에 도달할 수 있습니다.



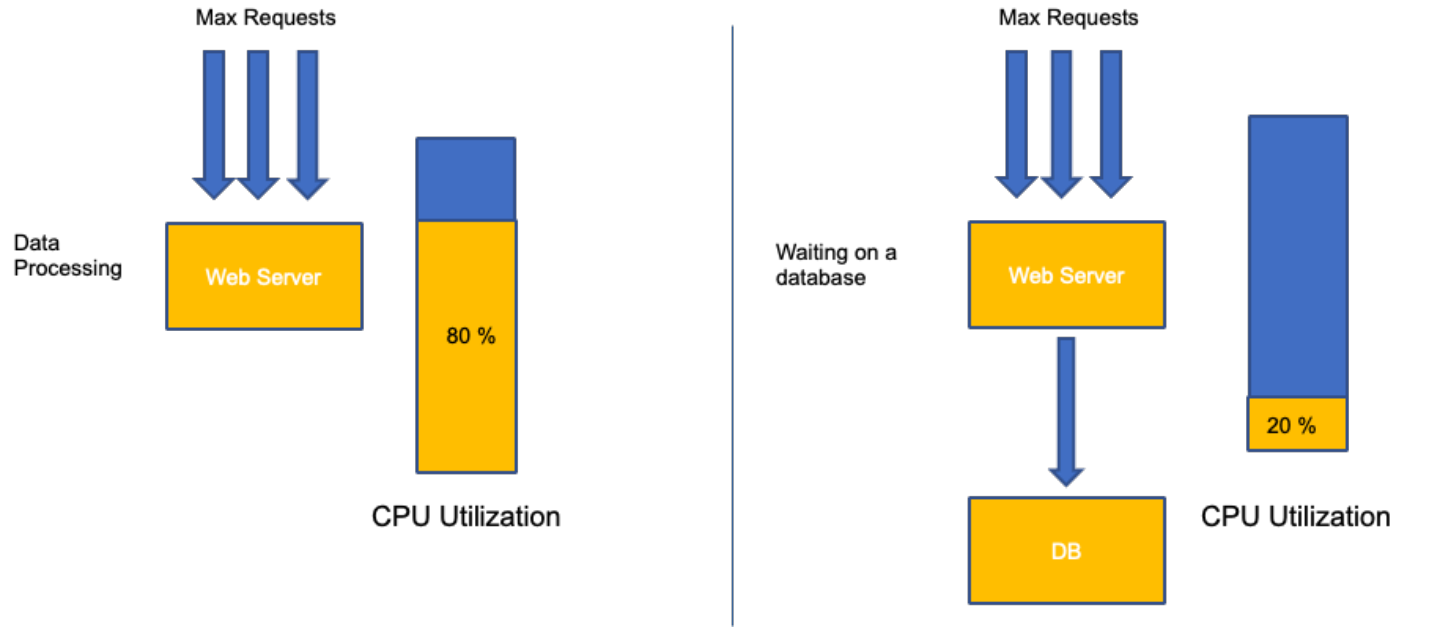
이 모든 것을 해결할 중요한 개념은 코어의 Kubernetes 스케줄러 개념을 사용하여 Linux 컨테이너 성능을 이해하면 관련이 없으므로 의사 결정이 잘못될 수 있다는 것입니다.

> **Note**
>
> Linux CFS에는 강력한 점이 있습니다. 이는 I/O 기반 워크로드의 경우 특히 그렇습니다. 그러나 애플리케이션이 사이드카 없이 전체 코어를 사용하고 I/O 요구 사항이 없는 경우 CPU 고정은이 프로세스에서 상당한 복잡성을 제거할 수 있으며 이러한 주의 사항에 권장됩니다.

사용률과 포화도 비교

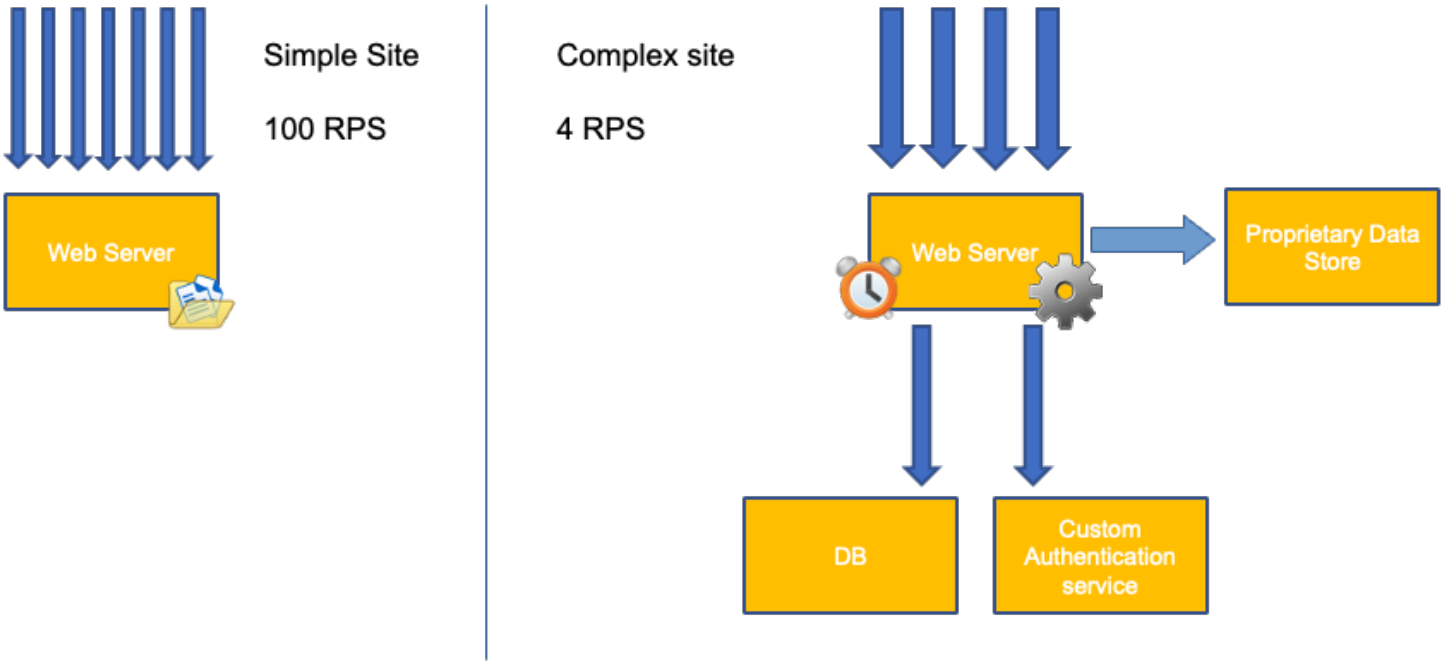
애플리케이션 조정의 일반적인 실수는 조정 지표에 CPU 사용률만 사용하는 것입니다. 복잡한 애플리케이션에서는 애플리케이션이 실제로 요청으로 포화되어 있음을 나타내는 지표가 거의 항상 좋지 않습니다. 왼쪽의 예에서는 모든 요청이 실제로 웹 서버에 도달하는 것을 볼 수 있으므로 CPU 사용률이 포화 상태로 잘 추적되고 있습니다.

실제 애플리케이션에서는 이러한 요청 중 일부가 데이터베이스 계층 또는 인증 계층 등에 의해 서비스될 가능성이 높습니다. 이처럼 일반적인 경우 다른 엔터티에서 요청을 처리하므로 CPU가 포화 상태로 추적되지 않습니다. 이 경우 CPU는 포화 지표가 매우 좋지 않습니다.



애플리케이션 성능에 잘못된 지표를 사용하는 것이 Kubernetes에서 불필요하고 예측할 수 없는 규모 조정의 가장 큰 이유입니다. 사용 중인 애플리케이션 유형에 맞는 올바른 포화 지표를 선택할 때는 각별히 주의해야 합니다. 제공할 수 있는 모든 권장 사항에 맞는 크기가 한 개도 없다는 점에 유의해야 합니다. 사용된 언어와 해당 애플리케이션 유형에 따라 포화도에 대한 다양한 지표 세트가 있습니다.

이 문제는 CPU 사용률에만 있다고 생각할 수 있지만 초당 요청과 같은 다른 일반적인 지표도 위에서 설명한 것과 정확히 동일한 문제가 될 수 있습니다. 요청이 웹 서버에서 직접 서비스되지 않는 DB 계층, 인증 계층으로 이동할 수도 있으므로 웹 서버 자체의 실제 포화에 대한 지표가 좋지 않습니다.



안타깝게도 올바른 포화 지표를 선택할 때 쉬운 답변은 없습니다. 다음은 고려해야 할 몇 가지 지침입니다.

- 언어 런타임 이해 - 여러 OS 스레드가 있는 언어는 단일 스레드 애플리케이션과 다르게 반응하므로 노드에 다르게 영향을 미칩니다.
- 올바른 수직 규모 이해 - 새 포드를 규모 조정하기 전에 애플리케이션 수직 규모에서 원하는 버퍼의 양
- 애플리케이션의 포화도를 실제로 반영하는 지표 - Kafka 생산자의 포화 지표는 복잡한 웹 애플리케이션과 상당히 다릅니다.
- 노드의 다른 모든 애플리케이션은 서로에게 어떤 영향을 미치는가 - 애플리케이션 성능은 노드의 다른 워크로드에 큰 영향을 미치지 않습니다.

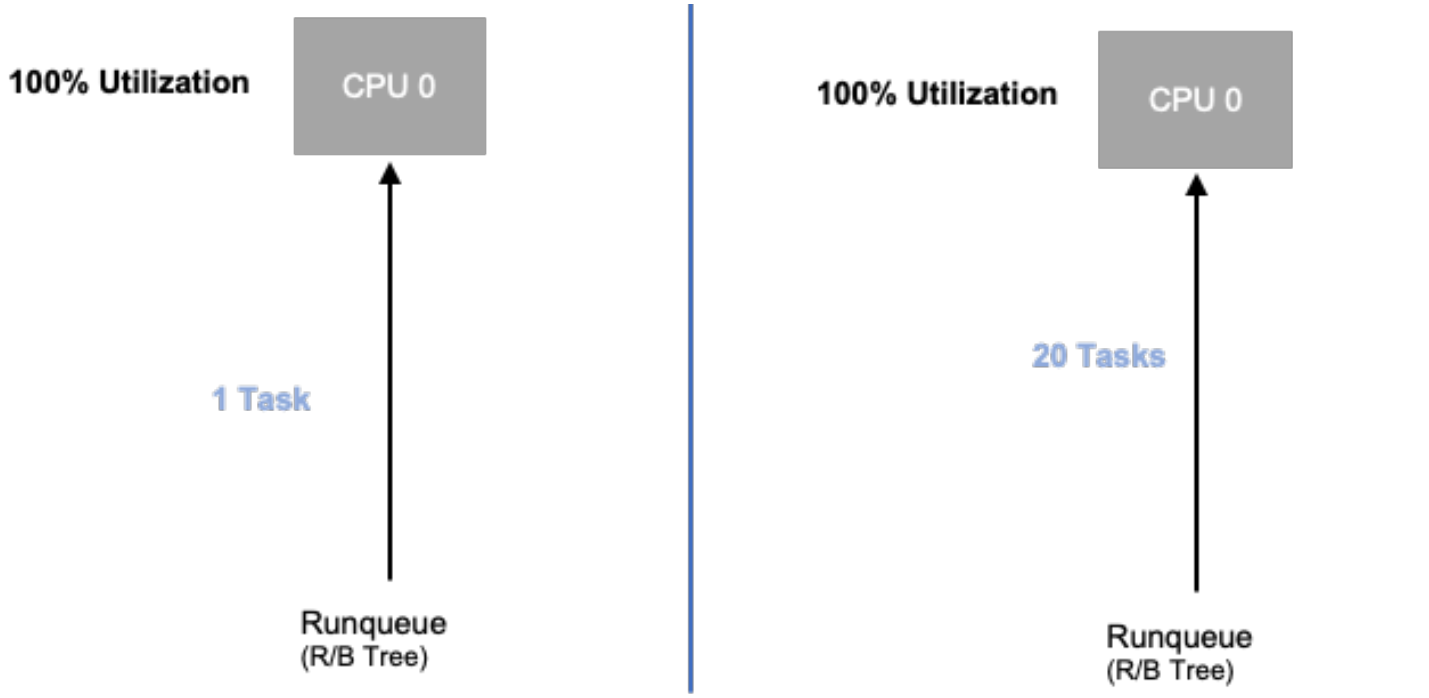
이 섹션을 닫으려면 위의 내용을 지나치게 복잡하고 불필요하게 무시하기 쉽습니다. 문제가 발생하는 경우가 많지만 잘못된 지표를 보고 있기 때문에 문제의 실제 특성을 알지 못합니다. 다음 섹션에서는 이러한 일이 어떻게 발생할 수 있는지 살펴보겠습니다.

노드 포화

이제 애플리케이션 포화를 살펴보았으므로 노드 관점에서이 동일한 개념을 살펴보겠습니다. 사용률과 포화도의 차이를 확인하기 위해 100% 활용되는 두 개의 CPUs를 살펴보겠습니다.

왼쪽의 vCPU는 100% 활용되지만이 vCPU에서 실행되기를 기다리는 다른 작업은 없습니다. 따라서 이론적으로는 이것이 매우 효율적입니다. 한편, 두 번째 예제에서는 vCPU에서 처리되기 위해 대기 중인 단일 스레드 애플리케이션 20개가 있습니다. 이제 20개 애플리케이션 모두 vCPU에서 처리되기를 기다리는 동안 일종의 지연 시간이 발생합니다. 즉, 오른쪽의 vCPU는 포화 상태입니다.

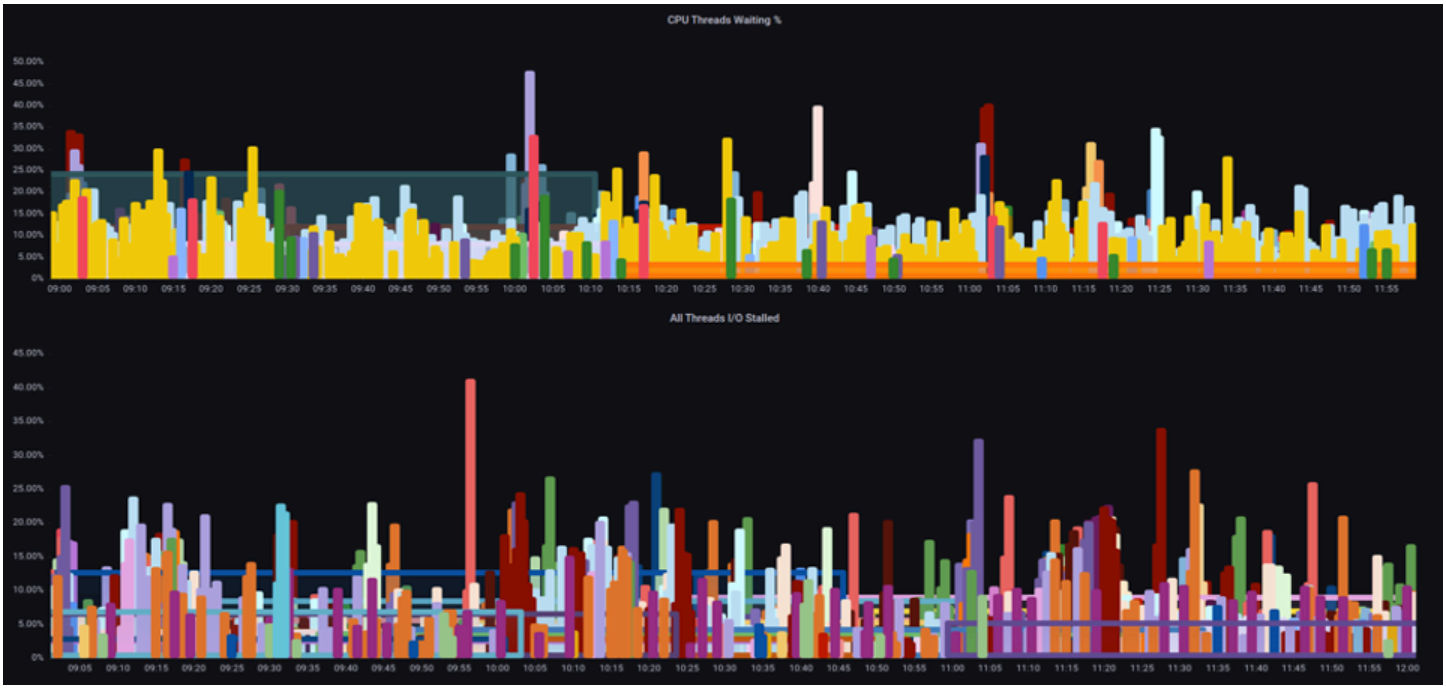
사용률을 살펴보는 경우이 문제가 발생하지 않을 뿐만 아니라이 지연 시간은 네트워킹과 같이 관련이 없는 문제로 인해 잘못된 경로로 이어질 수 있습니다.



노드가 과도하게 포화되었다는 사실을 쉽게 놓칠 수 있으므로 언제든지 노드에서 실행되는 총 포드 수를 늘릴 때 사용률 지표뿐만 아니라 포화 지표를 보는 것이 중요합니다. 이 작업의 경우 아래 차트와 같이 압력 스톨 정보 지표를 사용할 수 있습니다.

PromQL - 중지된 I/O

```
topk(3, ((irate(node_pressure_io_stalled_seconds_total[1m])) * 100))
```



> **Note**
>
> 압력 스톨 지표에 대한 자세한 내용은 https://facebookmicrosites.github.io/psi/docs/overview 참조하세요.*

이러한 지표를 사용하면 스레드가 CPU에서 대기 중인지 또는 상자의 모든 스레드가 메모리 또는 I/O와 같은 리소스에서 대기 중인지 알 수 있습니다. 예를 들어 인스턴스의 모든 스레드가 1분 동안 I/O를 대기한 비율을 확인할 수 있습니다.

```
topk(3, ((irate(node_pressure_io_stalled_seconds_total[1m])) * 100))
```

이 지표를 사용하면 위의 차트에서 상자의 모든 스레드가 높은 워터마크에서 I/O를 기다리는 시간의 45% 동안 중단되었음을 확인할 수 있습니다. 즉, 해당 1분 동안 모든 CPU 주기를 버리고 있었습니다. 이러한 일이 발생하고 있음을 이해하면 상당한 양의 vCPU 시간을 회수하여 규모 조정의 효율성을 높이는 데 도움이 될 수 있습니다.

HPA V2

HPA API의 autoscaling/v2 버전을 사용하는 것이 좋습니다. 이전 버전의 HPA API는 특정 엣지 케이스에서 조정이 중단될 수 있습니다. 또한 각 조정 단계에서 포드가 두 배로만 제한되어 빠르게 확장해야 하는 소규모 배포에 문제가 발생했습니다.

Autoscaling/v2를 사용하면 확장할 여러 기준을 포함할 수 있는 유연성이 향상되고 사용자 지정 및 외부 지표(비 K8s 지표)를 사용할 때 유연성이 크게 향상됩니다.

예를 들어 세 값 중 가장 높은 값을 조정할 수 있습니다(아래 참조). 모든 포드의 평균 사용률이 50%를 초과하는 경우, 사용자 지정 지표가 수신의 초당 패킷 수가 평균 1,000을 초과하거나 수신 객체가 초당 10K 요청을 초과하는 경우 확장됩니다.

> **Note**
>
> 이는 Auto Scaling API의 유연성을 보여주기 위한 것입니다. 프로덕션 환경에서 문제를 해결하기 어려울 수 있는 지나치게 복잡한 규칙을 사용하지 않는 것이 좋습니다.

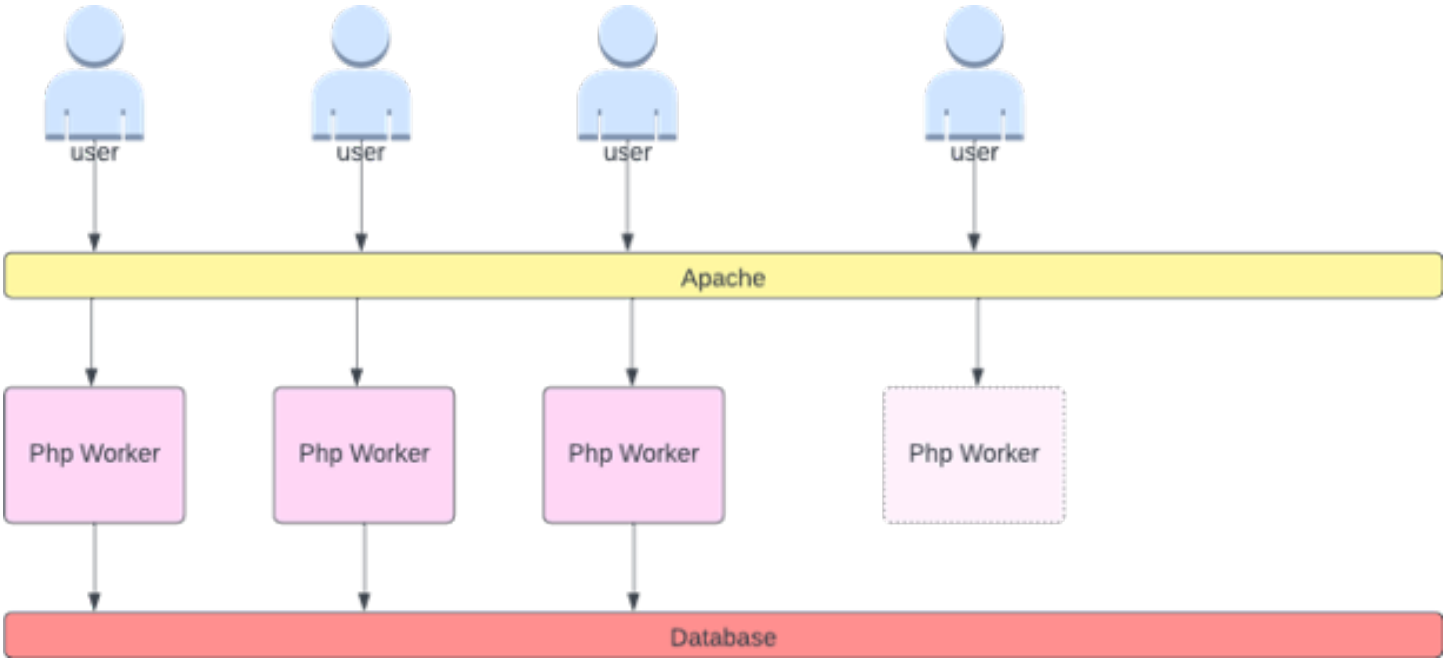
```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
  - type: Pods
    pods:
      metric:
        name: packets-per-second
      target:
        type: AverageValue
        averageValue: 1k
  - type: Object
    object:
      metric:
        name: requests-per-second
```

```
    describedObject:
      apiVersion: networking.k8s.io/v1
      kind: Ingress
      name: main-route
    target:
      type: Value
      value: 10k
```

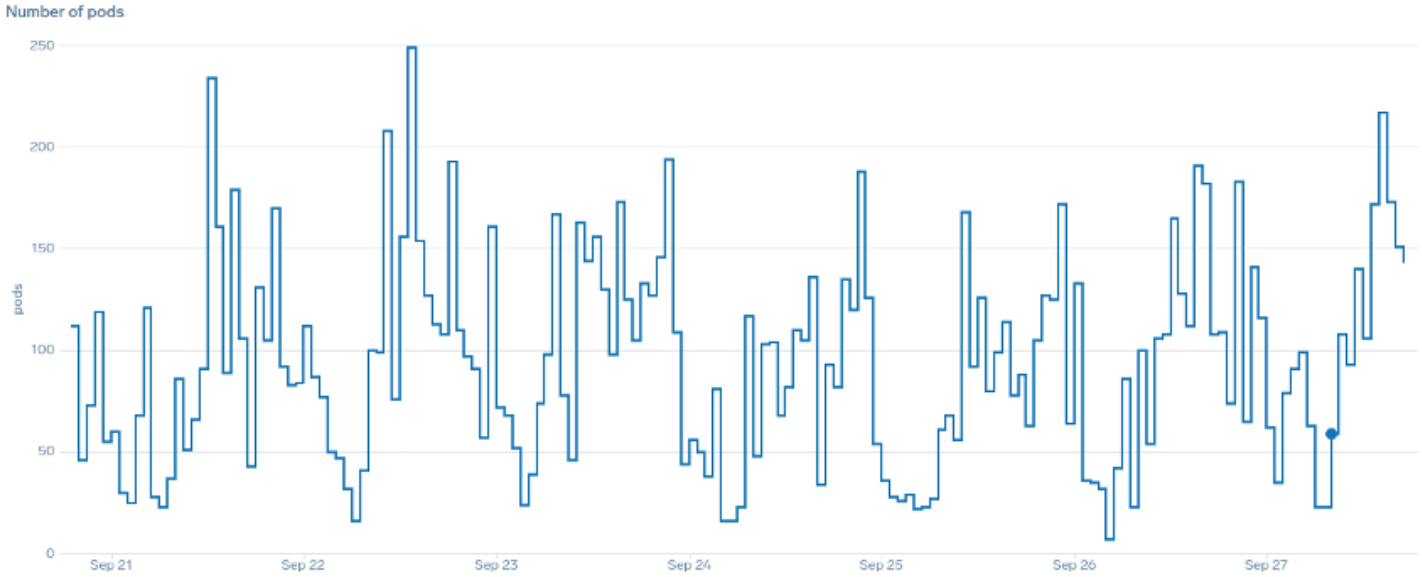
그러나 복잡한 웹 애플리케이션에 이러한 지표를 사용할 경우의 위험을 배웠습니다. 이 경우 애플리케이션의 포화도와 사용률을 정확하게 반영하는 사용자 지정 또는 외부 지표를 사용하면 더 나은 서비스를 제공할 수 있습니다. HPAv2는 모든 지표에 따라 확장할 수 있는 기능을 지원하지만, 여전히 사용할 수 있도록 해당 지표를 찾아 Kubernetes로 내보내야 합니다.

예를 들어 Apache에서 활성 스레드 대기열 수를 볼 수 있습니다. 이렇게 하면 "smoother" 조정 프로필이 생성되는 경우가 많습니다(이 용어는 곧 추가됨). 스레드가 활성 상태인 경우 해당 스레드가 데이터베이스 계층에서 대기 중인지 아니면 로컬에서 요청을 처리하는지는 중요하지 않습니다. 모든 애플리케이션 스레드가 사용 중인 경우 애플리케이션이 포화 상태임을 나타내는 좋은 표시입니다.

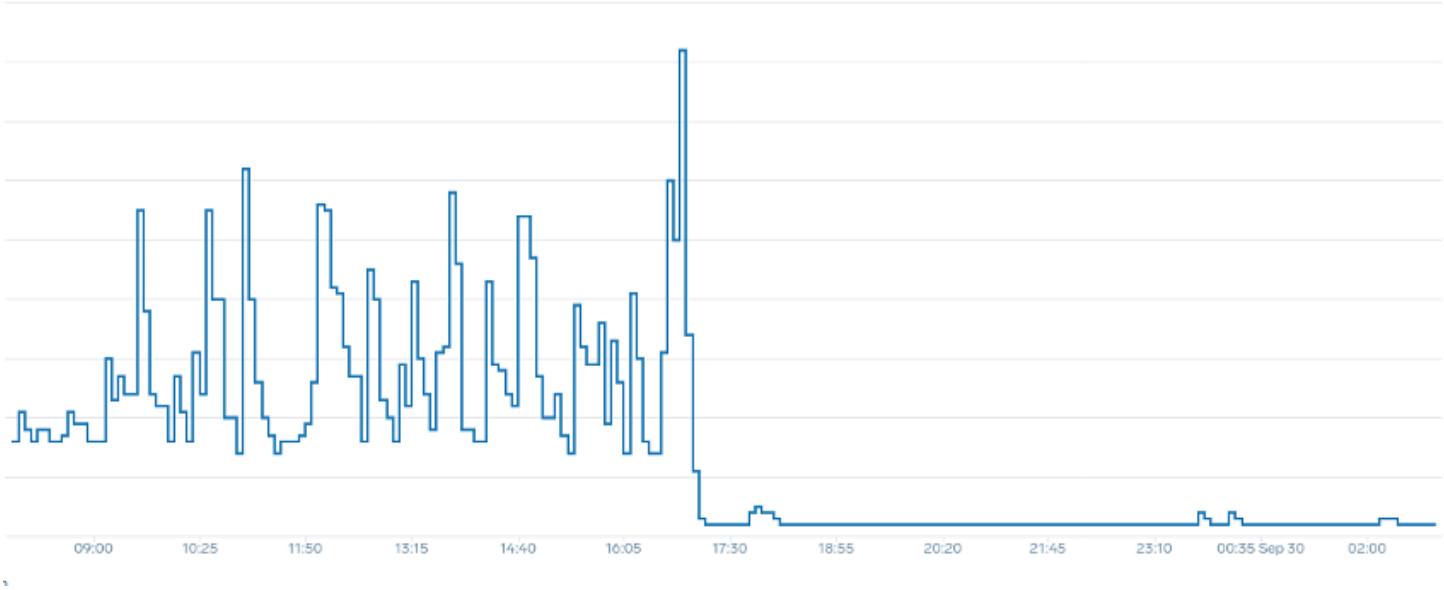
이 스레드 소진을 신호로 사용하여 완전히 사용 가능한 스레드 풀이 있는 새 포드를 생성할 수 있습니다. 또한 트래픽이 많을 때 애플리케이션에서 흡수할 버퍼의 크기를 제어할 수 있습니다. 예를 들어 총 스레드 풀이 10인 경우 사용된 스레드 4개와 사용된 스레드 8개로 조정하면 애플리케이션 조정 시 사용할 수 있는 버퍼에 큰 영향을 미칩니다. 4로 설정하면 과중한 부하에서 빠르게 확장해야 하는 애플리케이션에 적합할 것입니다.이 경우 시간이 지남에 따라 요청 수가 느리게 증가하거나 급격하게 증가하여 확장할 시간이 많으면 리소스에서 8로 설정하면 더 효율적입니다.



규모 조정과 관련하여 "부드러움"이라는 용어는 무엇을 의미합니까? CPU를 지표로 사용하는 아래 차트에 유의하십시오. 이 배포의 포드는 50개의 포드에서 잠시 동안 급증하며, 즉시 다시 스케일 다운하기 위해 포드는 최대 250개까지만 가능합니다. 이는 클러스터 이탈의 주요 원인입니다.



애플리케이션의 올바른 스위트 스팟(차트의 중간 부분)을 반영하는 지표로 변경한 후 어떻게 원활하게 확장할 수 있는지 확인합니다. 이제 확장이 효율적이며, 요청 설정을 조정하여 제공한 헤드룸에 따라 포드를 완전히 확장할 수 있습니다. 이제 더 작은 포드 그룹이 수백 개의 포드가 이전에 하던 작업을 수행하고 있습니다. 실제 데이터는 이것이 Kubernetes 클러스터의 확장성에서 가장 중요한 요소임을 보여줍니다.



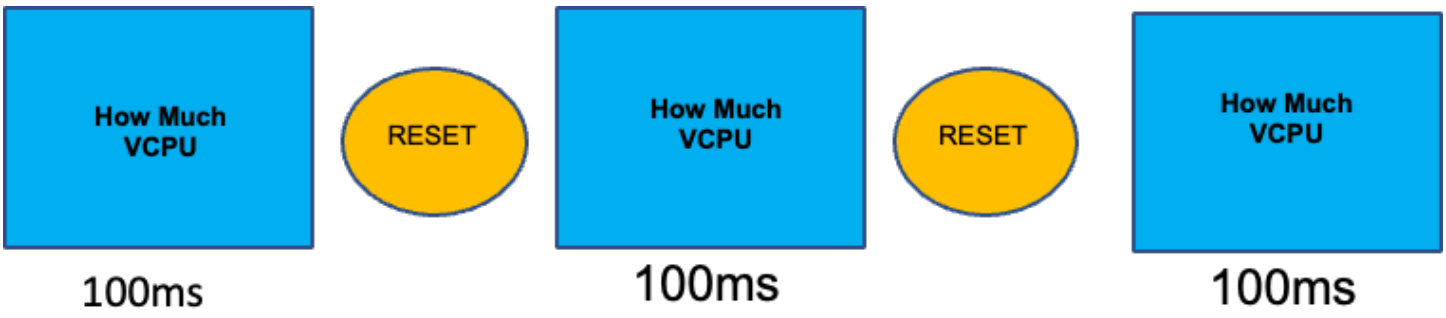
핵심은 CPU 사용률이 애플리케이션과 노드 성능의 한 가지 차원에 불과하다는 것입니다. CPU 사용률을 노드 및 애플리케이션의 유일한 상태 지표로 사용하면 규모 조정, 성능 및 비용에 문제가 발생하며,

이는 모두 밀접하게 연결된 개념입니다. 애플리케이션과 노드의 성능이 높을수록 규모 조정이 덜 필요하므로 비용이 절감됩니다.

특정 애플리케이션의 규모를 조정하기 위해 올바른 포화 지표를 찾고 사용하면 해당 애플리케이션의 실제 병목 현상을 모니터링하고 경보할 수도 있습니다. 이 중요한 단계를 건너뛰면 성능 문제에 대한 보고서를 이해하기가 불가능하지는 않지만 어렵습니다.

CPU 제한 설정

오해의 소지가 있는 주제에 대해이 단원을 마무리하기 위해 CPU 제한에 대해 살펴보겠습니다. 즉, 제한은 100ms마다 재설정되는 카운터가 있는 컨테이너와 연결된 메타데이터입니다. 이를 통해 Linux는 100ms 기간 동안 특정 컨테이너에서 노드 전체에서 사용되는 CPU 리소스 수를 추적할 수 있습니다.



제한 설정 시 발생하는 일반적인 오류는 애플리케이션이 단일 스레드이고 "`assigned"` vCPU에서만 실행된다고 가정하는 것입니다. 위 섹션에서는 CFS가 코어를 할당하지 않으며, 실제로는 큰 스레드 풀을 실행하는 컨테이너가 상자의 사용 가능한 모든 vCPU에 예약된다는 것을 배웠습니다.

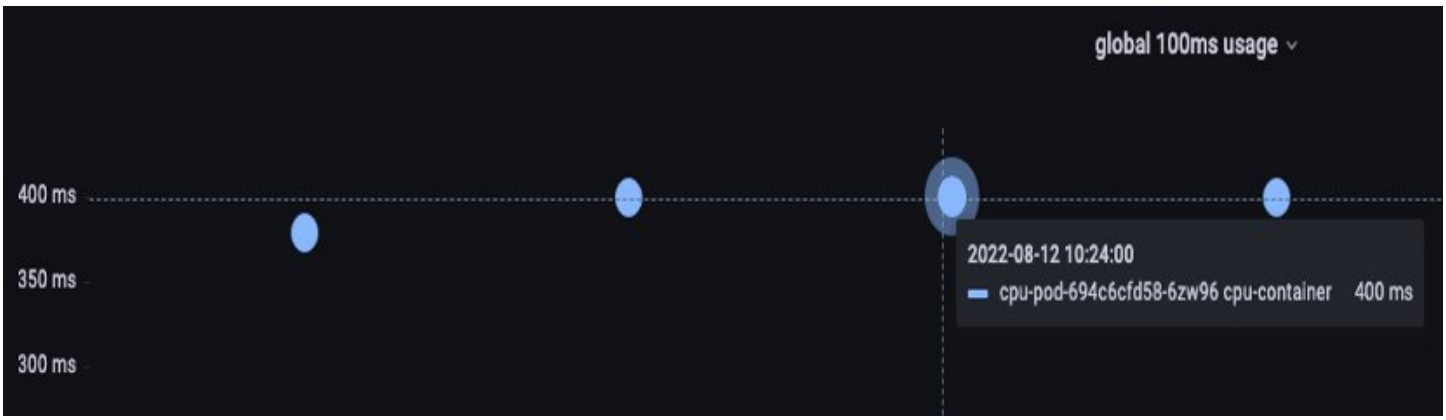
64개의 OS 스레드가 64개의 사용 가능한 코어에서 실행되는 경우(Linux 노드 관점에서) 64개의 모든 코어에서 실행되는 시간이 추가된 후 100ms 기간 동안 사용된 총 CPU 시간을 상당히 크게 계산합니다. 이는 가비지 수집 프로세스 중에만 발생할 수 있으므로 이와 같은 것을 놓치기 쉽습니다. 따라서 제한을 설정하기 전에 시간 경과에 따른 올바른 사용량을 보장하기 위해 지표를 사용해야 합니다.

다행히도 애플리케이션의 모든 스레드에서 사용되는 vCPU의 양을 정확히 확인할 수 있습니다. `container_cpu_usage_seconds_total`이 목적으로 지표를 사용합니다.

제한 로직은 100ms마다 발생하고이 지표는 초당 지표이므로이 100ms 기간과 일치하도록 PromQL을 사용합니다. 이 PromQL 문 작업을 자세히 알아보려면 다음 블로그를 참조하세요.

PromQL 쿼리:

```
topk(3, max by (pod, container)(rate(container_cpu_usage_seconds_total{image!="",
 instance="$instance"}[$__rate_interval]))) / 10
```



올바른 가치가 있다고 생각되면 프로덕션 환경에 제한을 둘 수 있습니다. 그런 다음 예기치 않은 문제로 인해 애플리케이션이 제한되고 있는지 확인해야 합니다. 이를 위해 다음을 살펴볼 수 있습니다. `container_cpu_throttled_seconds_total`

```
topk(3, max by (pod, container)(rate(container_cpu_cfs_throttled_seconds_total{image!
=``""``, instance=``"$instance"``}[$__rate_interval]))) / 10
```



Memory

메모리 할당은 Linux CGroup 동작에 대한 Kubernetes 스케줄링 동작을 혼동하기 쉬운 또 다른 예입니다. 이 주제는 CGroup v2가 Linux에서 메모리를 처리하는 방식이 크게 변경되었고 Kubernetes가 이를 반영하도록 구문을 변경했기 때문에 더 미묘한 차이가 있습니다. 자세한 내용은이 블로그를 참조하세요.

CPU 요청과 달리 메모리 요청은 예약 프로세스가 완료된 후 사용되지 않습니다. 이는 CPU와 동일한 방식으로 CGroup v1의 메모리를 압축할 수 없기 때문입니다. 이렇게 하면 포드를 완전히 종료하여 메모리 누수에 대한 페일 세이프 역할을 하도록 설계된 메모리 제한만 남게 됩니다. 이는 모두 또는 전혀 스타일 제안이 아니지만 이제이 문제를 해결할 수 있는 새로운 방법을 제공했습니다.

먼저 컨테이너에 적절한 양의 메모리를 설정하는 것은 나타나는 것처럼 간단하지 않다는 점을 이해하는 것이 중요합니다. Linux의 파일 시스템은 메모리를 캐시로 사용하여 성능을 개선합니다. 이 캐시는 시간이 지남에 따라 증가하며 캐시에 사용할 수 있는 메모리의 양을 파악하기 어려울 수 있지만 애플리케이션 성능에 큰 영향을 주지 않고 회수할 수 있습니다. 이로 인해 메모리 사용량이 잘못 해석되는 경우가 많습니다.

메모리를 "압축"하는 기능은 CGroup v2의 기본 동인 중 하나였습니다. CGroup V2가 필요한 이유에 대한 자세한 내용은 LISA21에서 Chris Down의 프레젠테이션을 참조하세요.이 프레젠테이션에서는 최소 메모리를 올바르게 설정할 수 없는 것이 CGroup v2 및 압력 스톨 지표를 생성하도록 유도한 이유 중 하나인 이유를 다룹니다.

다행히도 Kubernetes의 개념은 `memory.high` 아래에 `memory.min` 및 입니다`requests.memory`. 이렇게 하면 다른 컨테이너에서 사용할 수 있도록이 캐시된 메모리를 적극적으로 릴리스할 수 있습니다. 컨테이너가 메모리 상한에 도달하면 커널은에 설정된 값까지 해당 컨테이너의 메모리를 적극적으로 회수할 수 있습니다`memory.min`. 따라서 노드에 메모리 압력이 가해질 때 더 많은 유연성을 제공합니다.

핵심 질문은 입니다. 설정할 값은 무엇입니까`memory.min`? 여기에서 메모리 압력 스톨 지표가 작동합니다. 이러한 지표를 사용하여 컨테이너 수준에서 메모리 "스래싱"을 감지할 수 있습니다. 그런 다음 fbtax와 같은 컨트롤러를 사용하여이 메모리 스래싱을 `memory.min` 찾아에 대한 올바른 값을 감지하고 `memory.min` 값을이 설정으로 동적으로 설정할 수 있습니다.

요약

섹션을 요약하기 위해 다음 개념을 쉽게 혼동할 수 있습니다.

- 사용률 및 포화도
- Kubernetes 스케줄러 로직을 사용한 Linux 성능 규칙

이러한 개념을 분리하려면 각별히 주의해야 합니다. 성능과 규모는 심층적으로 연결됩니다. 불필요한 조정으로 인해 성능 문제가 발생하고, 이로 인해 조정 문제가 발생합니다.

Kubernetes 업스트림 SLOs

Amazon EKS는 업스트림 Kubernetes 릴리스와 동일한 코드를 실행하고 Kubernetes 커뮤니티에서 정의한 SLOs 내에서 EKS 클러스터가 작동하도록 합니다. Kubernetes SIG(Scalability Special Interest Group)는 확장성 목표를 정의하고 SLIs 및 SLOs.

SLIs는 요청 지연 시간 또는 수와 같이 시스템이 어떻게 "잘" 실행되고 있는지 결정하는 데 사용할 수 있는 지표 또는 측정치와 같은 시스템을 측정하는 방법입니다. SLOs는 요청 지연 시간이 3초 미만으로 유지되는 등 시스템이 "잘" 실행될 때 예상되는 값을 정의합니다. Kubernetes SLOs 및 SLIs는 Kubernetes 구성 요소의 성능에 초점을 맞추고 EKS 클러스터 엔드포인트의 가용성에 초점을 맞춘 Amazon EKS Service SLAs와 완전히 독립적입니다.

Kubernetes에는 CSI 드라이버, 승인 웹후크 및 오토 스케일러와 같은 사용자 지정 추가 기능 또는 드라이버로 시스템을 확장할 수 있는 여러 기능이 있습니다. 이러한 확장은 다양한 방식으로 Kubernetes 클러스터의 성능에 상당한 영향을 미칠 수 있습니다. 즉,를 사용하는 승인 웹후크는 웹후크 대상을 사용할 수 없는 경우 K8s API 요청에 지연 시간을 추가할 `failurePolicy=Ignore` 수 있습니다. Kubernetes Scalability SIG는 "You promise, We promise" 프레임워크를 사용하여 확장성을 정의합니다.

> 약속한 경우: - 클러스터를 올바르게 구성 - 확장성 기능 "합리적으로" 사용 - 클러스터의 로드를 권장 제한 이내로 유지
>
> 그러면 클러스터가 확장됩니다. 즉, 모든 SLOs 충족됩니다.

Kubernetes SLOs

Kubernetes SLOs는 작업자 노드 조정 또는 승인 웹후크와 같이 클러스터에 영향을 미칠 수 있는 모든 플러그인 및 외부 제한 사항을 고려하지 않습니다. 이러한 SLOs Kubernetes 구성 요소에 초점을 맞추고 Kubernetes 작업 및 리소스가 예상대로 작동하는지 확인합니다. SLOs는 Kubernetes 개발자가 Kubernetes 코드 변경으로 인해 전체 시스템의 성능이 저하되지 않도록 하는 데 도움이 됩니다.

Kuberntes Scalability SIG는 다음과 같은 공식 SLO/SLIs 정의합니다. Amazon EKS 팀은 이러한 SLOs/SLIs에 대해 EKS 클러스터에서 정기적으로 확장성 테스트를 실행하여 변경 사항이 적용되고 새 버전이 릴리스될 때 성능 저하를 모니터링합니다.

목표	정의	SLO
API 요청 지연 시간(변환 중)	지난 5분 동안 99번째 백분위수로 측정된 모든 (리소스, 동	기본 Kubernetes 설치에서는 가상 및 집계 리소스와 사용자

목표	정의	SLO
	사) 페어의 단일 객체에 대한 변형 API 호출 처리 지연 시간	지정 리소스 정의를 제외한 모든 (리소스, 동사) 페어에 대해 클러스터 데이당 99번째 백분위수 <= 1초
API 요청 지연 시간(읽기 전용)	지난 5분 동안 99번째 백분위수로 측정된 모든 (리소스, 범위) 페어에 대한 비스트리밍 읽기 전용 API 호출 처리 지연 시간	기본 Kubernetes 설치에서는 가상 및 집계 리소스와 사용자 지정 리소스 정의를 제외한 모든 (리소스, 범위) 페어에 대해 클러스터 데이당 99번째 백분위수: (a) <= 1s if `scope=resource` (b) <= 30s otherwise (`scope=namespace` 또는 인 경우`scope=cluster`)
포드 시작 지연 시간	포드 생성 타임스탬프부터 모든 컨테이너가 시작으로 보고되고 감시를 통해 관찰될 때까지 측정된 이미지를 가져오고 초기화 컨테이너를 실행하는 시간을 제외한 예약 가능한 상태 비저장 포드의 시작 지연 시간, 지난 5분 동안 99번째 백분위수로 측정됨	기본 Kubernetes 설치에서 클러스터 데이당 99번째 백분위수 <= 5초

API 요청 지연 시간

`kube-apiserver`는 `1m0s` 기본적으로 로 `--request-timeout` 정의되어 있습니다. 즉, 제한 시간이 초과되고 취소되기 전에 최대 1분(60초) 동안 요청을 실행할 수 있습니다. 지연 시간에 대해 정의된 SLOs는 변경 중이거나 읽기 전용일 수 있는 요청 유형으로 구분됩니다.

변경

Kubernetes에서 요청을 변경하면 생성, 삭제 또는 업데이트와 같은 리소스가 변경됩니다. 업데이트된 객체가 반환되기 전에 이러한 변경 사항을 etcd 백엔드에 작성해야 하므로 이러한 요청은 비용이 많이 듭니다. Etcd는 모든 Kubernetes 클러스터 데이터에 사용되는 분산 키-값 저장소입니다.

이 지연 시간은 Kubernetes 리소스의 (리소스, 동사) 페어에 대해 5분 동안 99번째 백분위수로 측정됩니다. 예를 들어, 이는 포드 생성 요청 및 노드 업데이트 요청의 지연 시간을 측정합니다. SLO를 충족하려면 요청 지연 시간이 <= 1초여야 합니다.

읽기 전용

읽기 전용 요청은 단일 리소스(예: Get Pod X) 또는 컬렉션(예: “네임스페이스 X에서 모든 포드 가져오기”)을 검색합니다. 는 객체의 캐시를 `kube-apiserver` 유지하므로 요청된 리소스가 캐시에서 반환되거나 먼저 etcd에서 검색해야 할 수 있습니다. 이러한 지연 시간은 5분 동안 99번째 백분위수로도 측정되지만 읽기 전용 요청에는 별도의 범위가 있을 수 있습니다. SLO는 두 가지 목표를 정의합니다.

- 단일 리소스(예: `kubectl get pod -n mynamespace my-controller-xxx`)에 대한 요청의 경우 요청 지연 시간은 <= 1초로 유지되어야 합니다.
- 네임스페이스 또는 클러스터(예: `kubectl get pods -A`)의 여러 리소스에 대해 이루어진 요청의 경우 지연 시간은 <= 30초로 유지되어야 합니다.

Kubernetes 리소스 목록에 대한 요청의 경우 요청의 모든 객체에 대한 세부 정보가 SLO 내에서 반환될 것으로 예상되므로 SLO의 대상 값은 서로 다릅니다. 대규모 클러스터 또는 대규모 리소스 모음에서 이로 인해 응답 크기가 커져 반환하는 데 다소 시간이 걸릴 수 있습니다. 예를 들어, JSON으로 인코딩될 때 각 포드가 약 1KiB인 수만 개의 포드를 실행하는 클러스터에서 클러스터의 모든 포드를 반환하는 작업은 10MB 이상으로 구성됩니다. Kubernetes 클라이언트는 APIListChunking을 사용하여 대규모 리소스 컬렉션을 검색하여이 응답 크기를 줄이는 데 도움이 될 수 있습니다.

포드 시작 지연 시간

이 SLO는 주로 포드 생성부터 해당 포드의 컨테이너가 실제로 실행을 시작하는 데 걸리는 시간과 관련이 있습니다. 이를 측정하기 위해 포드에 기록된 생성 타임스탬프와 차이를 측정하고 해당 포드의 WATCH가 컨테이너가 시작되었음을 보고하는 시점을 계산합니다(컨테이너 이미지 가져오기 및 컨테이너 실행 시작 시간 제외). SLO를 충족하려면이 포드 시작 지연 시간의 클러스터-일당 99번째 백분위수가 <=5초로 유지되어야 합니다.

이 SLO는 포드를 예약할 준비가 된 상태로이 클러스터에 작업자 노드가 이미 있다고 가정합니다. 이 SLO는 이미지 가져오기 또는 컨테이너 실행 시작을 고려하지 않으며 영구 스토리지 플러그인을 활용하지 않는 "상태 비저장 포드"로 테스트도 제한합니다.

Kubernetes SLI 지표

또한 Kubernetes는 시간 경과에 따라 이러한 SLIs 추적하는 Kubernetes 구성 요소에 Prometheus 지표를 추가하여 SLIs에 대한 관찰성을 개선하고 있습니다. Prometheus 쿼리 언어(PromQL)를 사용하면

Prometheus 또는 Grafana 대시보드와 같은 도구에서 시간 경과에 따른 SLI 성능을 표시하는 쿼리를 구축할 수 있습니다. 다음은 위의 SLOs에 대한 몇 가지 예입니다.

API 서버 요청 지연 시간

지표	정의
apiserver_request_sli_duration_seconds	각 동사, 그룹, 버전, 리소스, 하위 리소스, 범위 및 구성 요소에 대한 초 단위의 응답 지연 시간 분포(웹후크 기간 및 우선 순위 및 공정성 대기열 대기 시간은 계산하지 않음).
apiserver_request_duration_seconds	각 동사, 드라이 런 값, 그룹, 버전, 리소스, 하위 리소스, 범위 및 구성 요소에 대한 응답 지연 시간 분포(초 단위).

> **Note**
>
> apiserver_request_sli_duration_seconds 지표는 Kubernetes 1.27부터 사용할 수 있습니다.

이러한 지표를 사용하여 API 서버 응답 시간과 Kubernetes 구성 요소 또는 기타 플러그인/구성 요소에 병목 현상이 있는지 조사할 수 있습니다. 아래 쿼리는 커뮤니티 SLO 대시보드를 기반으로 합니다.

API 요청 지연 시간 SLI(변환 중) -이 시간에는 웹후크 실행 또는 대기열 대기 시간이 포함되지 않습니다. `histogram_quantile(0.99, sum(rate(apiserver_request_sli_duration_seconds_bucket{verb=~"CREATE|DELETE|PATCH|POST|PUT", subresource!~"proxy|attach|log|exec|portforward"}[5m])) by (resource, subresource, verb, scope, le)) > 0`

API 요청 지연 시간 합계(변환) - 요청이 API 서버에서 소요된 총 시간입니다.이 시간은 웹후크 실행과 API 우선 순위 및 공정성 대기 시간을 포함하므로 SLI 시간보다 길 수 있습니다. `histogram_quantile(0.99, sum(rate(apiserver_request_duration_seconds_bucket{verb=~"CREATE|DELETE|PATCH|POST|PUT", subresource!~"proxy|attach|log|exec|portforward"}[5m])) by (resource, subresource, verb, scope, le)) > 0`

이러한 쿼리에서는 kubectl port-forward 또는 요청(subresource!~"proxy|attach|log|exec|portforward")과 같이 즉시 반환되지 않는 스트리밍 API kubectl exec 요청을 제외하며 객체를 수정하는 Kubernetes 동사()에 대해서만 필터링합니다verb=~"CREATE|DELETE|PATCH|POST|PUT". 그런 다음 지난 5분 동안의 해당 지연 시간의 99번째 백분위수를 계산합니다.

읽기 전용 API 요청에 유사한 쿼리를 사용할 수 있으며, 필터링하려는 동사를 수정하여 읽기 전용 작업 LIST 및를 포함할 수 있습니다GET. 요청 범위에 따라 단일 리소스를 가져오거나 여러 리소스를 나열하는 등 다양한 SLO 임계값도 있습니다.

API 요청 지연 시간 SLI(읽기 전용) -이 시간에는 웹후크 실행 또는 대기열 대기 시간이 포함되지 않습니다. 단일 리소스의 경우(범위=리소스, 임계값=1초) histogram_quantile(0.99, sum(rate(apiserver_request_sli_duration_seconds_bucket{verb=~"GET", scope=~"resource"}[5m])) by (resource, subresource, verb, scope, le))

동일한 네임스페이스(scope=namespace, threshold=5s)에 있는 리소스 모음의 경우 histogram_quantile(0.99, sum(rate(apiserver_request_sli_duration_seconds_bucket{verb=~"LIST", scope=~"namespace"}[5m])) by (resource, subresource, verb, scope, le))

전체 클러스터의 리소스 모음(범위=클러스터, 임계값=30초) histogram_quantile(0.99, sum(rate(apiserver_request_sli_duration_seconds_bucket{verb=~"LIST", scope=~"cluster"}[5m])) by (resource, subresource, verb, scope, le))

API 요청 지연 시간 합계(읽기 전용) - API 서버에서 요청이 걸린 총 시간입니다. 이 시간은 웹후크 실행 및 대기 시간을 포함하므로 SLI 시간보다 길 수 있습니다. 단일 리소스의 경우(범위=리소스, 임계값=1초) histogram_quantile(0.99, sum(rate(apiserver_request_duration_seconds_bucket{verb=~"GET", scope=~"resource"}[5m])) by (resource, subresource, verb, scope, le))

동일한 네임스페이스(scope=namespace, threshold=5s)에 있는 리소스 모음의 경우 histogram_quantile(0.99, sum(rate(apiserver_request_duration_seconds_bucket{verb=~"LIST", scope=~"namespace"}[5m])) by (resource, subresource, verb, scope, le))

전체 클러스터의 리소스 모음(범위=클러스터, 임계값=30초) histogram_quantile(0.99, sum(rate(apiserver_request_duration_seconds_bucket{verb=~"LIST", scope=~"cluster"}[5m])) by (resource, subresource, verb, scope, le))

SLI 지표는 요청이 API 우선 순위 및 공정성 대기열에서 대기하거나 승인 웹후크 또는 기타 Kubernetes 확장을 통해 작업하는 데 소요되는 시간을 제외하여 Kubernetes 구성 요소가 어떻게 작동

하는지에 대한 인사이트를 제공합니다. 총 지표는 애플리케이션이 API 서버의 응답을 기다리는 시간을 반영하므로 보다 전체적인 보기를 제공합니다. 이러한 지표를 비교하면 요청 처리 지연이 발생하는 위치를 파악할 수 있습니다.

포드 시작 지연 시간

지표	정의
kubelet_pod_start_sli_duration_seconds	포드 생성 타임스탬프부터 모든 컨테이너가 시작됨으로 보고되고 감시를 통해 관찰될 때까지 측정된 이미지를 가져오고 초기화 컨테이너를 실행하는 시간을 제외하고 포드를 시작하는 초 단위의 기간
kubelet_pod_start_duration_seconds	포드가 처음 표시되는 kubelet부터 포드가 실행되기 시작하는 시점까지의 초 단위 기간입니다. 여기에는 포드를 예약하거나 작업자 노드 용량을 스케일 아웃하는 시간은 포함되지 않습니다.

> **Note**
>
> `kubelet_pod_start_sli_duration_seconds`는 Kubernetes 1.27부터 사용할 수 있습니다.

위의 쿼리와 마찬가지로 이러한 지표를 사용하여 Kubelet 작업과 비교하여 노드 크기 조정, 이미지 가져오기 및 초기화 컨테이너가 포드 시작을 지연시키는 시간을 파악할 수 있습니다.

포드 시작 지연 시간 SLI - 포드가 생성된 시점부터 애플리케이션 컨테이너가 실행 중으로 보고된 시점까지의 시간입니다. 여기에는 작업자 노드 용량을 사용할 수 있는 데 걸리는 시간과 포드를 예약하는 데 걸리는 시간이 포함되지만 이미지를 가져오거나 초기화 컨테이너를 실행하는 데 걸리는 시간은 포함되지 않습니다. `histogram_quantile(0.99, sum(rate(kubelet_pod_start_sli_duration_seconds_bucket[5m])) by (le))`

포드 시작 지연 시간 합계 - 포드를 처음 시작하는 데 kubelet이 걸리는 시간입니다. 이는 kubelet이 WATCH를 통해 포드를 수신하는 시점부터 측정되며, 작업자 노드 조정 또는 예약 시간은 포함되지 않습니다. 여기에는 이미지를 가져오고 실행

할 컨테이너를 초기화하는 시간이 포함됩니다. `histogram_quantile(0.99, sum(rate(kubelet_pod_start_duration_seconds_bucket[5m])) by (le))`

클러스터SLOs

EKS 클러스터의 Kubernetes 리소스에서 Prometheus 지표를 수집하는 경우 Kubernetes 컨트롤 플레인 구성 요소의 성능에 대한 심층적인 인사이트를 얻을 수 있습니다.

성능 테스트 리포지토리에는 테스트 중에 클러스터의 지연 시간 및 중요 성능 지표를 표시하는 Grafana 대시보드가 포함되어 있습니다. 성능 테스트 구성은 Kubernetes 지표를 수집하도록 구성된 오픈 소스 프로젝트인 kube-prometheus-stack을 활용하지만 Amazon Managed Prometheus 및 Amazon Managed Grafana를 사용할 수도 있습니다.

`kube-prometheus-stack` 또는 유사한 Prometheus 솔루션을 사용하는 경우 동일한 대시보드를 설치하여 클러스터의 SLOs 실시간으로 관찰할 수 있습니다.

1. 먼저에서 대시보드에 사용되는 Prometheus 규칙을 설치해야 합니다`kubectl apply -f prometheus-rules.yaml`. https://github.com/kubernetes/perf-tests/blob/master/clusterloader2/pkg/prometheus/manifests/prometheus-rules.yaml 규칙 사본을 다운로드할 수 있습니다.
   a. 파일의 네임스페이스가 환경과 일치하는지 확인합니다.
   b. 를 사용하는 경우 레이블이 `prometheus.prometheusSpec.ruleSelector` helm 값과 일치하는지 확인합니다. `kube-prometheus-stack`
2. 그런 다음 Grafana에 대시보드를 설치할 수 있습니다. 이를 생성하기 위한 json 대시보드 및 python 스크립트는 https://github.com/kubernetes/perf-tests/tree/master/clusterloader2/pkg/prometheus/manifests/dashboards 확인할 수 있습니다.
   a. `slo.json` 대시보드는 Kubernetes SLOs

SLOs는 클러스터에 있는 Kubernetes 구성 요소의 성능에 초점을 맞추고 있지만 클러스터에 다양한 관점이나 인사이트를 제공하는 추가 지표를 검토할 수 있습니다. Kube-state-metrics 커뮤니티 프로젝트는 클러스터의 추세를 빠르게 분석하는 데 도움이 될 수 있습니다. Kubernetes 커뮤니티의 대부분의 일반적인 플러그인 및 드라이버는 Prometheus 지표도 생성하므로 오토스케일러 또는 사용자 지정 스케줄러와 같은 항목을 조사할 수 있습니다.

관찰성 모범 사례 가이드에는 추가 인사이트를 얻는 데 사용할 수 있는 다른 Kubernetes 지표의 예가 나와 있습니다.

알려진 제한 및 Service Quotas

Amazon EKS는 다양한 워크로드에 사용할 수 있으며 다양한 AWS 서비스와 상호 작용할 수 있습니다. 고객 워크로드에는 비슷한 범위의 AWS 서비스 할당량과 확장성을 저해하는 기타 문제가 발생하는 것을 확인했습니다.

AWS 계정에는 기본 할당량(팀이 요청할 수 있는 각 AWS 리소스 수의 상한)이 있습니다. 각 AWS 서비스는 자체 할당량을 정의하며 할당량은 일반적으로 리전별로 다릅니다. 일부 할당량(소프트 제한)에 대해 증가를 요청할 수 있으며, 다른 할당량은 늘릴 수 없습니다(하드 제한). 애플리케이션을 설계할 때는 이러한 값을 고려해야 합니다. 이러한 서비스 제한을 정기적으로 검토하고 애플리케이션 설계에 통합하는 것이 좋습니다.

계정의 사용량을 검토하고 AWS Service Quotas 콘솔에서 또는 AWS CLI를 사용하여 할당량 증가 요청을 열 수 있습니다. Service Quotas에 대한 자세한 내용과 증가에 대한 추가 제한 또는 공지는 해당 AWS Service의 AWS 설명서를 참조하세요.

> **Note**
>
> Amazon EKS Service Quotas는 서비스 할당량을 나열하고 가능한 경우 증가를 요청할 수 있는 링크가 있습니다.

기타 AWS Service Quotas

다른 AWS 서비스에 대해 아래 나열된 할당량의 영향을 받는 EKS 고객을 확인했습니다. 이 중 일부는 특정 사용 사례 또는 구성에만 적용될 수 있지만 솔루션이 확장될 때 이러한 문제가 발생하는지 고려할 수 있습니다. 할당량은 서비스별로 구성되며 각 할당량에는 AWS Service Quotas 콘솔에서 조회하는 데 사용할 수 있는 L-XXXXXXXX 형식의 ID가 있습니다.

서비스:	할당량(L-xxxxx)	Impact	ID(L-xxxxx)	기본값
IAM	계정당 역할 수	계정의 클러스터 또는 IRSA 역할 수를 제한할 수 있습니다.	L-FE177D64	1,000
IAM	계정당 OpenId 연결 공급자	계정당 클러스터 수를 제한할 수	L-858F3967	100

서비스:	할당량(L-xxxxx)	Impact	ID(L-xxxxx)	기본값
		있으며, OpenID Connect는 IRSA에서 사용됩니다.		
IAM	역할 신뢰 정책 길이	IRSA에 대해 IAM 역할이 연결된 클러스터 수를 제한할 수 있습니다.	L-C07B4B0D	2,048
VPC	네트워크 인터페이스당 보안 그룹	클러스터에 대한 네트워킹의 제어 또는 연결을 제한할 수 있습니다.	L-2AFB9258	5
VPC	VPC당 IPv4 CIDR 블록	EKS 작업자 노드 수 제한 가능	L-83CA0A9D	5
VPC	라우팅 테이블에 따른 경로	클러스터에 대한 네트워킹의 제어 또는 연결을 제한할 수 있습니다.	L-93826ACB	50
VPC	VPC당 활성 VPC 피어링 연결	클러스터에 대한 네트워킹의 제어 또는 연결을 제한할 수 있습니다.	L-7E9ECCDB	50
VPC	보안 그룹당 인바운드 또는 아웃바운드 규칙.	클러스터에 대한 네트워킹의 제어 또는 연결을 제한할 수 있으며, EKS의 일부 컨트롤러는 새 규칙을 생성합니다.	L-0EA8095F	50

서비스:	할당량(L-xxxxx)	Impact	ID(L-xxxxx)	기본값
VPC	리전당 VPC	계정당 클러스터 수 또는 클러스터에 대한 네트워킹의 제어 또는 연결을 제한할 수 있습니다.	L-F678F1CE	5
VPC	리전당 인터넷 게이트웨이	계정당 클러스터 수 또는 클러스터에 대한 네트워킹의 제어 또는 연결을 제한할 수 있습니다.	L-A4707A72	5
VPC	리전당 네트워크 인터페이스	EKS 작업자 노드 수를 제한하거나 EKS 컨트롤 플레인 조정/업데이트 활동에 영향을 미칠 수 있습니다.	L-DF5E4CA3	5,000
VPC	네트워크 주소 사용량	계정당 클러스터 수 또는 클러스터에 대한 네트워킹의 제어 또는 연결을 제한할 수 있습니다.	L-BB24F6E5	64,000
VPC	피어링된 네트워크 주소 사용량	계정당 클러스터 수 또는 클러스터에 대한 네트워킹의 제어 또는 연결을 제한할 수 있습니다.	L-CD17FD4B	128,000

서비스:	할당량(L-xxxxx)	Impact	ID(L-xxxxx)	기본값
ELB	Network Load Balancer당 리스너	클러스터로의 트래픽 수신 제어를 제한할 수 있습니다.	L-57A373D6	50
ELB	리전당 대상 그룹	클러스터로의 트래픽 수신 제어를 제한할 수 있습니다.	L-B22855CB	3,000
ELB	Application Load Balancer당 대상	클러스터로의 트래픽 수신 제어를 제한할 수 있습니다.	L-7E6692B2	1,000
ELB	Network Load Balancer당 대상	클러스터로의 트래픽 수신 제어를 제한할 수 있습니다.	L-EEF1AD04	3,000
ELB	Network Load Balancer당 가용 영역당 대상	클러스터로의 트래픽 수신 제어를 제한할 수 있습니다.	L-B211E961	500
ELB	리전당 대상 그룹당 대상	클러스터로의 트래픽 수신 제어를 제한할 수 있습니다.	L-A0D0B863	1,000
ELB	리전당 Application Load Balancer	클러스터로의 트래픽 수신 제어를 제한할 수 있습니다.	L-53DA6B97	50

서비스:	할당량(L-xxxxx)	Impact	ID(L-xxxxx)	기본값
ELB	리전당 Classic Load Balancer	클러스터로의 트래픽 수신 제어를 제한할 수 있습니다.	L-E9E9831D	20
ELB	리전당 Network Load Balancer	클러스터로의 트래픽 수신 제어를 제한할 수 있습니다.	L-69A177A2	50
EC2	온디맨드 표준(A, C, D, H, I, M, R, T, Z) 인스턴스 실행(최대 vCPU 수)	EKS 작업자 노드 수 제한 가능	L-1216C47A	5
EC2	모든 표준(A, C, D, H, I, M, R, T, Z) 스팟 인스턴스 요청(최대 vCPU 수)	EKS 작업자 노드 수 제한 가능	L-34B43A08	5
EC2	EC2-VPC 탄력적 IP	NAT GWs(따라서 VPCs) 수를 제한하여 리전의 클러스터 수를 제한할 수 있습니다.	L-0263D0A3	5
EBS	지역당 스냅샷 수	상태 저장 워크로드에 대한 백업 전략을 제한할 수 있습니다.	L-309BACF6	100,000건

서비스:	할당량(L-xxxxx)	Impact	ID(L-xxxxx)	기본값
EBS	범용 SSD(gp3) 볼륨을 위한 스토리지(TiB)	EKS 작업자 노드 또는 PersistentVolume	L-7A658B76	50
EBS	범용 SSD(gp2) 볼륨을 위한 스토리지(TiB)	EKS 작업자 노드 또는 PersistentVolume	L-D18FCD1D	50
ECR	등록된 리포지토리	클러스터의 워크로드 수를 제한할 수 있습니다.	L-CFEB8E8D	100,000건
ECR	리포지토리당 이미지 수	클러스터의 워크로드 수를 제한할 수 있습니다.	L-03A36CE1	20,000건
SecretsManager	리전당 보안 암호	클러스터의 워크로드 수를 제한할 수 있습니다.	L-2F66C23C	500,000

AWS 요청 조절

또한 AWS 서비스는 모든 고객이 성능을 유지하고 사용할 수 있도록 요청 제한을 구현합니다. Service Quotas와 마찬가지로 각 AWS 서비스는 자체 요청 제한 임계값을 유지합니다. 워크로드에서 많은 수의 API 호출을 빠르게 실행해야 하거나 애플리케이션에서 요청 제한 오류가 발생하는 경우 해당 AWS 서비스 설명서를 검토하는 것이 좋습니다.

EC2 네트워크 인터페이스 또는 IP 주소 프로비저닝과 관련된 EC2 API 요청에는 대규모 클러스터에서 또는 클러스터가 크게 확장될 때 요청 제한이 발생할 수 있습니다. 아래 표에는 고객이 요청 제한에 직면한 몇 가지 API 작업이 나와 있습니다. 속도 제한에 대한 EC2 설명서에서 EC2 속도 제한 기본값과 속도 제한 증가를 요청하는 단계를 검토할 수 있습니다. EC2

작업 변경	읽기 전용 작업
AssignPrivateIpAddresses	DescribeDhcpOptions

작업 변경	읽기 전용 작업
AttachNetworkInterface	DescribeInstances
CreateNetworkInterface	DescribeNetworkInterfaces
DeleteNetworkInterface	DescribeSecurityGroups
DeleteTags	DescribeTags
DetachNetworkInterface	DescribeVpcs
ModifyNetworkInterfaceAttribute	DescribeVolumes
UnassignPrivateIpAddresses	

기타 알려진 제한

- 또한 Route 53에는 Route 53 API에 대한 초당 요청 5개의 매우 낮은 속도 제한이 있습니다. 외부 DNS와 같은 프로젝트로 업데이트할 도메인이 많은 경우 도메인 업데이트에 속도 제한 및 지연이 발생할 수 있습니다.
  - 일부 Nitro 인스턴스 유형에는 Amazon EBS 볼륨, 네트워크 인터페이스 및 NVMe 인스턴스 스토어 볼륨 간에 공유되는 볼륨 연결 제한이 28개입니다. NVMe 워크로드가 많은 EBS 볼륨을 탑재하는 경우 이러한 인스턴스 유형으로 달성할 수 있는 포드 밀도에 제한이 발생할 수 있습니다.
  - Ec2 인스턴스당 추적할 수 있는 최대 연결 수가 있습니다. 워크로드가 많은 수의 연결을 처리하는 경우이 최대값에 도달했기 때문에 통신 실패 또는 오류가 발생할 수 있습니다. `conntrack_allowance_available` 및 `conntrack_allowance_exceeded` 네트워크 성능 지표를 사용하여 EKS 작업자 노드에서 추적된 연결 수를 모니터링할 수 있습니다.
  - EKS 환경에서 etcd 스토리지 한도는 업스트림 지침에 따라 8GiB입니다. etcd db 크기를 추적`apiserver_storage_size_bytes`하려면 지표를 모니터링하십시오. 알림 규칙 `etcdBackendQuotaLowSpace` 및를 참조하여이 모니터링을 설정할 수 `etcdExcessiveDatabaseGrowth` 있습니다.

클러스터 업그레이드 모범 사례

이 가이드는 클러스터 관리자에게 Amazon EKS 업그레이드 전략을 계획하고 실행하는 방법을 보여줍니다. 또한 자체 관리형 노드, 관리형 노드 그룹, Karpenter 노드 및 Fargate 노드를 업그레이드하는 방법도 설명합니다. 여기에는 EKS Anywhere, 자체 관리형 Kubernetes, AWS Outposts 또는 AWS Local Zones에 대한 지침은 포함되지 않습니다.

개요

Kubernetes 버전은 컨트롤 플레인과 데이터 플레인을 모두 포함합니다. 원활한 작동을 보장하려면 컨트롤 플레인과 데이터 플레인 모두 1.24와 같은 동일한 Kubernetes 마이너 버전을 실행해야 합니다. AWS가 컨트롤 플레인을 관리하고 업그레이드하는 동안 데이터 플레인의 작업자 노드를 업데이트하는 것은 사용자의 책임입니다.

- 컨트롤 플레인 - 컨트롤 플레인의 버전은 Kubernetes API 서버에 의해 결정됩니다. Amazon EKS 클러스터에서 AWS는이 구성 요소를 관리합니다. AWS API를 통해 컨트롤 플레인 업그레이드를 시작할 수 있습니다.
- 데이터 영역 - 데이터 영역 버전은 개별 노드에서 실행되는 Kubelet 버전과 연결됩니다. 동일한 클러스터의 노드가 다른 버전을 실행하도록 할 수 있습니다. 를 실행하여 모든 노드의 버전을 확인할 수 있습니다`kubectl get nodes`.

업그레이드 전

Amazon EKS에서 Kubernetes 버전을 업그레이드하려는 경우 업그레이드를 시작하기 전에 마련해야 하는 몇 가지 중요한 정책, 도구 및 절차가 있습니다.

- 사용 중단 정책 이해 - Kubernetes 사용 중단 정책의 작동 방식을 심층적으로 이해합니다. 기존 애플리케이션에 영향을 미칠 수 있는 예정된 변경 사항에 유의하십시오. 최신 버전의 Kubernetes는 종종 특정 APIs 및 기능을 단계적으로 폐지하여 애플리케이션 실행에 문제를 일으킬 수 있습니다.
- Kubernetes 변경 로그 검토 - Amazon EKS Kubernetes 버전과 함께 Kubernetes 변경 로그를 철저히 검토하여 워크로드에 영향을 미칠 수 있는 변경 사항 중단과 같이 클러스터에 미칠 수 있는 영향을 파악합니다.
- 클러스터 추가 기능 호환성 평가 - Amazon EKS는 새 버전이 릴리스되거나 클러스터를 새 Kubernetes 마이너 버전으로 업데이트한 후 추가 기능을 자동으로 업데이트하지 않습니다. 추가 기능 업데이트를 검토하여 업그레이드하려는 클러스터 버전과 기존 클러스터 추가 기능의 호환성을 이해합니다.

- 컨트롤 플레인 로깅 활성화 - 컨트롤 플레인 로깅을 활성화하여 업그레이드 프로세스 중에 발생할 수 있는 로그, 오류 또는 문제를 캡처합니다. 이러한 로그에서 이상이 있는지 검토하는 것이 좋습니다. 비프로덕션 환경에서 클러스터 업그레이드를 테스트하거나 자동화된 테스트를 지속적 통합 워크플로에 통합하여 애플리케이션, 컨트롤러 및 사용자 지정 통합과의 버전 호환성을 평가합니다.
- 클러스터 관리를 위한 eksctl 살펴보기 - eksctl을 사용하여 EKS 클러스터를 관리하는 것이 좋습니다. 컨트롤 플레인을 업데이트하고, 추가 기능을 관리하고, 작업자 노드 업데이트를 즉시 처리할 수 있는 기능을 제공합니다. out-of-the-box
- Auto Mode 또는 Managed Node Groups 옵트인 - Auto Mode 또는 EKS 관리형 노드 그룹을 사용하여 작업자 노드 업그레이드를 간소화하고 자동화합니다. 이러한 옵션은 프로세스를 간소화하고 수동 개입을 줄입니다.
- kubectl 변환 플러그인 활용 - kubectl 변환 플러그인을 활용하여 다양한 API 버전 간에 Kubernetes 매니페스트 파일을 쉽게 변환할 수 있습니다. 이렇게 하면 구성이 새 Kubernetes 버전과 호환되도록 할 수 있습니다.

클러스터up-to-date 유지

Amazon EKS의 공동 책임 모델을 반영하여 안전하고 효율적인 EKS 환경을 위해서는 Kubernetes 업데이트를 최신 상태로 유지하는 것이 중요합니다. 이러한 전략을 운영 워크플로에 통합하면 최신 기능과 개선 사항을 최대한 활용하는 안전한 up-to-date 클러스터를 유지할 수 있습니다. 전술:

- 지원되는 버전 정책 - Kubernetes 커뮤니티에 따라 Amazon EKS는 일반적으로 세 가지 활성 Kubernetes 버전을 제공합니다. Kubernetes 마이너 버전은 릴리스 후 처음 14개월 동안 Amazon EKS에서 표준 지원을 받습니다. 버전 표준 지원 종료일이 지나면 다음 12개월 동안의 추가 지원이 시작됩니다. 사용 중단 알림은 버전이 표준 지원 종료일에 도달하기 최소 60일 전에 발행됩니다. 자세한 내용은 EKS 버전 수명 주기 문서를 참조하세요.
- 자동 업그레이드 정책 - EKS 클러스터의 Kubernetes 업데이트와 동기화하는 것이 좋습니다. 26개월의 수명 주기(14개월의 표준 지원 + 12개월의 확장 지원)를 마친 Kubernetes 버전에서 실행되는 클러스터는 다음 버전으로 자동 업그레이드됩니다. 추가 지원을 비활성화할 수 있습니다. 버전의 end-of-life 전에 사전 예방적 업그레이드에 실패하면 자동 업그레이드가 트리거되어 워크로드와 시스템이 중단될 수 있습니다. 자세한 내용은 EKS 버전 FAQs.
- 업그레이드 런북 생성 - 업그레이드를 관리하기 위한 잘 문서화된 프로세스를 설정합니다. 선제적 접근 방식의 일환으로 업그레이드 프로세스에 맞는 런북과 특수 도구를 개발합니다. 이렇게 하면 준비 상태가 향상될 뿐만 아니라 복잡한 전환도 간소화됩니다. 최소 1년에 한 번 클러스터를 업그레이드하는 것을 표준 관행으로 삼으세요. 이 관행은 지속적인 기술 발전에 맞게 조정되므로 환경의 효율성과 보안이 향상됩니다.

EKS 릴리스 일정 검토

EKS Kubernetes 릴리스 일정을 검토하여 새 버전이 출시되는 시기와 특정 버전에 대한 지원이 종료되는 시기를 알아봅니다. 일반적으로 EKS는 매년 세 가지 마이너 버전의 Kubernetes를 릴리스하며, 각 마이너 버전은 약 14개월 동안 지원됩니다.

또한 업스트림 Kubernetes 릴리스 정보를 검토합니다.

공동 책임 모델을 클러스터 업그레이드에 적용하는 방법 이해

클러스터 컨트롤 플레인과 데이터 플레인 모두에 대한 업그레이드를 시작할 책임은 사용자에게 있습니다. 업그레이드를 시작하는 방법을 알아봅니다. 클러스터 업그레이드를 시작하면 AWS가 클러스터 컨트롤 플레인 업그레이드를 관리합니다. Fargate 포드 및 추가 기능을 포함하여 데이터 영역을 업그레이드하는 것은 사용자의 책임입니다. 클러스터 업그레이드 후 가용성 및 작업이 영향을 받지 않도록 클러스터에서 실행되는 워크로드에 대한 업그레이드를 검증하고 계획해야 합니다.

클러스터 인플레이스 업그레이드

EKS는 인플레이스 클러스터 업그레이드 전략을 지원합니다. 이렇게 하면 클러스터 리소스가 유지되고 클러스터 구성이 일관되게 유지됩니다(예: API 엔드포인트, OIDC, ENIs, 로드 밸런서). 이는 클러스터 사용자에게 덜 방해가 되며 워크로드를 재배포하거나 외부 리소스(예: DNS, 스토리지)를 마이그레이션할 필요 없이 클러스터의 기존 워크로드와 리소스를 사용합니다.

인플레이스 클러스터 업그레이드를 수행할 때는 한 번에 하나의 마이너 버전 업그레이드만 실행할 수 있다는 점에 유의해야 합니다(예: 1.24에서 1.25로).

즉, 여러 버전을 업데이트해야 하는 경우 일련의 순차적 업그레이드가 필요합니다. 순차적 업그레이드를 계획하는 것은 더 복잡하며 가동 중지 위험이 더 높습니다. 이 경우 단원을 참조하십시오the section called “인플레이스 클러스터 업그레이드의 대안으로 블루/그린 클러스터 평가”.

컨트롤 플레인 및 데이터 플레인을 순서대로 업그레이드

클러스터를 업그레이드하려면 다음 작업을 수행해야 합니다.

1. Kubernetes 및 EKS 릴리스 정보를 검토합니다.
2. 클러스터를 백업합니다(선택 사항).
3. 워크로드에서 더 이상 사용되지 않거나 제거된 API 사용량을 식별하고 해결합니다.

4. 관리형 노드 그룹을 사용하는 경우 관리형 노드 그룹이 컨트롤 플레인과 동일한 Kubernetes 버전에 있는지 확인합니다. EKS Fargate Profiles에서 생성한 EKS 관리형 노드 그룹 및 노드는 Kubernetes 버전 1.27 이하의 컨트롤 플레인과 데이터 플레인 간에 2개의 마이너 버전 스큐를 지원합니다. 1.28 이상부터 EKS Fargate Profiles에서 생성한 EKS 관리형 노드 그룹 및 노드는 컨트롤 플레인과 데이터 플레인 간에 3개의 마이너 버전 스큐를 지원합니다. 예를 들어 EKS 컨트롤 플레인 버전이 1.28인 경우 kubelet 버전을 1.25 이전 버전으로 안전하게 사용할 수 있습니다. EKS 버전이 1.27인 경우 사용할 수 있는 가장 오래된 kubelet 버전은 1.25입니다.
5. AWS 콘솔 또는 cli를 사용하여 클러스터 컨트롤 플레인을 업그레이드합니다.
6. 추가 기능 호환성을 검토합니다. 필요에 따라 Kubernetes 추가 기능 및 사용자 지정 컨트롤러를 업그레이드합니다.
7. kubectl을 업데이트합니다.
8. 클러스터 데이터 영역을 업그레이드합니다. 노드를 업그레이드된 클러스터와 동일한 Kubernetes 마이너 버전으로 업그레이드합니다.

> **Tip**
>
> EKS Auto Mode를 사용하여 클러스터를 생성한 경우 클러스터 데이터 영역을 업그레이드할 필요가 없습니다. 컨트롤 플레인을 업그레이드한 후 EKS Auto Mode는 모든 포드 중단 예산을 준수하면서 관리형 노드를 점진적으로 업데이트하기 시작합니다. 이러한 업데이트를 모니터링하여 운영 요구 사항을 준수하는지 확인합니다.

EKS 설명서를 사용하여 업그레이드 체크리스트 생성

EKS Kubernetes 버전 설명서에는 각 버전에 대한 자세한 변경 사항 목록이 포함되어 있습니다. 각 업그레이드에 대한 체크리스트를 작성합니다.

특정 EKS 버전 업그레이드 지침은 설명서를 검토하여 각 버전에 대한 주목할 만한 변경 사항 및 고려 사항을 확인하세요.

- 표준 지원에서 Kubernetes 버전 릴리스 정보 검토
- 추가 지원 시 Kubernetes 버전 릴리스 정보 검토

Kubernetes API를 사용하여 추가 기능 및 구성 요소 업그레이드

클러스터를 업그레이드하기 전에 사용 중인 Kubernetes 구성 요소의 버전을 이해해야 합니다. 클러스터 구성 요소를 인벤토리로 만들고 Kubernetes API를 직접 사용하는 구성 요소를 식별합니다. 여기에는 모니터링 및 로깅 에이전트, 클러스터 오토스케일러, 컨테이너 스토리지 드라이버(예: EBS CSI, EFS CSI), 수신 컨트롤러, Kubernetes API에 직접 의존하는 기타 워크로드 또는 추가 기능과 같은 중요한 클러스터 구성 요소가 포함됩니다.

> **Tip**
>
> 중요 클러스터 구성 요소는 종종 `*-system` 네임스페이스에 설치됩니다.

```
kubectl get ns | grep '-system'
```

Kubernetes API를 사용하는 구성 요소를 식별한 후에는 해당 설명서에서 버전 호환성 및 업그레이드 요구 사항을 확인하세요. 예를 들어 버전 호환성은 AWS Load Balancer Controller 설명서를 참조하세요. 클러스터 업그레이드를 진행하기 전에 일부 구성 요소를 업그레이드하거나 구성을 변경해야 할 수 있습니다. 확인할 몇 가지 중요한 구성 요소에는 CoreDNS, kube-proxy, VPC CNI 및 스토리지 드라이버가 포함됩니다.

클러스터에는 종종 Kubernetes API를 사용하는 많은 워크로드가 포함되어 있으며 수신 컨트롤러, 지속적 전송 시스템 및 모니터링 도구와 같은 워크로드 기능에 필요합니다. EKS 클러스터를 업그레이드할 때 추가 기능 및 타사 도구도 업그레이드하여 호환되는지 확인해야 합니다.

일반적인 추가 기능의 다음 예제와 관련 업그레이드 설명서를 참조하세요.

- Amazon VPC CNI: 각 클러스터 버전에 대한 Amazon VPC CNI 추가 기능의 권장 버전은 Kubernetes용 Amazon VPC CNI 플러그인 업데이트를 참조하세요. Amazon EKS 추가 기능으로 설치된 경우 한 번에 하나의 마이너 버전만 업그레이드할 수 있습니다.
- kube-proxy: Kubernetes kube-proxy 자체 관리형 추가 기능 업데이트를 참조하세요.
- CoreDNS: CoreDNS 자체 관리형 추가 기능 업데이트를 참조하세요.
- AWS Load Balancer Controller: AWS Load Balancer Controller는 배포한 EKS 버전과 호환되어야 합니다. 자세한 내용은 설치 안내서를 참조하세요.
- Amazon Elastic Block Store(Amazon EBS) 컨테이너 스토리지 인터페이스(CSI) 드라이버: 설치 및 업그레이드 정보는 Amazon EBS CSI 드라이버를 Amazon EKS 추가 기능으로 관리를 참조하세요.

- Amazon Elastic File System(Amazon EFS) 컨테이너 스토리지 인터페이스(CSI) 드라이버: 설치 및 업그레이드 정보는 Amazon EFS CSI 드라이버를 참조하세요.
- Kubernetes Metrics Server: 자세한 내용은 GitHub의 metrics-server를 참조하세요.
- Kubernetes Cluster Autoscaler: Kubernetes Cluster Autoscaler의 버전을 업그레이드하려면 배포에서 이미지의 버전을 변경합니다. Cluster Autoscaler는 Kubernetes 스케줄러와 긴밀하게 연결됩니다. 클러스터를 업그레이드할 때는 항상 업그레이드해야 합니다. GitHub 릴리스를 검토하여 Kubernetes 마이너 버전에 해당하는 최신 릴리스의 주소를 찾습니다.
- Karpenter: 설치 및 업그레이드 정보는 Karpenter 설명서를 참조하세요.

> **Tip**
>
> 컴퓨팅 오토 스케일링, 블록 스토리지 및 로드 밸런싱 기능을 포함하여 Amazon EKS Auto Mode의 기능을 수동으로 업그레이드할 필요가 없습니다.

업그레이드하기 전에 기본 EKS 요구 사항 확인

AWS에서 업그레이드 프로세스를 완료하려면 계정의 특정 리소스가 필요합니다. 이러한 리소스가 없으면 클러스터를 업그레이드할 수 없습니다. 컨트롤 플레인 업그레이드에는 다음 리소스가 필요합니다.

1. 사용 가능한 IP 주소: Amazon EKS는 클러스터를 업데이트할 때 지정한 서브넷에서 최대 5개의 사용 가능한 IP 주소가 필요합니다. 그렇지 않은 경우 버전 업데이트를 수행하기 전에 새 클러스터 서브넷을 포함하도록 클러스터 구성을 업데이트합니다.
2. EKS IAM 역할: 컨트롤 플레인 IAM 역할은 필요한 권한을 가진 계정에 여전히 존재합니다.
3. 클러스터에 보안 암호 암호화가 활성화된 경우 클러스터 IAM 역할에 AWS Key Management Service(AWS KMS) 키를 사용할 수 있는 권한이 있는지 확인합니다.

사용 가능한 IP 주소 확인

클러스터를 업데이트하려면 Amazon EKS에서 클러스터를 생성할 때 지정된 서브넷의 사용 가능한 IP 주소 최대 5개가 필요합니다.

서브넷에 클러스터를 업그레이드하기에 충분한 IP 주소가 있는지 확인하려면 다음 명령을 실행할 수 있습니다.

```
CLUSTER=<cluster name>
aws ec2 describe-subnets --subnet-ids \
  $(aws eks describe-cluster --name ${CLUSTER} \
  --query 'cluster.resourcesVpcConfig.subnetIds' \
  --output text) \
  --query 'Subnets[*].[SubnetId,AvailabilityZone,AvailableIpAddressCount]' \
  --output table

-------------------------------------------------------
|                  DescribeSubnets                    |
+---------------------------+--------------+-------+
|  subnet-067fa8ee8476abbd6 |  us-east-1a  |  8184 |
|  subnet-0056f7403b17d2b43 |  us-east-1b  |  8153 |
|  subnet-09586f8fb3addbc8c |  us-east-1a  |  8120 |
|  subnet-047f3d276a22c6bce |  us-east-1b  |  8184 |
+---------------------------+--------------+-------+
```

VPC CNI 지표 도우미를 사용하여 VPC 지표에 대한 CloudWatch 대시보드를 생성할 수 있습니다. Amazon EKS는 클러스터 생성 중에 처음 지정된 서브넷의 IP 주소가 부족한 경우 Kubernetes 버전 업그레이드를 시작하기 전에 "UpdateClusterConfiguration" API를 사용하여 클러스터 서브넷을 업데이트할 것을 권장합니다. 제공될 새 서브넷을 확인하십시오.

- 는 클러스터 생성 중에 선택된 것과 동일한 AZs 세트에 속합니다.
- 클러스터 생성 중에 제공된 것과 동일한 VPC에 속함

기존 VPC CIDR 블록의 IP 주소가 부족하면 추가 CIDR 블록을 연결하는 것이 좋습니다. AWS를 사용하면 추가 CIDR 블록을 기존 클러스터 VPC와 연결하여 IP 주소 풀을 효과적으로 확장할 수 있습니다. 이 확장은 추가 프라이빗 IP 범위(RFC 1918) 또는 필요한 경우 퍼블릭 IP 범위(비 RFC 1918)를 도입하여 수행할 수 있습니다. Amazon EKS가 새 CIDR을 사용하려면 새 VPC CIDR 블록을 추가하고 VPC 새로 고침을 완료하도록 허용해야 합니다. 그런 다음 새로 설정된 CIDR 블록을 기반으로 서브넷을 VPC로 업데이트할 수 있습니다.

EKS IAM 역할 확인

IAM 역할을 사용할 수 있고 계정에 올바른 수임 역할 정책이 있는지 확인하려면 다음 명령을 실행할 수 있습니다.

```
CLUSTER=<cluster name>
ROLE_ARN=$(aws eks describe-cluster --name ${CLUSTER} \
```

```
  --query 'cluster.roleArn' --output text)
aws iam get-role --role-name ${ROLE_ARN##*/} \
  --query 'Role.AssumeRolePolicyDocument'

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "eks.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

EKS 추가 기능으로 마이그레이션

Amazon EKS는 모든 클러스터에 대해 Kubernetes, 및 CoreDNS용 Amazon VPC CNI 플러그인kube-proxy과 같은 추가 기능을 자동으로 설치합니다. 추가 기능은 자체 관리형이거나 Amazon EKS 추가 기능으로 설치될 수 있습니다. Amazon EKS 추가 기능은 EKS API를 사용하여 추가 기능을 관리하는 대체 방법입니다.

Amazon EKS 추가 기능을 사용하여 단일 명령으로 버전을 업데이트할 수 있습니다. 예:

```
aws eks update-addon –cluster-name my-cluster –addon-name vpc-cni –addon-version
 version-number \
--service-account-role-arn arn:aws:iam::111122223333:role/role-name –configuration-
values '{}' –resolve-conflicts PRESERVE
```

다음을 사용하여 EKS 추가 기능이 있는지 확인합니다.

```
aws eks list-addons --cluster-name <cluster name>
```

> **Warning**
>
> EKS 추가 기능은 컨트롤 플레인 업그레이드 중에 자동으로 업그레이드되지 않습니다. EKS 추가 기능 업데이트를 시작하고 원하는 버전을 선택해야 합니다.

- 사용 가능한 모든 버전에서 호환되는 버전을 선택하는 것은 사용자의 책임입니다. 추가 기능 버전 호환성에 대한 지침을 검토합니다.
- Amazon EKS 추가 기능은 한 번에 하나의 마이너 버전만 업그레이드할 수 있습니다.

EKS 추가 기능으로 사용할 수 있는 구성 요소와 시작하는 방법에 대해 자세히 알아봅니다.

EKS 추가 기능에 사용자 지정 구성을 제공하는 방법을 알아봅니다.

컨트롤 플레인을 업그레이드하기 전에 제거된 API 사용량 식별 및 해결

EKS 컨트롤 플레APIs 사용량을 식별해야 합니다. 이렇게 하려면 실행 중인 클러스터 또는 렌더링된 정적 Kubernetes 매니페스트 파일을 확인할 수 있는 도구를 사용하는 것이 좋습니다.

정적 매니페스트 파일에 대해 검사를 실행하는 것이 일반적으로 더 정확합니다. 라이브 클러스터에 대해 실행되는 경우 이러한 도구는 거짓 긍정을 반환할 수 있습니다.

더 이상 사용되지 않는 Kubernetes API는 API가 제거되었음을 의미하지 않습니다. API 제거가 워크로드에 미치는 영향을 이해하려면 Kubernetes 사용 중단 정책을 확인해야 합니다.

클러스터 인사이트

Cluster Insights는 EKS 클러스터를 최신 버전의 Kubernetes로 업그레이드하는 기능에 영향을 미칠 수 있는 문제에 대한 조사 결과를 제공하는 기능입니다. 이러한 결과는 Amazon EKS에서 큐레이션하고 관리하며 문제 해결 방법에 대한 권장 사항을 제공합니다. Cluster Insights를 활용하면 최신 Kubernetes 버전으로 업그레이드하는 데 드는 노력을 최소화할 수 있습니다.

EKS 클러스터의 인사이트를 보려면 명령을 실행합니다.

```
aws eks list-insights --region <region-code> --cluster-name <my-cluster>

{
    "insights": [
        {
            "category": "UPGRADE_READINESS",
            "name": "Deprecated APIs removed in Kubernetes v1.29",
            "insightStatus": {
                "status": "PASSING",
```

```
                "reason": "No deprecated API usage detected within the last 30 days."
            },
            "kubernetesVersion": "1.29",
            "lastTransitionTime": 1698774710.0,
            "lastRefreshTime": 1700157422.0,
            "id": "123e4567-e89b-42d3-a456-579642341238",
            "description": "Checks for usage of deprecated APIs that are scheduled for
 removal in Kubernetes v1.29. Upgrading your cluster before migrating to the updated
 APIs supported by v1.29 could cause application impact."
        }
    ]
}
```

수신된 인사이트에 대한 보다 설명적인 출력을 위해 명령을 실행할 수 있습니다.

```
aws eks describe-insight --region <region-code> --id <insight-id> --cluster-name <my-
cluster>
```

Amazon EKS 콘솔에서 인사이트를 볼 수도 있습니다. 클러스터 목록에서 클러스터를 선택하면 `Upgrade Insights` 탭 아래에 인사이트 조사 결과가 있습니다.

에서 클러스터 인사이트를 찾으면 클러스터 업그레이드를 수행하기 전에 문제를 해결`"status": ERROR`해야 합니다. 명령을 실행`aws eks describe-insight`하여 다음과 같은 문제 해결 조언을 공유합니다.

영향을 받는 리소스:

```
"resources": [
      {
        "insightStatus": {
          "status": "ERROR"
        },
        "kubernetesResourceUri": "/apis/policy/v1beta1/podsecuritypolicies/null"
      }
]
```

APIs 더 이상 사용되지 않습니다.

```
"deprecationDetails": [
      {
        "usage": "/apis/flowcontrol.apiserver.k8s.io/v1beta2/flowschemas",
```

```
        "replacedWith": "/apis/flowcontrol.apiserver.k8s.io/v1beta3/flowschemas",
        "stopServingVersion": "1.29",
        "clientStats": [],
        "startServingReplacementVersion": "1.26"
      }
]
```

수행할 권장 작업:

```
"recommendation": "Update manifests and API clients to use newer Kubernetes APIs if
 applicable before upgrading to Kubernetes v1.26."
```

EKS 콘솔 또는 CLI를 통해 클러스터 인사이트를 활용하면 EKS 클러스터 버전을 성공적으로 업그레이드하는 프로세스의 속도를 높일 수 있습니다. * 공식 EKS 문서 * Cluster Insights 시작 블로그 리소스에 대해 자세히 알아보세요.

Kube-no-trouble

Kube-no-trouble은 명령이 있는 오픈 소스 명령줄 유틸리티입니다kubent. 인수 kubent 없이를 실행하면 현재 KubeConfig 컨텍스트를 사용하고 클러스터를 스캔하고 사용 중단되고 제거될 APIs가 포함된 보고서를 인쇄합니다.

```
kubent

4:17PM INF >>> Kube No Trouble `kubent` <<<
4:17PM INF version 0.7.0 (git sha d1bb4e5fd6550b533b2013671aa8419d923ee042)
4:17PM INF Initializing collectors and retrieving data
4:17PM INF Target K8s version is 1.24.8-eks-ffeb93d
4:1 INF Retrieved 93 resources from collector name=Cluster
4:17PM INF Retrieved 16 resources from collector name="Helm v3"
4:17PM INF Loaded ruleset name=custom.rego.tmpl
4:17PM INF Loaded ruleset name=deprecated-1-16.rego
4:17PM INF Loaded ruleset name=deprecated-1-22.rego
4:17PM INF Loaded ruleset name=deprecated-1-25.rego
4:17PM INF Loaded ruleset name=deprecated-1-26.rego
4:17PM INF Loaded ruleset name=deprecated-future.rego
__________________________________________________________________________________________
>>> Deprecated APIs removed in 1.25 <<<
------------------------------------------------------------------------------------------
KIND                NAMESPACE     NAME             API_VERSION      REPLACE_WITH
 (SINCE)
```

```
PodSecurityPolicy   <undefined>   eks.privileged   policy/v1beta1   <removed> (1.21.0)
```

정적 매니페스트 파일 및 헬름 패키지를 스캔하는 데에도 사용할 수 있습니다. 매니페스트를 배포하기 전에 문제를 식별하기 위해 지속적 통합(CI) 프로세스의 kubent 일부로를 실행하는 것이 좋습니다. 또한 매니페스트 스캔은 라이브 클러스터를 스캔하는 것보다 더 정확합니다.

Kube-no-trouble은 클러스터를 스캔하기 위한 적절한 권한을 샘플 서비스 계정 및 역할에 제공합니다.

Pluto

또 다른 옵션은 라이브 클러스터, 매니페스트 파일, 헬름 차트 스캔을 지원하고 CI 프로세스에 포함할 수 있는 GitHub 작업이 kubent 있기 때문에와 유사한 pluto입니다.

```
pluto detect-all-in-cluster

NAME             KIND                VERSION          REPLACEMENT   REMOVED
 DEPRECATED   REPL AVAIL
eks.privileged   PodSecurityPolicy   policy/v1beta1                 false     true
    true
```

리소스

업그레이드 전에 클러스터가 더 이상 사용되지 않는 APIs를 사용하지 않는지 확인하려면 다음을 모니터링해야 합니다.

- Kubernetes v1.19 apiserver_requested_deprecated_apis 이후 지표:

```
kubectl get --raw /metrics | grep apiserver_requested_deprecated_apis

apiserver_requested_deprecated_apis{group="policy",removed_release="1.25",resource="podsecurity
 1
```

- 가 로 k8s.io/deprecated 설정된 감사 로그의 이벤트true:

```
CLUSTER="<cluster_name>"
QUERY_ID=$(aws logs start-query \
 --log-group-name /aws/eks/${CLUSTER}/cluster \
 --start-time $(date -u --date="-30 minutes" "+%s") # or date -v-30M "+%s" on MacOS \
 --end-time $(date "+%s") \
```

```
 --query-string 'fields @message | filter `annotations.k8s.io/deprecated`="true"' \
 --query queryId --output text)

echo "Query started (query id: $QUERY_ID), please hold ..." && sleep 5 # give it some
 time to query

aws logs get-query-results --query-id $QUERY_ID
```

더 이상 사용되지 않는 APIs


```
{
    "results": [
        [
            {
                "field": "@message",
                "value": "{\"kind\":\"Event\",\"apiVersion\":\"audit.k8s.io/v1\",
\"level\":\"Request\",\"auditID\":\"8f7883c6-b3d5-42d7-967a-1121c6f22f01\",\"stage
\":\"ResponseComplete\",\"requestURI\":\"/apis/policy/v1beta1/podsecuritypolicies?
allowWatchBookmarks=true\\u0026resourceVersion=4131\\u0026timeout=9m19s\
\u0026timeoutSeconds=559\\u0026watch=true\",\"verb\":\"watch\",\"user\":{\"username
\":\"system:apiserver\",\"uid\":\"8aabfade-da52-47da-83b4-46b16cab30fa\",
\"groups\":[\"system:masters\"]},\"sourceIPs\":[\"::1\"],\"userAgent\":\"kube-
apiserver/v1.24.16 (linux/amd64) kubernetes/af930c1\",\"objectRef\":{\"resource
\":\"podsecuritypolicies\",\"apiGroup\":\"policy\",\"apiVersion\":\"v1beta1\"},
\"responseStatus\":{\"metadata\":{},\"code\":200},\"requestReceivedTimestamp\":
\"2023-10-04T12:36:11.849075Z\",\"stageTimestamp\":\"2023-10-04T12:45:30.850483Z\",
\"annotations\":{\"authorization.k8s.io/decision\":\"allow\",\"authorization.k8s.io/
reason\":\"\",\"k8s.io/deprecated\":\"true\",\"k8s.io/removed-release\":\"1.25\"}}"
            },
[...]
```


Kubernetes 워크로드를 업데이트합니다. kubectl-convert를 사용하여 매니페스트 업데이트

업데이트해야 하는 워크로드 및 매니페스트를 식별한 후에는 매니페스트 파일의 리소스 유형(예: PodSecurityPolicies를 PodSecurityStandards로)을 변경해야 할 수 있습니다. 이를 위해서는 교체되는 리소스에 따라 리소스 사양 및 추가 조사를 업데이트해야 합니다.

리소스 유형이 동일하게 유지되지만 API 버전을 업데이트해야 하는 경우 `kubectl-convert` 명령을 사용하여 매니페스트 파일을 자동으로 변환할 수 있습니다. 예를 들어 이전 배포를 로 변환합니다`apps/v1`. 자세한 내용은 Kubernetes 웹 사이트의 kubectl 변환 플러그인 설치를 참조하세요.

```
kubectl-convert -f <file> --output-version <group>/<version>
```

데이터 영역이 업그레이드되는 동안 워크로드의 가용성을 보장하기 위해 PodDisruptionBudgets 및 topologySpreadConstraints 구성

데이터 영역이 업그레이드되는 동안 워크로드의 가용성을 보장하기 위해 워크로드에 적절한 PodDisruptionBudgets 및 topologySpreadConstraints가 있는지 확인합니다. 모든 워크로드에 동일한 수준의 가용성이 필요한 것은 아니므로 워크로드의 규모와 요구 사항을 검증해야 합니다.

워크로드가 여러 가용 영역과 토폴로지 분산이 있는 여러 호스트에 분산되어 있는지 확인하면 워크로드가 인시던트 없이 새 데이터 영역으로 자동으로 마이그레이션된다는 높은 수준의 신뢰도를 얻을 수 있습니다.

다음은 복제본의 80%를 항상 사용할 수 있고 영역 및 호스트에 복제본을 분산하는 워크로드의 예입니다.

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: myapp
spec:
  minAvailable: "80%"
  selector:
    matchLabels:
      app: myapp
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 10
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
```

```
    - image: public.ecr.aws/eks-distro/kubernetes/pause:3.2
      name: myapp
      resources:
        requests:
          cpu: "1"
          memory: 256M
    topologySpreadConstraints:
    - labelSelector:
        matchLabels:
          app: host-zone-spread
      maxSkew: 2
      topologyKey: kubernetes.io/hostname
      whenUnsatisfiable: DoNotSchedule
    - labelSelector:
        matchLabels:
          app: host-zone-spread
      maxSkew: 2
      topologyKey: topology.kubernetes.io/zone
      whenUnsatisfiable: DoNotSchedule
```

AWS Resilience Hub는 Amazon Elastic Kubernetes Service(Amazon EKS)를 지원되는 리소스로 추가했습니다. Resilience Hub는 애플리케이션의 복원력을 정의, 검증 및 추적할 수 있는 단일 위치를 제공하므로 소프트웨어, 인프라 또는 운영 중단으로 인한 불필요한 가동 중지를 방지할 수 있습니다.

관리형 노드 그룹 또는 Karpenter를 사용하여 데이터 영역 업그레이드 간소화

관리형 노드 그룹과 Karpenter는 모두 노드 업그레이드를 단순화하지만 다른 접근 방식을 취합니다.

관리형 노드 그룹은 노드의 프로비저닝 및 수명 주기 관리를 자동화합니다. 즉, 단일 작업으로 노드를 생성, 자동 업데이트 또는 종료할 수 있습니다.

기본 구성에서 Karpenter는 호환되는 최신 EKS Optimized AMI를 사용하여 새 노드를 자동으로 생성합니다. EKS가 업데이트된 EKS 최적화 AMIs 릴리스하거나 클러스터가 업그레이드되면 Karpenter는 이러한 이미지를 자동으로 사용하기 시작합니다. Karpenter는 노드 만료를 구현하여 노드를 업데이트합니다.

Karpenter는 사용자 지정 AMIs. Karpenter에서 사용자 지정 AMIs 사용하는 경우 kubelet 버전에 대한 책임은 사용자에게 있습니다.

기존 노드 및 컨트롤 플레인과의 버전 호환성 확인

Amazon EKS에서 Kubernetes 업그레이드를 진행하기 전에 관리형 노드 그룹, 자체 관리형 노드 및 컨트롤 플레인 간의 호환성을 보장하는 것이 중요합니다. 호환성은 사용 중인 Kubernetes 버전에 따라 결정되며 시나리오에 따라 다릅니다. 전술:

- Kubernetes v1.28+ - ** Kubernetes 버전 1.28 이상부터 코어 구성 요소에 대한 보다 관대한 버전 정책이 있습니다. 특히 Kubernetes API 서버와 kubelet 간에 지원되는 스큐는 n-2에서 n-3으로 마이너 버전 1개로 확장되었습니다. 예를 들어 EKS 컨트롤 플레인 버전이 1.28인 경우 kubelet 버전을 1.25 이전 버전으로 안전하게 사용할 수 있습니다. 이 버전 스큐는 AWS Fargate, 관리형 노드 그룹 및 자체 관리형 노드에서 지원됩니다. 보안상의 이유로 Amazon Machine Image(AMI) 버전을 up-to-date 유지하는 것이 좋습니다. 이전 kubelet 버전은 이전 kubelet 버전 사용의 이점을 능가할 수 있는 잠재적 일반적인 취약성 및 노출(CVEs)로 인해 보안 위험을 초래할 수 있습니다.
- Kubernetes < v1.28 - v1.28 이전 버전을 사용하는 경우 API 서버와 kubelet 간에 지원되는 스큐는 n-2입니다. 예를 들어 EKS 버전이 1.27인 경우 사용할 수 있는 가장 오래된 kubelet 버전은 1.25입니다. 이 버전 스큐는 AWS Fargate, 관리형 노드 그룹 및 자체 관리형 노드에 적용됩니다.

Karpenter 관리형 노드에 대한 노드 만료 활성화

Karpenter가 노드 업그레이드를 구현하는 한 가지 방법은 노드 만료 개념을 사용하는 것입니다. 이렇게 하면 노드 업그레이드에 필요한 계획이 줄어듭니다. 프로비저너에서 ttlSecondsUntilExpired 값을 설정하면 노드 만료가 활성화됩니다. 노드가 몇 초 안에 정의된 수명에 도달하면 안전하게 드레이닝되고 삭제됩니다. 이는 사용 중인 경우에도 마찬가지이므로 노드를 새로 프로비저닝된 업그레이드된 인스턴스로 바꿀 수 있습니다. 노드가 교체되면 Karpenter는 최신 EKS 최적화 AMIs 사용합니다. 자세한 내용은 Karpenter 웹 사이트의 프로비저닝 해제를 참조하세요.

Karpenter는이 값에 지터를 자동으로 추가하지 않습니다. 과도한 워크로드 중단을 방지하려면 Kubernetes 설명서에 표시된 대로 포드 중단 예산을 정의합니다.

프로비저너에서 ttlSecondsUntilExpired를 구성하는 경우 이는 프로비저너와 연결된 기존 노드에 적용됩니다.

Karpenter 관리형 노드에 드리프트 기능 사용

Karpenter의 드리프트 기능은 Karpenter 프로비저닝 노드를 자동으로 업그레이드하여 EKS 컨트롤 플레인과 동기화 상태를 유지할 수 있습니다. Karpenter Drift는 현재 특성 게이트를 사용하여 활성화해야

합니다. Karpenter의 기본 구성은 EKS 클러스터의 컨트롤 플레인과 동일한 메이저 및 마이너 버전에 최신 EKS 최적화 AMI를 사용합니다.

EKS 클러스터 업그레이드가 완료되면 Karpenter의 드리프트 기능은 Karpenter 프로비저닝 노드가 이전 클러스터 버전에 EKS 최적화 AMIs를 사용하고 있음을 감지하고 해당 노드를 자동으로 코딩, 드레이닝 및 교체합니다. 새 노드로 이동하는 포드를 지원하려면 적절한 포드 리소스 할당량을 설정하고 포드 중단 예산(PDB)을 사용하여 Kubernetes 모범 사례를 따르세요. Karpenter의 프로비저닝 해제는 포드 리소스 요청에 따라 대체 노드를 사전 구동하고 노드 프로비저닝 해제 시 PDBs를 준수합니다.

eksctl을 사용하여 자체 관리형 노드 그룹의 업그레이드 자동화

자체 관리형 노드 그룹은 계정에 배포되고 EKS 서비스 외부의 클러스터에 연결된 EC2 인스턴스입니다. 이는 일반적으로 일종의 자동화 도구에 의해 배포되고 관리됩니다. 자체 관리형 노드 그룹을 업그레이드하려면 도구 설명서를 참조해야 합니다.

예를 들어 eksctl은 자체 관리형 노드 삭제 및 드레이닝을 지원합니다.

몇 가지 일반적인 도구는 다음과 같습니다.

- eksctl
- kOps
- EKS 블루프린트

업그레이드하기 전에 클러스터 백업

새로운 버전의 Kubernetes는 Amazon EKS 클러스터에 상당한 변경 사항을 도입합니다. 클러스터를 업그레이드한 후에는 다운그레이드할 수 없습니다.

Velero는 기존 클러스터를 백업하고 새 클러스터에 백업을 적용하는 데 사용할 수 있는 커뮤니티 지원 오픈 소스 도구입니다.

현재 EKS에서 지원하는 Kubernetes 버전에 대해서만 새 클러스터를 생성할 수 있습니다. 클러스터가 현재 실행 중인 버전이 여전히 지원되고 업그레이드에 실패하는 경우 원래 버전으로 새 클러스터를 생성하고 데이터 영역을 복원할 수 있습니다. IAM을 포함한 AWS 리소스는 Velero의 백업에 포함되지 않습니다. 이러한 리소스는 다시 생성해야 합니다.

컨트롤 플레인 업그레이드 후 Fargate 배포 다시 시작

Fargate 데이터 영역 노드를 업그레이드하려면 워크로드를 재배포해야 합니다. `-o wide` 옵션으로 모든 포드를 나열하여 Fargate 노드에서 실행 중인 워크로드를 식별할 수 있습니다. 로 시작하는 모든 노드 이름은 클러스터에 재배포해야 `fargate-` 합니다.

인플레이스 클러스터 업그레이드의 대안으로 블루/그린 클러스터 평가

일부 고객은 블루/그린 업그레이드 전략을 선호합니다. 여기에는 이점이 있을 수 있지만 고려해야 할 단점도 포함됩니다.

이점은 다음과 같습니다.

- 여러 EKS 버전을 한 번에 변경할 수 있음(예: 1.23에서 1.25로)
- 이전 클러스터로 다시 전환할 수 있음
- 최신 시스템(예: terraform)으로 관리할 수 있는 새 클러스터를 생성합니다.
- 워크로드를 개별적으로 마이그레이션할 수 있습니다.

몇 가지 단점은 다음과 같습니다.

- 소비자 업데이트가 필요한 API 엔드포인트 및 OIDC 변경(예: kubectl 및 CI/CD)
- 마이그레이션 중에 클러스터 2개를 병렬로 실행해야 하므로 비용이 많이 들고 리전 용량이 제한될 수 있습니다.
- 워크로드가 함께 마이그레이션하기 위해 서로 의존하는 경우 더 많은 조정이 필요합니다.
- 로드 밸런서와 외부 DNS는 여러 클러스터에 쉽게 적용할 수 없습니다.

이 전략은 가능하지만 현재 위치 업그레이드보다 비용이 많이 들고 조정 및 워크로드 마이그레이션에 더 많은 시간이 필요합니다. 경우에 따라 필요할 수 있으므로 신중하게 계획해야 합니다.

GitOps와 같은 높은 수준의 자동화 및 선언적 시스템을 사용하면 더 쉽게 수행할 수 있습니다. 데이터가 백업되고 새 클러스터로 마이그레이션되도록 상태 저장 워크로드에 대한 추가 예방 조치를 취해야 합니다.

자세한 내용은 다음 블로그 게시물을 검토하세요.

- Kubernetes 클러스터 업그레이드: 블루-그린 배포 전략
- 상태 비저장 ArgoCD 워크로드를 위한 블루/그린 또는 카나리아 Amazon EKS 클러스터 마이그레이션

Kubernetes 프로젝트에서 계획된 주요 변경 사항 추적 - 미리 생각하기

다음 버전만 보지 마세요. 새 버전의 Kubernetes가 릴리스되면 검토하고 주요 변경 사항을 식별합니다. 예를 들어 일부 애플리케이션은 Docker API를 직접 사용했으며, Kubernetes에서 Docker용 컨테이너 런타임 인터페이스(CRI)(Dockershim이라고도 함)에 대한 지원이 제거되었습니다`1.24`. 이러한 종류의 변화에 대비하려면 더 많은 시간이 필요합니다.

업그레이드하려는 버전에 대해 문서화된 모든 변경 사항을 검토하고 필요한 업그레이드 단계를 기록해 둡니다. 또한 Amazon EKS 관리형 클러스터와 관련된 요구 사항 또는 절차를 기록해 둡니다.

- Kubernetes 변경 로그

기능 제거에 대한 특정 지침

1.25에서 Dockershim 제거 - Docker 소켓용 감지기(DDS) 사용

1.25용 EKS 최적화 AMI에는 더 이상 Dockershim에 대한 지원이 포함되지 않습니다. Docker 소켓을 탑재하는 등 Dockershim에 대한 종속성이 있는 경우 작업자 노드를 1.25로 업그레이드하기 전에 해당 종속성을 제거해야 합니다.

1.25로 업그레이드하기 전에 Docker 소켓에 종속된 인스턴스를 찾습니다. kubectl 플러그인인 Docker Socket(DDS)용 감지기를 사용하는 것이 좋습니다.

1.25에서 PodSecurityPolicy 제거 - 포드 보안 표준 또는 policy-as-code 솔루션으로 마이그레이션

`PodSecurityPolicy`는 Kubernetes 1.21에서 더 이상 사용되지 않으며 Kubernetes 1.25에서 제거되었습니다. 클러스터에서 PodSecurityPolicy를 사용하는 경우 워크로드 중단을 방지하기 위해 클러스터를 버전 1.25로 업그레이드하기 전에 기본 제공 Kubernetes 포드 보안 표준(PSS) 또는 policy-as-code 솔루션으로 마이그레이션해야 합니다.

AWS는 EKS 설명서에 자세한 FAQ를 게시했습니다.

포드 보안 표준(PSS) 및 포드 보안 승인(PSA) 모범 사례를 검토합니다.

Kubernetes 웹 사이트에서 PodSecurityPolicy 사용 중단 블로그 게시물을 검토합니다.

1.23에서 인트리 스토리지 드라이버 사용 중단 - 컨테이너 스토리지 인터페이스(CSI) 드라이버로 마이그레이션

컨테이너 스토리지 인터페이스(CSI)는 Kubernetes가 기존 트리 내 스토리지 드라이버 메커니즘을 교체할 수 있도록 설계되었습니다. Amazon EBS 컨테이너 스토리지 인터페이스(CSI) 마이그레이션 기능은 기본적으로 Amazon EKS `1.23` 이후 클러스터에 활성화되어 있습니다. 버전 `1.22` 또는 이전 클러스터에서 실행 중인 포드가 있는 경우 서비스 중단을 방지하기 위해 클러스터를 버전으로 업데이트하기 전에 Amazon EBS CSI 드라이버`1.23`를 설치해야 합니다.

Amazon EBS CSI 마이그레이션 자주 묻는 질문을 검토합니다.

추가 리소스

ClowdHaus EKS 업그레이드 지침

ClowdHaus EKS 업그레이드 지침은 Amazon EKS 클러스터를 업그레이드하는 데 도움이 되는 CLI입니다. 업그레이드 전에 해결해야 할 잠재적 문제가 있는지 클러스터를 분석할 수 있습니다.

GoNoGo

GoNoGo는 클러스터 추가 기능의 업그레이드 신뢰도를 결정하는 알파 단계 도구입니다.

비용 최적화 모범 사례

Cost Optimization은 최저 가격대로 비즈니스 성과를 달성하는 것입니다. 이 가이드의 설명서에 따라 Amazon EKS 워크로드를 최적화합니다.

일반 가이드라인

클라우드에는 마이크로서비스의 비용 최적화를 달성하는 데 도움이 되는 여러 가지 일반 지침이 있습니다.

- Amazon EKS에서 실행되는 워크로드가 컨테이너를 실행하기 위한 특정 인프라 유형과 독립적이어야 합니다. 이렇게 하면 비용이 가장 적게 드는 인프라 유형에서 실행할 때 유연성이 향상됩니다. Amazon EKS를 EC2와 함께 사용하는 동안 워크로드의 특성으로 인해 GPU 또는 기타 인스턴스 유형과 같이 특정 유형의 EC2 인스턴스 유형이 필요한 워크로드가 있는 경우 예외가 있을 수 있습니다.
- 최적으로 프로파일링된 컨테이너 인스턴스 선택 - Amazon CloudWatch Container Insights for Amazon EKS 또는 Kubernetes 에코시스템에서 사용할 수 있는 타사 도구와 같은 서비스를 사용하여 프로덕션 또는 사전 프로덕션 환경을 프로파일링하고 CPU 및 메모리와 같은 중요한 지표를 모니터링합니다. 이렇게 하면 적절한 양의 리소스를 할당하고 리소스 낭비를 방지할 수 있습니다.
- 온디맨드, 스팟 및 Savings Plan과 같이 EC2로 EKS를 실행하기 위해 AWS에서 사용할 수 있는 다양한 구매 옵션을 활용합니다.

EKS 비용 최적화 모범 사례

클라우드에서 비용 최적화를 위한 세 가지 일반적인 모범 사례 영역이 있습니다.

- 비용 효율적인 리소스(Auto Scaling, Down Scaling, 정책 및 구매 옵션)
- 지출 인식(AWS 및 타사 도구 사용)
- 시간 경과에 따른 최적화(오른쪽 크기 조정)

모든 지침과 마찬가지로 장단점이 있습니다. 조직과 협력하여이 워크로드의 우선 순위와 가장 중요한 모범 사례를 이해해야 합니다.

이 설명서의 사용법

이 가이드는 EKS 클러스터와 지원하는 워크로드를 구현하고 관리하는 데 책임이 있는 팀을 위해 작성되었습니다. 이 가이드는 사용하기 쉽도록 다양한 모범 사례 영역으로 구성되어 있습니다. 각 주제에는 EKS 클러스터의 비용 최적화에 사용할 권장 사항, 도구 및 모범 사례 목록이 있습니다. 주제는 특정 순서로 읽을 필요가 없습니다.

주요 AWS 서비스 및 Kubernetes 기능

비용 최적화는 다음 AWS 서비스 및 기능에서 지원됩니다.

- EC2 인스턴스 유형, Savings Plan(및 예약 인스턴스) 및 스팟 인스턴스를 다양한 가격으로 제공합니다.
- Kubernetes 기본 Auto Scaling 정책과 함께 Auto Scaling. 예측 가능한 워크로드Savings Plan(이전 예약 인스턴스)을 고려하세요. 애플리케이션 데이터의 탄력성과 내구성을 위해 EBS 및 EFS와 같은 관리형 데이터 스토어를 사용합니다.
- Billing and Cost Management 콘솔 대시보드와 AWS Cost Explorer는 AWS 사용량에 대한 개요를 제공합니다. 세분화된 결제 세부 정보는 AWS Organizations를 사용합니다. 여러 타사 도구에 대한 세부 정보도 공유되었습니다.
- Amazon CloudWatch 컨테이너 지표는 EKS 클러스터의 리소스 사용에 대한 지표를 제공합니다. Kubernetes 대시보드 외에도 Kubernetes 에코시스템에는 낭비를 줄이는 데 사용할 수 있는 여러 도구가 있습니다.

이 안내서에는 Amazon EKS 클러스터의 비용 최적화를 개선하는 데 사용할 수 있는 권장 사항 세트가 포함되어 있습니다.

Feedback

이 가이드는 더 광범위한 EKS/Kubernetes 커뮤니티에서 직접적인 피드백과 제안을 수집하기 위해 GitHub에서 릴리스됩니다. 가이드에 포함해야 한다고 생각되는 모범 사례가 있는 경우 GitHub 리포지토리에 문제를 제기하거나 PR을 제출하십시오. 새로운 기능이 서비스에 추가되거나 새로운 모범 사례가 발전할 때 가이드를 정기적으로 업데이트하는 것이 목적입니다.

비용 최적화 프레임워크

AWS Cloud Economics는 고객이 Amazon EKS와 같은 최신 컴퓨팅 기술을 채택하여 효율성을 높이고 비용을 절감할 수 있도록 지원하는 분야입니다. 이 원칙에서는 다음 4가지 원칙으로 구성된 "Cloud Financial Management(CFM) 프레임워크"라는 방법론을 따를 것을 권장합니다.



See 원칙: 측정 및 책임

See 원칙은 클라우드 지출에 대한 책임을 측정, 모니터링 및 생성하는 방법을 정의하는 일련의 기본 활동 및 기술입니다. "관찰성", "계측" 또는 "텔레메트리"라고도 합니다. "관찰성" 인프라의 기능과 제한에 따라 최적화할 수 있는 것이 결정됩니다. 어디에서 시작하는지 알아야 하므로 비용을 명확하게 파악하는 것은 비용 최적화의 중요한 첫 단계입니다. 이러한 유형의 가시성은 환경을 추가로 최적화하기 위해 수행해야 하는 활동 유형도 안내합니다.

다음은 See 원칙의 모범 사례에 대한 간략한 개요입니다.

- 워크로드에 대한 태그 지정 전략을 정의하고 유지 관리합니다.
  - 인스턴스 태그 지정, EKS 클러스터 태그 지정을 사용하면 개별 클러스터 비용을 확인하고 비용 및 사용 보고서에서 할당할 수 있습니다.
- Kubecost와 같은 기술을 사용하여 EKS 사용량에 대한 보고 및 모니터링을 설정합니다.
  - 리소스에 적절한 태그를 지정하고 시각화를 사용하여 비용을 측정하고 추정할 수 있도록 Cloud Intelligence Dashboards를 활성화합니다.
- 애플리케이션, LoBs of Business) 및 수익 스트림에 클라우드 비용을 할당합니다.

- 비즈니스 이해관계자와 함께 효율성/가치 KPIs. 예를 들어 트랜잭션당 비용을 측정하는 "단위 지표" KPI를 생성합니다. 예를 들어 승차 공유 서비스는 "승차당 비용"에 대한 KPI를 가질 수 있습니다.

이 원칙과 관련된 권장 기술 및 활동에 대한 자세한 내용은이 가이드의 비용 최적화 - 관찰성 섹션을 참조하세요.

절감 원칙: 비용 최적화

이 원칙은 "See" 원칙에서 개발된 기술과 기능을 기반으로 합니다. 다음 활동은 일반적으로이 원칙에 속합니다.

- 환경에서 낭비를 식별하고 제거합니다.
- 비용 효율성을 위한 설계 및 설계.
- 온디맨드 인스턴스와 스팟 인스턴스 등 가장 적합한 구매 옵션을 선택합니다.
- 서비스가 발전함에 따라 적응: AWS 서비스가 발전함에 따라 해당 서비스를 효율적으로 사용하는 방법이 변경될 수 있습니다. 이러한 변경 사항을 고려하여 조정할 의향이 있어야 합니다.

이러한 활동은 운영되므로 환경의 특성에 따라 크게 달라집니다. 비용의 주요 동인은 무엇인지 자문해 보세요. 다양한 환경이 제공하는 비즈니스 가치는 무엇입니까? 인스턴스 패밀리 유형과 같은 구매 옵션 및 인프라 선택 사항은 각 환경에 가장 적합합니까?

다음은 EKS 클러스터의 가장 일반적인 비용 동인의 우선 순위 목록입니다.

1. 컴퓨팅 비용: 여러 유형의 인스턴스 패밀리, 구매 옵션, 확장성과 가용성의 균형을 결합하려면 신중하게 고려해야 합니다. 자세한 내용은이 가이드의 비용 최적화 - 컴퓨팅 섹션의 권장 사항을 참조하세요.
2. 네트워킹 비용: EKS 클러스터에 3AZs를 사용하면 잠재적으로 AZ 간 트래픽 비용이 증가할 수 있습니다. HA 요구 사항과 네트워크 트래픽 비용 절감의 균형을 맞추는 방법에 대한 권장 사항은이 가이드의 비용 최적화 - 네트워킹 섹션을 참조하세요.
3. 스토리지 비용: EKS 클러스터에서 워크로드의 상태 저장/상태 비저장 특성과 다양한 스토리지 유형이 사용되는 방식에 따라 스토리지를 워크로드의 일부로 고려할 수 있습니다. EKS 스토리지 비용과 관련된 고려 사항은이 가이드의 비용 최적화 - 스토리지 섹션을 참조하세요.

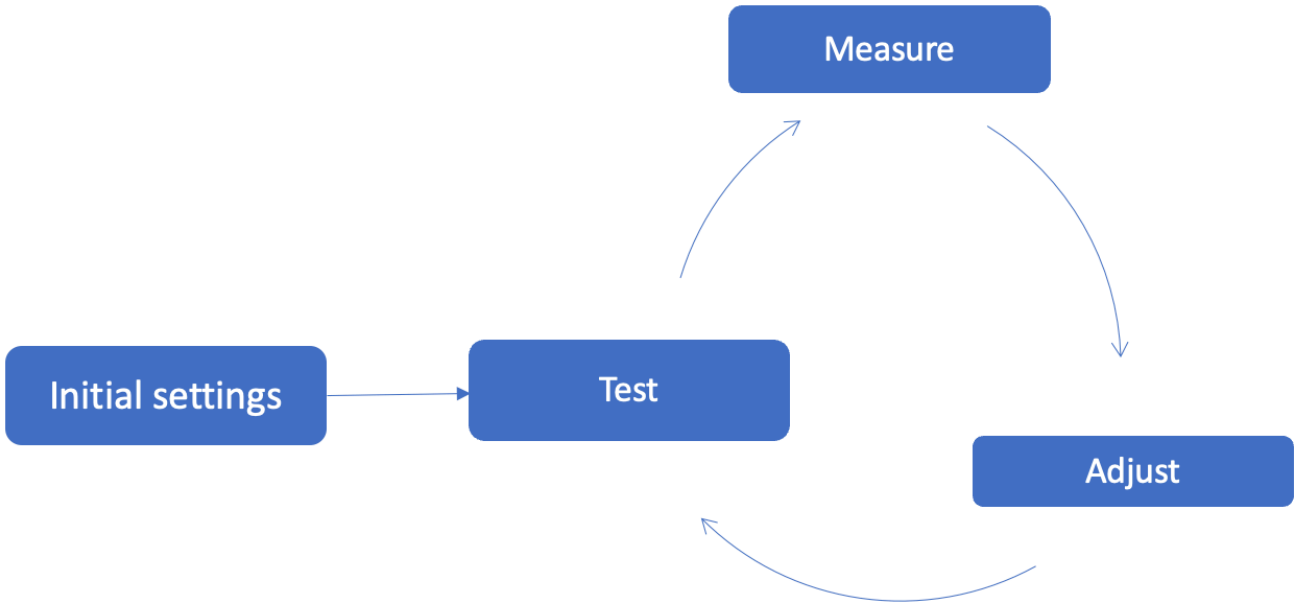
계획 원칙: 계획 및 예측

See 원칙의 권장 사항이 구현되면 클러스터가 지속적으로 최적화됩니다. 클러스터를 효율적으로 운영하면서 경험을 쌓으면 계획 및 예측 활동에 집중할 수 있습니다.

- 클라우드 비용을 동적으로 예산 책정 및 예측합니다.
- EKS 컨테이너 서비스에서 제공하는 비즈니스 가치 정량화.
- EKS 클러스터 비용 관리를 IT 재무 관리 계획과 통합합니다.

실행 원칙

비용 최적화는 지속적인 프로세스이며 점진적 개선의 플라이 휠을 포함합니다.



이러한 유형의 활동에 대한 경영진 후원을 확보하는 것은 EKS 클러스터 최적화를 조직의 "FinOps" 노력에 통합하는 데 매우 중요합니다. 이를 통해 이해관계자는 EKS 클러스터 비용에 대한 이해를 공유하고, EKS 클러스터 비용 가드레일을 구현하고, 도구, 자동화 및 활동이 조직의 필요에 따라 발전하도록 할 수 있습니다.

참조

- AWS Cloud Economics, 클라우드 재무 관리

지출 인식

지출 인식은 EKS 클러스터에서 누가, 어디서, 무엇이 지출을 유발하는지 이해하는 것입니다. 이 데이터를 정확하게 파악하면 지출에 대한 인식을 높이고 해결해야 할 영역을 강조하는 데 도움이 됩니다.

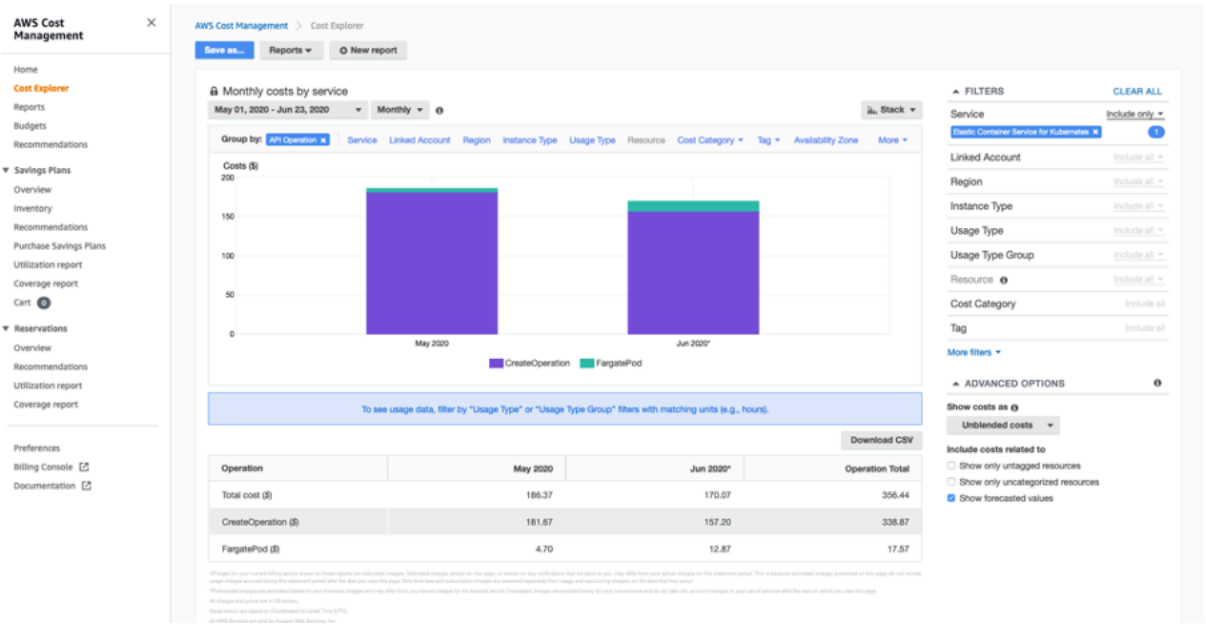
권장 사항

Cost Explorer 사용

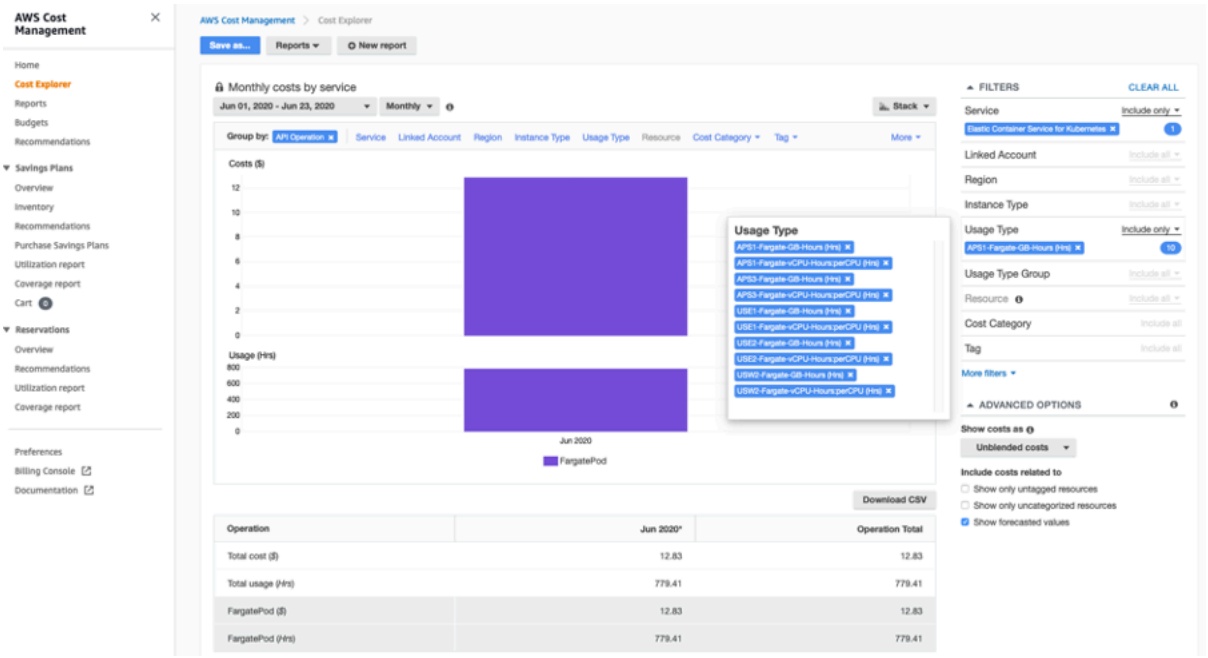
AWS Cost Explorer에는 시간이 지남에 따라 AWS 비용 및 사용량을 시각화, 이해 및 관리할 수 있는 easy-to-use 인터페이스가 있습니다. Cost Explorer에서 사용할 수 있는 필터를 사용하여 다양한 수준에서 비용 및 사용량 데이터를 분석할 수 있습니다.

EKS 컨트롤 플레인 및 EKS Fargate 비용

아래 다이어그램과 같이 필터를 사용하여 컨트롤 플레인 및 Fargate 포드에서 EKS 비용에 대해 발생한 비용을 쿼리할 수 있습니다.



필터를 사용하여 아래 다이어그램과 같이 CPU당 vCPU 시간 및 GB 시간 모두를 포함하는 EKS의 리전에서 Fargate 포드에 대해 발생한 총 비용을 쿼리할 수 있습니다.



리소스 태그 지정

Amazon EKS는 Amazon EKS 클러스터에 AWS 태그 추가를 지원합니다. 이렇게 하면 클러스터 관리를 위해 EKS API에 대한 액세스를 쉽게 제어할 수 있습니다. EKS 클러스터에 추가된 태그는 AWS EKS 클러스터 리소스에 고유하며 EC2 인스턴스 또는 로드 밸런서와 같이 클러스터에서 사용하는 다른 AWS 리소스로 전파되지 않습니다. 현재 클러스터 태그 지정은 AWS API, 콘솔 및 SDKs.

AWS Fargate는 컨테이너에 적합한 크기의 온디맨드 컴퓨팅 용량을 제공하는 기술입니다. 클러스터의 Fargate에 포드를 예약하려면 먼저 포드가 시작될 때 Fargate를 사용할 포드를 지정하는 Fargate 프로필을 하나 이상 정의해야 합니다.

EKS 클러스터에 태그 추가 및 나열:

```
$ aws eks tag-resource --resource-arn arn:aws:eks:us-west-2:xxx:cluster/ekscluster1 --
tags team=devops,env=staging,bu=cio,costcenter=1234
$ aws eks list-tags-for-resource --resource-arn arn:aws:eks:us-west-2:xxx:cluster/
ekscluster1
{
    "tags": {
        "bu": "cio",
        "env": "staging",
        "costcenter": "1234",
        "team": "devops"
    }
}
```

AWS Cost Explorer에서 비용 할당 태그를 활성화하면 AWS는 비용 할당 태그를 사용하여 비용 할당 보고서에 리소스 비용을 구성하므로 AWS 비용을 쉽게 분류하고 추적할 수 있습니다.

태그는 Amazon EKS에는 아무런 의미가 없으며 엄격하게 문자열로 해석됩니다. 예를 들어, Amazon EKS 클러스터에 태그 집합을 정의하면 클러스터의 소유자 및 스택 수준을 추적하는 데 도움이 됩니다.

AWS Trusted Advisor 사용

AWS Trusted Advisor는 비용 최적화, 보안, 내결함성, 성능 및 서비스 제한의 5가지 범주에 걸쳐 다양한 모범 사례 확인 및 권장 사항을 제공합니다.

비용 최적화의 경우 Trusted Advisor는 미사용 및 유휴 리소스를 제거하는 데 도움이 되며 예약된 용량을 약정할 것을 권장합니다. Amazon EKS에 도움이 되는 주요 작업 항목은 사용률이 낮은 EC2 인스턴스, 연결되지 않은 탄력적 IP 주소, 유휴 로드 밸런서, 사용률이 낮은 EBS 볼륨 등입니다. 검사의 전체 목록은 https://aws.amazon.com/premiumsupport/technology/trusted-advisor/best-practice-checklist/ 제공됩니다.

또한 Trusted Advisor는 할인 요금에 대한 대가로 일관된 사용량을 약정할 수 있도록 EC2 인스턴스 및 Fargate에 대한 Savings Plans 및 예약 인스턴스 권장 사항을 제공합니다.

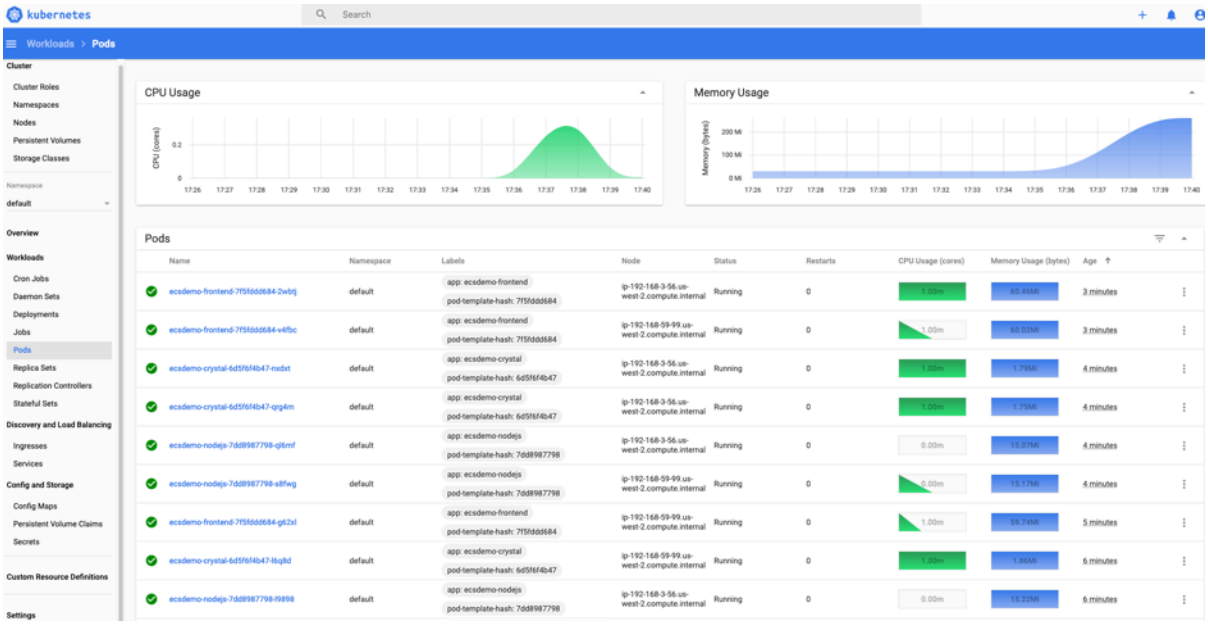
> **Note**
>
> Trusted Advisor의 권장 사항은 일반적인 권장 사항이며 EKS에만 국한되지 않습니다.

Kubernetes 대시보드 사용

Kubernetes 대시보드

Kubernetes 대시보드는 Kubernetes 클러스터용 범용 웹 기반 UI로, 클러스터, 노드 및 포드 수준의 리소스 사용량을 포함하여 Kubernetes 클러스터에 대한 정보를 제공합니다. Amazon EKS 클러스터에 Kubernetes 대시보드를 배포하는 방법은 Amazon EKS 설명서에 설명되어 있습니다.

대시보드는 각 노드 및 포드에 대한 리소스 사용 내역과 포드, 서비스, 배포 및 기타 Kubernetes 객체에 대한 자세한 메타데이터를 제공합니다. 이 통합 정보는 Kubernetes 환경에 대한 가시성을 제공합니다.



kubectl 상단 및 설명 명령

kubectl top 및 kubectl을 사용하여 리소스 사용량 지표를 보면 명령을 설명합니다. kubectl top은 클러스터 전체의 포드 또는 노드 또는 특정 포드 또는 노드에 대한 현재 CPU 및 메모리 사용량을 표시합니다. kubectl describe 명령은 특정 노드 또는 포드에 대한 자세한 정보를 제공합니다.

```
$ kubectl top pods
$ kubectl top nodes
$ kubectl top pod pod-name --namespace mynamespace --containers
```

출력에는 최상위 명령을 사용하여 노드가 사용하는 총 CPU(코어) 및 메모리(MiB) 양과 해당 숫자가 나타내는 노드의 할당 용량 백분율이 표시됩니다. 그런 다음 --containers 플래그를 추가하여 포드 내의 다음 수준인 컨테이너 수준으로 드릴다운할 수 있습니다.

```
$ kubectl describe node <node>
$ kubectl describe pod <pod>
```

kubectl describe는 각 리소스 요청 또는 제한이 나타내는 총 가용 용량의 백분율을 반환합니다.

kubectl Top and describe는 kubernetes 포드, 노드 및 컨테이너에서 CPU, 메모리 및 스토리지와 같은 중요한 리소스의 사용률과 가용성을 추적합니다. 이러한 인식은 리소스 사용량을 이해하고 비용을 관리하는 데 도움이 됩니다.

CloudWatch Container Insights 사용

CloudWatch Container Insights를 사용하여 컨테이너화된 애플리케이션 및 마이크로서비스에서 지표와 로그를 수집, 집계 및 요약할 수 있습니다. Container Insights는 EC2의 Amazon Elastic Kubernetes Service와 Amazon EC2의 Kubernetes 플랫폼에서 사용할 수 있습니다. 이 지표에는 CPU, 메모리, 디스크, 네트워크 같은 리소스 사용률이 포함되어 있습니다.

인사이트 설치는 설명서에 나와 있습니다.

CloudWatch는 클러스터, 노드, 포드, 작업 및 서비스 수준에서 집계된 지표를 CloudWatch 지표로 생성합니다.

다음 쿼리는 평균 노드 CPU 사용률을 기준으로 정렬된 노드 목록을 보여줍니다.

```
STATS avg(node_cpu_utilization) as avg_node_cpu_utilization by NodeName
| SORT avg_node_cpu_utilization DESC
```

컨테이너 이름별 CPU 사용량

```
stats pct(container_cpu_usage_total, 50) as CPUPercMedian by kubernetes.container_name
| filter Type="Container"
```

컨테이너 이름별 디스크 사용량

```
stats floor(avg(container_filesystem_usage/1024)) as container_filesystem_usage_avg_kb
 by InstanceId, kubernetes.container_name, device
| filter Type="ContainerFS"
| sort container_filesystem_usage_avg_kb desc
```

Container Insights 문서에 더 많은 샘플 쿼리가 제공됩니다.

이러한 인식은 리소스 사용량을 이해하고 비용을 관리하는 데 도움이 됩니다.

지출 인식 및 지침에 Kubecost 사용

kubecost와 같은 타사 도구를 Amazon EKS에 배포하여 Kubernetes 클러스터 실행 비용을 파악할 수도 있습니다. Kubecost를 사용하여 비용을 추적하려면이 AWS 블로그를 참조하세요.

Helm 3을 사용하여 kubecost 배포:

```
$ curl -sSL https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 |
 bash
```

```
$ helm version --short
v3.2.1+gfe51cd1
$ helm repo add stable https://kubernetes-charts.storage.googleapis.com/
$ helm repo add stable https://kubernetes-charts.storage.googleapis.com/c^C
$ kubectl create namespace kubecost
namespace/kubecost created
$ helm repo add kubecost https://kubecost.github.io/cost-analyzer/
"kubecost" has been added to your repositories

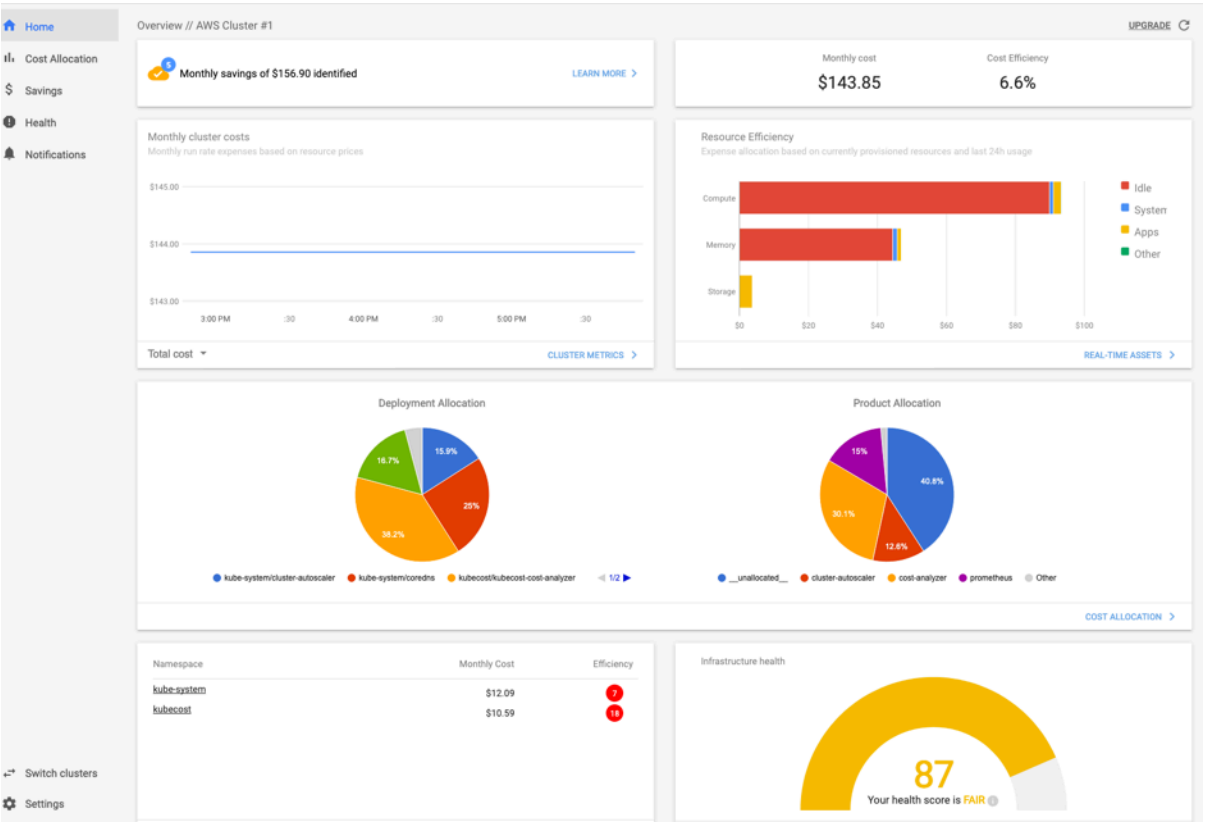
$ helm install kubecost kubecost/cost-analyzer --namespace kubecost --set
 kubecostToken="aGRoZEBqc2pzLmNvbQ==xm343yadf98"
NAME: kubecost
LAST DEPLOYED: Mon May 18 08:49:05 2020
NAMESPACE: kubecost
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
----------------------------------------------------Kubecost has been successfully
 installed. When pods are Ready, you can enable port-forwarding with the following
 command:

    kubectl port-forward --namespace kubecost deployment/kubecost-cost-analyzer 9090

Next, navigate to http://localhost:9090 in a web browser.
$ kubectl port-forward --namespace kubecost deployment/kubecost-cost-analyzer 9090

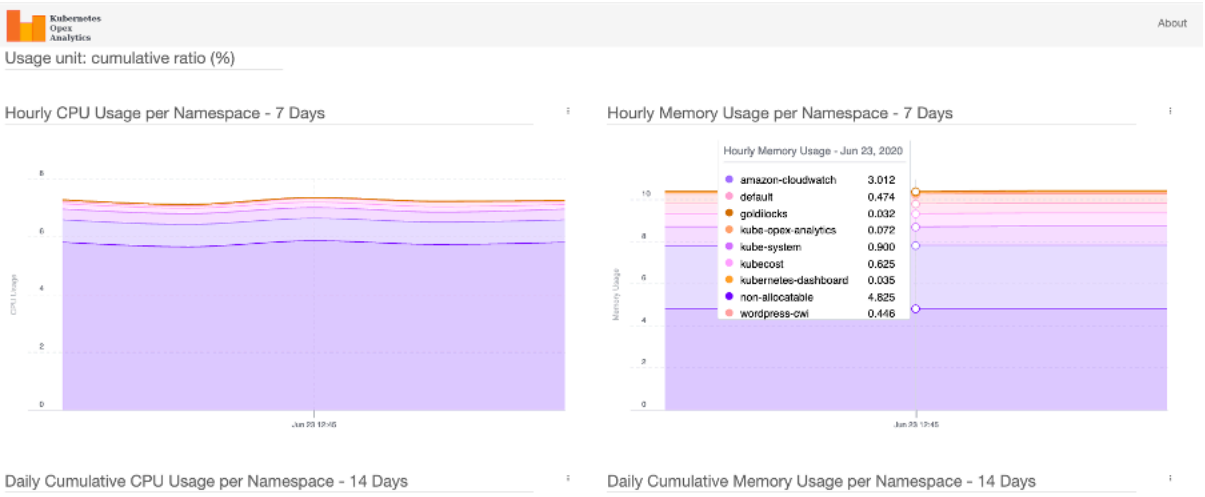
NOTE: If you are using Cloud 9 or have a need to forward it to a different port like
 8080, issue the following command
$ kubectl port-forward --namespace kubecost deployment/kubecost-cost-analyzer 8080:9090
```

Kubecost 대시보드 -



Kubernetes 비용 할당 및 용량 계획 분석 도구 사용

Kubernetes Opex Analytics는 조직이 Kubernetes 클러스터에서 사용 중인 리소스를 추적하여 초과 지불을 방지하는 데 도움이 되는 도구입니다. 이를 위해 각 프로젝트가 시간 경과에 따라 지출하는 리소스의 양에 대한 관련 인사이트를 보여주는 단기(7일), 중기(14일) 및 장기(12개월) 사용 보고서를 생성합니다.



요타스케일

Yotascale은 Kubernetes 비용을 정확하게 할당하는 데 도움이 됩니다. Yotascale Kubernetes 비용 할당 기능은 일반 시장 요금 추정 대신 예약 인스턴스 할인 및 스팟 인스턴스 요금을 포함하는 실제 비용 데이터를 활용하여 총 Kubernetes 비용 발자국을 알립니다.

자세한 내용은 웹 사이트에서 확인할 수 있습니다.

Alcide Advisor

Alcide는 AWS 파트너 네트워크(APN) 고급 기술 파트너입니다. Alcide Advisor를 사용하면 Amazon EKS 클러스터, 노드 및 포드 구성이 보안 모범 사례 및 내부 지침에 따라 실행되도록 조정할 수 있습니다. Alcide Advisor는 프로덕션으로 이동하기 전에 개발 단계를 강화하여 원활하고 안전한 DevSecOps 흐름을 보장하도록 구축된 Kubernetes 감사 및 규정 준수를 위한 에이전트 없는 서비스입니다.

자세한 내용은이 블로그 게시물에서 확인할 수 있습니다.

기타 도구

Kubernetes 가비지 수집

Kubernetes 가비지 수집기의 역할은 한 번 소유자가 있었지만 더 이상 소유자가 없는 특정 객체를 삭제하는 것입니다.

Fargate 수

Fargatecount는 AWS 고객이 사용자 지정 CloudWatch 지표를 사용하여 특정 계정의 특정 리전에서 Fargate에 배포된 총 EKS 포드 수를 추적할 수 있는 유용한 도구입니다. 이렇게 하면 EKS 클러스터에서 실행되는 모든 Fargate 포드를 추적하는 데 도움이 됩니다.

팝아이 - Kubernetes 클러스터 새니타이저

Popeye - Kubernetes 클러스터 새니타이저는 라이브 Kubernetes 클러스터를 스캔하고 배포된 리소스 및 구성과 관련된 잠재적 문제를 보고하는 유틸리티입니다. 디스크에 있는 것이 아니라 배포된 것을 기반으로 클러스터를 삭제합니다. 클러스터를 스캔하면 잘못된 구성을 감지하고 모범 사례가 적용되도록 하는 데 도움이 됩니다.

리소스

비용 최적화 모범 사례에 대해 자세히 알아보려면 다음 리소스를 참조하세요.

설명서 및 블로그

- Amazon EKS에서 태그 지정 지원

도구

- AWS Billing and Cost Management란 무엇입니까?
- Amazon CloudWatch Container Insights
- Kubecost를 사용하여 멀티 테넌트 Amazon EKS 클러스터의 비용을 추적하는 방법
- Kubecost
- Kube Opsview
- Kubernetes Opex 분석

컴퓨팅 및 오토 스케일링

개발자는 CPU 및 메모리와 같은 애플리케이션의 리소스 요구 사항을 추정하지만 지속적으로 조정하지 않으면 비용이 증가하고 성능과 신뢰성이 저하될 수 있습니다. 애플리케이션의 리소스 요구 사항을 지속적으로 조정하는 것이 처음부터 올바르게 조정하는 것보다 중요합니다.

아래에 언급된 모범 사례는 비용을 최소화하고 조직이 투자 수익을 극대화할 수 있도록 하면서 비즈니스 성과를 달성하는 비용 인식 워크로드를 구축하고 운영하는 데 도움이 됩니다. 클러스터 컴퓨팅 비용 최적화의 중요도는 다음과 같습니다.

1. 적절한 크기의 워크로드
2. 미사용 용량 감소
3. 컴퓨팅 용량 유형(예: 스팟) 및 액셀러레이터(예: GPUs) 최적화

워크로드 크기 조정

대부분의 EKS 클러스터에서 대부분의 비용은 컨테이너화된 워크로드를 실행하는 데 사용되는 EC2 인스턴스에서 발생합니다. 워크로드 요구 사항을 이해하지 않으면 컴퓨팅 리소스의 크기를 조정할 수 없습니다. 따라서 적절한 요청 및 제한을 사용하고 필요에 따라 해당 설정을 조정해야 합니다. 또한 인스턴스 크기 및 스토리지 선택과 같은 종속성은 워크로드 성능에 영향을 미쳐 비용 및 신뢰성에 의도하지 않은 다양한 결과를 초래할 수 있습니다.

요청은 실제 사용률과 일치해야 합니다. 컨테이너의 요청이 너무 높으면 총 클러스터 비용에서 큰 요인인 미사용 용량이 있습니다. 애플리케이션 및 사이드카와 같은 포드의 각 컨테이너에는 집계 포드 제한이 최대한 정확하도록 자체 요청 및 제한이 설정되어 있어야 합니다.

컨테이너에 대한 리소스 요청 및 제한을 추정하는 Goldilocks, KRR 및 Kubecost와 같은 도구를 활용합니다. 애플리케이션의 특성, 성능/비용 요구 사항 및 복잡성에 따라 확장하기에 가장 적합한 지표, 애플리케이션 성능이 저하되는 시점(포화점), 그에 따라 요청 및 제한을 조정하는 방법을 평가해야 합니다. 이 주제에 대한 추가 지침은 애플리케이션 오른쪽 크기 조정을 참조하세요.

Horizontal Pod Autoscaler(HPA)를 사용하여 실행해야 하는 애플리케이션의 복제본 수를 제어하고, Vertical Pod Autoscaler(VPA)를 사용하여 복제본당 애플리케이션 요구 사항을 조정하고 제한하며, Karpenter 또는 Cluster Autoscaler와 같은 노드 Autoscaler를 사용하여 클러스터의 총 노드 수를 지속적으로 조정하는 것이 좋습니다. Karpenter 및 Cluster Autoscaler를 사용한 비용 최적화 기법은이 문서의 뒷부분에 설명되어 있습니다.

Vertical Pod Autoscaler는 워크로드가 최적으로 실행되도록 컨테이너에 할당된 요청 및 제한을 조정할 수 있습니다. VPA가 자동으로 변경되지 않고 포드를 다시 시작하지 않도록 감사 모드에서 VPA를 실행해야 합니다. 관찰된 지표를 기반으로 변경 사항을 제안합니다. 프로덕션 워크로드에 영향을 미치는 변경 사항이 있는 경우 먼저 비프로덕션 환경에서 변경 사항을 검토하고 테스트해야 합니다. 이러한 변경 사항은 애플리케이션의 신뢰성과 성능에 영향을 미칠 수 있기 때문입니다.

소비 감소

비용을 절감하는 가장 좋은 방법은 리소스를 더 적게 프로비저닝하는 것입니다. 이를 위한 한 가지 방법은 현재 요구 사항에 따라 워크로드를 조정하는 것입니다. 워크로드가 요구 사항을 정의하고 동적으로 확장되도록 하여 비용 최적화 작업을 시작해야 합니다. 이렇게 하려면 애플리케이션에서 지표를 가져오고 `PodDisruptionBudgets` 및 포드 준비 게이트와 같은 구성을 설정하여 애플리케이션이 동적으로 안전하게 확장 및 축소될 수 있도록 해야 합니다. Cluster Autoscaler와 Karpenter는 모두 PodDisruptionBudgets를 준수하므로 제한적인 PodDisruptionBudgets. PodDisruptionBudget의 'minAvailable' 값은 항상 배포의 포드 수보다 작아야 하며 두 포드 사이에 적절한 버퍼를 유지해야 합니다. 예를 들어, 6개의 포드를 배포하여 항상 최소 4개의 포드를 실행하려면 PodDisruptionBidget의 'minAvailable'을 4로 설정합니다. 이렇게 하면 Cluster Autoscaler와 Karpenter가 노드 스케일 다운 이벤트 중에 사용률이 낮은 노드에서 포드를 안전하게 드레이닝하고 제거할 수 있습니다. Cluster Autoscaler FAQ 문서를 참조하세요.

Horizontal Pod Autoscaler는 애플리케이션의 성능 및 안정성 요구 사항을 충족하는 데 필요한 복제본 수를 조정할 수 있는 유연한 워크로드 Autoscaler입니다. CPU, 메모리 또는 대기열 깊이, 포드에 대한 연결 수 등과 같은 사용자 지정 지표와 같은 다양한 지표를 기반으로 스케일 업 및 스케일 다운 시기를 정의하는 유연한 모델이 있습니다.

Kubernetes 지표 서버는 CPU 및 메모리 사용량과 같은 기본 제공 지표에 대한 응답으로 규모 조정을 활성화하지만 Amazon CloudWatch 또는 SQS 대기열 깊이와 같은 다른 지표를 기반으로 규모 조정하려는 경우 KEDA와 같은 이벤트 기반 Auto Scaling 프로젝트를 고려해야 합니다. CloudWatch 지표와 함께 KEDA를 사용하는 방법은 이 블로그 게시물을 참조하세요. 모니터링 및 확장 기준이 되는 지표를 잘 모르는 경우 중요한 지표 모니터링에 대한 모범 사례를 확인하세요.

워크로드 소비를 줄이면 클러스터에 초과 용량이 생성되고 적절한 오토 스케일링 구성을 통해 노드를 자동으로 스케일 다운하고 총 지출을 줄일 수 있습니다. 컴퓨팅 용량을 수동으로 최적화하지 않는 것이 좋습니다. Kubernetes 스케줄러 및 노드 오토스케일러는이 프로세스를 처리하도록 설계되었습니다.

미사용 용량 감소

애플리케이션의 올바른 크기를 결정하여 초과 요청을 줄이면 프로비저닝된 컴퓨팅 용량을 줄일 수 있습니다. 위의 섹션에서 워크로드의 크기를 올바르게 조정하는 데 시간이 걸린 경우 동적으로이 작업을 수행할 수 있습니다. AWS의 Kubernetes에는 두 개의 프라이머리 노드 오토스케일러가 사용됩니다.

Karpenter 및 Cluster Autoscaler

포드가 생성되거나 제거되고 컴퓨팅 요구 사항이 변경되면 Karpenter와 Kubernetes Cluster Autoscaler 모두 클러스터의 노드 수를 조정합니다. 두 가지 모두의 기본 목표는 동일하지만 Karpenter는 노드 관리 프로비저닝 및 프로비저닝 해제에 대해 다른 접근 방식을 취하여 비용을 절감하고 클러스터 전체 사용을 최적화하는 데 도움이 될 수 있습니다.

클러스터의 크기가 커지고 워크로드가 다양해지면 노드 그룹 및 인스턴스를 사전 구성하기가 더 어려워집니다. 워크로드 요청과 마찬가지로 초기 기준을 설정하고 필요에 따라 지속적으로 조정하는 것이 중요합니다.

Cluster Autoscaler를 사용하는 경우 각 Auto Scaling 그룹(ASG)의 "최소" 및 "최대" 값을 준수하고 "원하는" 값만 조정합니다. Cluster Autoscaler는 "최소" 수를 초과하여 ASG를 스케일 다운할 수 없으므로 기본 ASG에 대해 이러한 값을 설정하는 동안 주의해야 합니다. "원하는" 수를 정상 업무 시간에 필요한 노드 수로 설정하고 "최소"를 업무 외 시간에 필요한 노드 수로 설정합니다. Cluster Autoscaler FAQ 문서를 참조하세요.

Cluster Autoscaler Priority 확장 프로그램

Kubernetes Cluster Autoscaler는 애플리케이션이 확장 및 축소됨에 따라 노드 그룹이라고 하는 노드 그룹을 확장 및 축소하여 작동합니다. 워크로드를 동적으로 조정하지 않는 경우 Cluster Autoscaler는 비용을 절감하는 데 도움이 되지 않습니다. Cluster Autoscaler를 사용하려면 클러스터 관리자가 노드 그룹을 미리 생성해야 합니다. 노드 그룹은 "profile"이 동일한 인스턴스, 즉 대략 동일한 양의 CPU와 메모리를 사용하도록 구성해야 합니다.

여러 노드 그룹을 가질 수 있으며, 우선 순위 조정 수준을 설정하도록 Cluster Autoscaler를 구성할 수 있으며, 각 노드 그룹에는 다양한 크기의 노드가 포함될 수 있습니다. 노드 그룹은 다양한 용량 유형을 가질 수 있으며 우선 순위 확장기를 사용하여 더 저렴한 그룹을 먼저 확장할 수 있습니다.

다음은 온디맨드 인스턴스를 사용하기 전에 ConfigMap`를 사용하여 예약 용량의 우선 순위를 지정하는 클러스터 구성 조각의 예입니다. 동일한 기술을 사용하여 다른 유형보다 Graviton 또는 스팟 인스턴스의 우선 순위를 지정할 수 있습니다.

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: my-cluster
managedNodeGroups:
  - name: managed-ondemand
    minSize: 1
    maxSize: 7
    instanceType: m5.xlarge
  - name: managed-reserved
    minSize: 2
    maxSize: 10
    instanceType: c5.2xlarge
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-autoscaler-priority-expander
  namespace: kube-system
data:
  priorities: |-
    10:
      - .*ondemand.*
    50:
      - .*reserved.*
```

노드 그룹을 사용하면 기본 컴퓨팅 리소스가 기본적으로 예상되는 작업을 수행하는 데 도움이 될 수 있습니다. 예를 들어 노드를 AZs에 분산하지만 모든 워크로드에 동일한 요구 사항 또는 기대치가 있는 것은 아니며 애플리케이션이 요구 사항을 명시적으로 선언하도록 하는 것이 좋습니다. Cluster Autoscaler에 대한 자세한 내용은 모범 사례 섹션을 참조하세요.

디스케줄러

Cluster Autoscaler는 예약이 필요한 새 포드 또는 사용률이 낮은 노드를 기반으로 클러스터에서 노드 용량을 추가하고 제거할 수 있습니다. 노드로 예약된 후에는 포드 배치를 전체적으로 볼 수 없습니다. Cluster Autoscaler를 사용하는 경우 클러스터의 용량이 낭비되지 않도록 Kubernetes 스케줄러도 확인해야 합니다.

클러스터에 노드가 10개 있고 각 노드가 60% 사용되는 경우 클러스터에서 프로비저닝된 용량의 40%를 사용하지 않는 것입니다. Cluster Autoscaler를 사용하면 노드당 사용률 임계값을 60%로 설정할 수 있지만 사용률이 60% 미만으로 떨어진 후에만 단일 노드를 축소하려고 시도합니다.

디스케줄러를 사용하면 포드가 예약되거나 노드가 클러스터에 추가된 후 클러스터 용량과 사용률을 확인할 수 있습니다. 클러스터의 총 용량을 지정된 임계값 이상으로 유지하려고 시도합니다. 또한 노드 테인트 또는 클러스터에 조인하는 새 노드를 기반으로 포드를 제거하여 포드가 최적의 컴퓨팅 환경에서 실행되고 있는지 확인할 수 있습니다. 디스케줄러는 제거된 포드의 교체를 예약하지 않지만 이에 대한 기본 스케줄러를 사용합니다.

Karpenter 통합

Karpenter는 노드 관리에 "그룹 없는" 접근 방식을 취합니다. 이 접근 방식은 다양한 워크로드 유형에 대해 더 유연하며 클러스터 관리자의 사전 구성이 덜 필요합니다. 그룹을 사전 정의하고 워크로드에 따라 각 그룹을 조정하는 대신 Karpenter는 프로비저너와 노드 템플릿을 사용하여 생성할 수 있는 EC2 인스턴스 유형과 인스턴스가 생성될 때 인스턴스에 대한 설정을 광범위하게 정의합니다.

빈 패킹은 더 적은 수의 최적의 크기의 인스턴스에 더 많은 워크로드를 패킹하여 인스턴스의 리소스를 더 많이 활용하는 방법입니다. 이렇게 하면 워크로드가 사용하는 리소스만 프로비저닝하여 컴퓨팅 비용을 줄이는 데 도움이 되지만 장단점이 있습니다. 특히 대규모 조정 이벤트 시 용량을 클러스터에 추가해야 하므로 새 워크로드를 시작하는 데 더 오래 걸릴 수 있습니다. 빈 패킹을 설정할 때 비용 최적화, 성능 및 가용성 간의 균형을 고려하세요.

Karpenter는 지속적으로 모니터링하고 binpack하여 인스턴스 리소스 사용률을 개선하고 컴퓨팅 비용을 절감할 수 있습니다. Karpenter는 워크로드에 대해 보다 비용 효율적인 작업자 노드를 선택할 수도 있습니다. 이는 프로비저너에서 "통합" 플래그를 true로 설정하여 달성할 수 있습니다(아래 샘플 코드 조각). 아래 예제는 통합을 활성화하는 프로비저너 예제를 보여줍니다. 이 안내서를 작성할 때 Karpenter는 실행 중인 스팟 인스턴스를 더 저렴한 스팟 인스턴스로 대체하지 않습니다. Karpenter 통합에 대한 자세한 내용은 이 블로그를 참조하세요.

```
apiVersion: karpenter.sh/v1
kind: Provisioner
```

```
metadata:
  name: enable-binpacking
spec:
  consolidation:
    enabled: true
```

체크포인트가 없는 장기 실행 배치 작업과 같이 중단이 불가능할 수 있는 워크로드의 경우 포드에 do-not-evict 주석을 추가하는 것이 좋습니다. 포드를 제거에서 옵트아웃하면 Karpenter에이 포드가 포함된 노드를 자발적으로 제거해서는 안 된다고 알립니다. 그러나 노드가 do-not-evict 드레이닝되는 동안 포드가 노드에 추가되면 나머지 포드는 여전히 제거되지만 해당 포드는 제거될 때까지 종료를 차단합니다. 어느 경우든 노드에 추가 작업이 예약되지 않도록 노드가 연결됩니다. 다음은 주석을 설정하는 방법을 보여주는 예제입니다.

```
apiVersion: v1
kind: Pod
metadata:
  name: label-demo
  labels:
    environment: production
  annotations: +
    "karpenter.sh/do-not-evict": "true"
spec:
  containers:

* name: nginx
image: nginx
ports:
 ** containerPort: 80
```

Cluster Autoscaler 파라미터를 조정하여 사용률이 낮은 노드 제거

노드 사용률은 요청된 리소스의 합계를 용량으로 나눈 값으로 정의됩니다. 기본적으로 scale-down-utilization-threshold는 50%로 설정됩니다. 이 파라미터는 및와 함께 사용할 수 scale-down-unneeded-time있습니다.이 파라미터는 노드가 스케일 다운에 적합하기 전에 필요 없는 시간을 결정합니다. 기본값은 10분입니다. 축소된 노드에서 계속 실행 중인 포드는 kube-scheduler에 의해 다른 노드에서 예약됩니다. 이러한 설정을 조정하면 사용률이 낮은 노드를 제거하는 데 도움이 될 수 있지만 클러스터가 조기에 축소되지 않도록 먼저 이러한 값을 테스트하는 것이 중요합니다.

제거 비용이 많이 드는 포드가 Cluster Autoscaler에서 인식하는 레이블로 보호되도록 하여 스케일 다운이 발생하지 않도록 할 수 있습니다. 이렇게 하려면 제거 비용이 많이 드는 포드에 주석이 있어야 합

니다`cluster-autoscaler.kubernetes.io/safe-to-evict=false`. 다음은 주석을 설정하는 yaml의 예입니다.

```
apiVersion: v1
kind: Pod
metadata:
  name: label-demo
  labels:
    environment: production
  annotations: +
    "cluster-autoscaler.kubernetes.io/safe-to-evict": "false"
spec:
  containers:

* name: nginx
image: nginx
ports:
 ** containerPort: 80
```

Cluster Autoscaler 및 Karpenter를 사용하여 노드에 태그 지정

AWS 리소스 태그는 리소스를 구성하고 AWS 비용을 세부적으로 추적하는 데 사용됩니다. 비용 추적을 위해 Kubernetes 레이블과 직접 연관되지 않습니다. Kubernetes 리소스 레이블 지정으로 시작하고 Kubecost와 같은 도구를 활용하여 포드, 네임스페이스 등의 Kubernetes 레이블을 기반으로 인프라 비용 보고를 받는 것이 좋습니다.

작업자 노드에는 AWS Cost Explorer에 결제 정보를 표시하는 태그가 있어야 합니다. Cluster Autoscaler를 사용하면 시작 템플릿을 사용하여 관리형 노드 그룹 내의 작업자 노드에 태그를 지정합니다. 자체 관리형 노드 그룹의 경우 EC2 Auto Scaling 그룹을 사용하여 인스턴스에 태그를 지정합니다. Karpenter에서 프로비저닝한 인스턴스의 경우 노드 템플릿에서 spec.tags를 사용하여 태그를 지정합니다.

다중 테넌트 클러스터

다른 팀에서 공유하는 클러스터에서 작업하는 경우 동일한 노드에서 실행되는 다른 워크로드를 볼 수 없을 수 있습니다. 리소스 요청은 CPU 공유와 같은 일부 "시끄러운 이웃" 문제를 격리하는 데 도움이 될 수 있지만 디스크 I/O 소진과 같은 모든 리소스 경계를 격리하지는 못할 수 있습니다. 워크로드별 모든 소모성 리소스가 격리되거나 제한될 수 있는 것은 아닙니다. 다른 워크로드보다 더 높은 속도로 공유 리소스를 사용하는 워크로드는 노드 테인트 및 허용 오차를 통해 격리해야 합니다. 이러한 워크로드에 대한 또 다른 고급 기법은 컨테이너에 대한 공유 CPU 대신 배타적인 CPU를 보장하는 CPU 고정입니다.

노드 수준에서 워크로드를 격리하면 비용이 더 많이 들 수 있지만 예약 인스턴스, Graviton 프로세서 또는 스팟을 사용하여 BestEffort 작업을 예약하거나 추가 비용 절감을 활용할 수 있습니다.

공유 클러스터에는 IP 소진, Kubernetes 서비스 제한 또는 API 조정 요청과 같은 클러스터 수준 리소스 제약이 있을 수도 있습니다. 확장성 모범 사례 가이드를 검토하여 클러스터가 이러한 제한을 피하도록 해야 합니다.

네임스페이스 또는 Karpenter 프로비저너 수준에서 리소스를 격리할 수 있습니다. Resource Quotas는 네임스페이스의 리소스 워크로드가 소비할 수 있는 리소스 수에 대한 제한을 설정하는 방법을 제공합니다. 이는 좋은 초기 가드레일일 수 있지만 워크로드의 규모 조정을 인위적으로 제한하지 않도록 지속적으로 평가해야 합니다.

Karpenter 프로비저너는 클러스터의 일부 소모성 리소스(예: CPU, GPU)에 대한 제한을 설정할 수 있지만 적절한 프로비저너를 사용하도록 테넌트 애플리케이션을 구성해야 합니다. 이렇게 하면 단일 프로비저너가 클러스터에 노드를 너무 많이 생성하지 못할 수 있지만, 제한이 너무 낮게 설정되지 않았는지 확인하고 워크로드가 확장되지 않도록 지속적으로 평가해야 합니다.

예약된 Autoscaling

주말과 업무 시간에 클러스터를 스케일 다운해야 할 수 있습니다. 이는 사용하지 않을 때 0으로 스케일 다운하려는 테스트 및 비프로덕션 클러스터와 특히 관련이 있습니다. 클러스터 턴다운과 같은 솔루션은 cron 일정에 따라 복제본을 0으로 축소할 수 있습니다. 다음 AWS 블로그에 설명된 Karpenter를 사용하여이 문제를 해결할 수도 있습니다.

컴퓨팅 용량 유형 최적화

클러스터의 총 컴퓨팅 용량을 최적화하고 빈 패킹을 활용한 후에는 클러스터에서 프로비저닝한 컴퓨팅 유형과 이러한 리소스에 대한 비용을 지불하는 방법을 살펴봐야 합니다. AWS에는 다음과 같은 용량 유형으로 분류할 컴퓨팅 비용을 줄일 수 있는 컴퓨팅 절감형 플랜이 있습니다.

- 스팟
- 절감형 플랜
- 온디맨드
- Fargate

각 용량 유형에는 관리 오버헤드, 가용성 및 장기 약정에 대한 장단점이 서로 다르므로 환경에 적합한 것을 결정해야 합니다. 어떤 환경도 단일 용량 유형에 의존해서는 안 되며 단일 클러스터에서 여러 실행 유형을 혼합하여 특정 워크로드 요구 사항과 비용을 최적화할 수 있습니다.

스팟

스팟 용량 유형은 가용 영역의 예비 용량에서 EC2 인스턴스를 프로비저닝합니다. 스팟은 최대 90%까지 가장 큰 할인을 제공하지만 다른 곳에서 필요할 때 해당 인스턴스를 중단할 수 있습니다. 또한 새 스팟 인스턴스를 프로비저닝할 수 있는 용량이 항상 있는 것은 아니며 2분 중단 알림으로 기존 스팟 인스턴스를 회수할 수 있습니다. 애플리케이션에 긴 시작 또는 종료 프로세스가 있는 경우 스팟 인스턴스가 최선의 옵션이 아닐 수 있습니다.

스팟 컴퓨팅은 스팟 용량을 사용할 수 없을 가능성을 줄이기 위해 다양한 인스턴스 유형을 사용해야 합니다. 노드를 안전하게 종료하려면 인스턴스 중단을 처리해야 합니다. Karpenter 또는 관리형 노드 그룹의 일부로 프로비저닝된 노드는 인스턴스 중단 알림을 자동으로 지원합니다. 자체 관리형 노드를 사용하는 경우 노드 종료 핸들러를 개별적으로 실행하여 스팟 인스턴스를 정상적으로 종료해야 합니다.

단일 클러스터에서 스팟 인스턴스와 온디맨드 인스턴스의 균형을 맞출 수 있습니다. Karpenter를 사용하면 가중 프로비저너를 생성하여 다양한 용량 유형의 균형을 맞출 수 있습니다. Cluster Autoscaler를 사용하면 스팟 및 온디맨드 또는 예약 인스턴스를 사용하여 혼합 노드 그룹을 생성할 수 있습니다.

다음은 Karpenter를 사용하여 온디맨드 인스턴스보다 먼저 스팟 인스턴스의 우선순위를 지정하는 예제입니다. 프로비저너를 생성할 때 스팟, 온디맨드 또는 둘 다(아래 참조)를 지정할 수 있습니다. 둘 다 지정하고 포드가 스팟을 사용해야 하는지 온디맨드를 사용해야 하는지 명시적으로 지정하지 않으면 Karpenter는 price-capacity-optimization 할당 전략으로 노드를 프로비저닝할 때 스팟의 우선 순위를 지정합니다.

```
apiVersion: karpenter.sh/v1
kind: Provisioner
metadata:
  name: spot-prioritized
spec:
  requirements:
    - key: "karpenter.sh/capacity-type"
      operator: In
        values: ["spot", "on-demand"]
```

Savings Plans, 예약 인스턴스 및 AWS EDP

컴퓨팅 절감형 플랜을 사용하여 컴퓨팅 지출을 줄일 수 있습니다. 절감형 플랜은 1년 또는 3년 약정의 컴퓨팅 사용량에 대해 할인된 가격을 제공합니다. 사용량은 EKS 클러스터의 EC2 인스턴스에 적용될 수 있지만 Lambda 및 Fargate와 같은 모든 컴퓨팅 사용량에도 적용됩니다. 절감형 플랜을 사용하면 비용을 절감하고 약정 기간 동안 모든 EC2 인스턴스 유형을 선택할 수 있습니다.

컴퓨팅 절감형 플랜은 사용하려는 인스턴스 유형, 패밀리 또는 리전에 대한 약정 없이 EC2 비용을 최대 66% 절감할 수 있습니다. 절감액은 인스턴스를 사용할 때 인스턴스에 자동으로 적용됩니다.

EC2 Instance Savings Plans C 패밀리의 인스턴스와 같은 특정 리전 및 EC2 패밀리에서 사용량을 약정하여 컴퓨팅 비용을 최대 72% 절감합니다. 사용량을 리전 내의 모든 AZ로 전환하고, c5 또는 c6와 같은 인스턴스 패밀리의 모든 세대를 사용하고, 패밀리 내의 모든 크기의 인스턴스를 사용할 수 있습니다. 할인은 절감형 플랜 기준과 일치하는 계정의 모든 인스턴스에 자동으로 적용됩니다.

예약 인스턴스는 EC2 인스턴스 Savings Plan과 유사하지만 온디맨드 인스턴스에 비해 가용 영역 또는 리전의 용량을 보장하고 비용을 최대 72% 절감합니다. 필요한 예약 용량을 계산한 후에는 예약 기간(1년 또는 3년)을 선택할 수 있습니다. 계정에서 해당 EC2 인스턴스를 실행하면 할인이 자동으로 적용됩니다.

고객은 AWS와의 엔터프라이즈 계약에 등록할 수도 있습니다. 엔터프라이즈 계약은 고객의 요구에 가장 적합한 계약을 맞춤 설정할 수 있는 옵션을 제공합니다. 고객은 AWS EDP(엔터프라이즈 할인 프로그램)에 따라 요금 할인을 받을 수 있습니다. 엔터프라이즈 계약에 대한 자세한 내용은 AWS 영업 담당자에게 문의하십시오.

온디맨드

온디맨드 EC2 인스턴스는 절감형 플랜에 비해 스팟과 장기 약정이 없는 경우 중단 없는 가용성의 이점을 누릴 수 있습니다. 클러스터에서 비용을 절감하려면 온디맨드 EC2 인스턴스의 사용량을 줄여야 합니다.

워크로드 요구 사항을 최적화한 후에는 클러스터의 최소 및 최대 용량을 계산할 수 있어야 합니다. 이 숫자는 시간이 지남에 따라 변경될 수 있지만 감소하는 경우는 거의 없습니다. 최소 미만의 모든 항목에 Savings Plan을 사용하고 애플리케이션 가용성에 영향을 주지 않는 용량을 확보하는 것이 좋습니다. 지속적으로 사용되지 않거나 가용성에 필요한 다른 모든 것은 온디맨드 방식으로 사용할 수 있습니다.

이 단원에서 언급했듯이 사용량을 줄이는 가장 좋은 방법은 리소스를 적게 사용하고 프로비저닝한 리소스를 최대한 활용하는 것입니다. Cluster Autoscaler를 사용하면 `scale-down-utilization-threshold` 설정으로 사용률이 낮은 노드를 제거할 수 있습니다. Karpenter에서는 통합을 활성화하는 것이 좋습니다.

워크로드에 사용할 수 있는 EC2 인스턴스 유형을 수동으로 식별하려면 각 리전에서 사용할 수 있는 인스턴스와 EKS와 호환되는 인스턴스를 표시할 수 있는 ec2-instance-selector를 사용해야 합니다. x86 프로세스 아키텍처, 4Gb 메모리, vCPUs 있고 us-east-1 리전에서 사용할 수 있는 인스턴스의 사용 예입니다.

```
ec2-instance-selector --memory 4 --vcpus 2 --cpu-architecture x86_64 \
  -r us-east-1 --service eks
c5.large
c5a.large
c5ad.large
c5d.large
c6a.large
c6i.large
t2.medium
t3.medium
t3a.medium
```

비프로덕션 환경의 경우 야간 및 주말과 같은 미사용 시간에 클러스터를 자동으로 축소할 수 있습니다. kubecost 프로젝트 클러스터 가동 중지는 설정된 일정에 따라 클러스터를 자동으로 축소할 수 있는 컨트롤러의 예입니다.

Fargate 컴퓨팅

Fargate 컴퓨팅은 EKS 클러스터를 위한 완전 관리형 컴퓨팅 옵션입니다. Kubernetes 클러스터에서 노드당 하나의 포드를 예약하여 포드 격리를 제공합니다. 이를 통해 컴퓨팅 노드의 크기를 워크로드의 CPU 및 RAM 요구 사항에 맞게 조정하여 클러스터의 워크로드 사용량을 엄격하게 제어할 수 있습니다.

Fargate는 0.5GB 메모리가 있는 .25 vCPU만큼 작은 워크로드와 120GB 메모리가 있는 16 vCPU만큼 큰 워크로드를 확장할 수 있습니다. 사용 가능한 포드 크기 변형 수에는 제한이 있으며 워크로드가 Fargate 구성에 가장 적합한 방법을 이해해야 합니다. 예를 들어 워크로드에 메모리가 0.5GB인 vCPU 1개가 필요한 경우 가장 작은 Fargate 포드는 메모리가 2GB인 vCPU 1개입니다.

Fargate는 EC2 인스턴스 또는 운영 체제 관리가 없는 등 많은 이점이 있지만 배포된 모든 포드가 클러스터에서 별도의 노드로 격리되어 있기 때문에 기존 EC2 인스턴스보다 더 많은 컴퓨팅 용량이 필요할 수 있습니다. 이렇게 하려면 일반적으로 노드에 배포하는 Kubelet, 로깅 에이전트 및 DaemonSets와 같은 사물에 대해 더 많은 중복이 필요합니다. DaemonSets는 Fargate에서 지원되지 않으므로 포드 "`sidecars"로 변환하고 애플리케이션과 함께 실행해야 합니다.

워크로드 경계가 워크로드 간에 버스트하거나 공유할 수 없는 노드이므로 Fargate는 빈 패킹 또는 CPU 오버프로비저닝의 이점을 누릴 수 없습니다. Fargate는 비용이 드는 EC2 인스턴스 관리 시간을 절약하지만 CPU 및 메모리 비용은 다른 EC2 용량 유형보다 더 높을 수 있습니다. Fargate 포드는 컴퓨팅 절감형 플랜을 활용하여 온디맨드 비용을 절감할 수 있습니다.

컴퓨팅 사용량 최적화

컴퓨팅 인프라 비용을 절감하는 또 다른 방법은 워크로드에 보다 효율적인 컴퓨팅을 사용하는 것입니다. 이는 x86보다 최대 20% 저렴하고 60% 더 에너지 효율적인 Graviton 프로세서와 같은 고성능 범용 컴퓨팅 또는 GPUs 및 FPGAs와 같은 워크로드별 액셀러레이터에서 발생할 수 있습니다. 팔 아키텍처에서 실행할 수 있는 컨테이너를 구축하고 워크로드에 적합한 액셀러레이터로 노드를 설정해야 합니다.

EKS는 혼합 아키텍처(예: amd64 및 arm64)를 사용하여 클러스터를 실행할 수 있으며 컨테이너가 여러 아키텍처에 맞게 컴파일된 경우 프로비저너에서 두 아키텍처를 모두 허용하여 Karpenter를 사용하는 Graviton 프로세서를 활용할 수 있습니다. 그러나 일관된 성능을 유지하려면 각 워크로드를 단일 컴퓨팅 아키텍처에 유지하고 사용 가능한 추가 용량이 없는 경우에만 다른 아키텍처를 사용하는 것이 좋습니다.

프로비저너는 여러 아키텍처로 구성할 수 있으며 워크로드 사양에서 특정 아키텍처를 요청할 수도 있습니다.

```
apiVersion: karpenter.sh/v1
kind: Provisioner
metadata:
  name: default
spec:
  requirements:
  - key: "kubernetes.io/arch"
    operator: In
    values: ["arm64", "amd64"]
```

Cluster Autoscaler를 사용하면 Graviton 인스턴스에 대한 노드 그룹을 생성하고 워크로드에서 노드 허용치를 설정하여 새 용량을 활용해야 합니다.

GPUs 및 FPGAs는 워크로드의 성능을 크게 높일 수 있지만 액셀러레이터를 사용하도록 워크로드를 최적화해야 합니다. 기계 학습 및 인공 지능을 위한 많은 워크로드 유형은 컴퓨팅에 GPUs 사용할 수 있으며, 인스턴스를 클러스터에 추가하고 리소스 요청을 사용하여 워크로드에 탑재할 수 있습니다.

```
spec:
  template:
    spec:
    - containers:
      ...
      resources:
```

```
          limits:
            nvidia.com/gpu: "1"
```

일부 GPU 하드웨어는 여러 워크로드에서 공유할 수 있으므로 단일 GPU를 프로비저닝하고 사용할 수 있습니다. 워크로드 GPU 공유를 구성하는 방법을 알아보려면 가상 GPU 디바이스 플러그인에서 자세한 내용을 참조하세요. 다음 블로그를 참조할 수도 있습니다.

- NVIDIA 시분할 및 가속화된 EC2 인스턴스를 사용하여 Amazon EKS에서 GPU 공유
- Amazon EKS에서 NVIDIA의 다중 인스턴스 GPU(MIG)를 사용하여 GPU 사용률 극대화: 성능 향상을 위해 GPU당 더 많은 포드 실행

비용 최적화 - 네트워킹

고가용성(HA)을 위한 아키텍처 시스템은 복원력과 내결함성을 달성하기 위한 모범 사례입니다. 실제로 이는 워크로드와 기본 인프라를 지정된 AWS 리전의 여러 가용 영역(AZs)에 분산시키는 것을 의미합니다. Amazon EKS 환경에 이러한 특성이 적용되도록 하면 시스템의 전반적인 신뢰성이 향상됩니다. 이와 함께 EKS 환경은 다양한 구성 요소(예: VPCs), 구성 요소(예: ELBs) 및 통합(예: ECR 및 기타 컨테이너 레지스트리)으로 구성될 수도 있습니다.

고가용성 시스템과 기타 사용 사례별 구성 요소의 조합은 데이터가 전송되고 처리되는 방식에 중요한 역할을 할 수 있습니다. 이로 인해 데이터 전송 및 처리로 인해 발생하는 비용에 영향을 미칩니다.

아래에 자세히 설명된 사례는 다양한 도메인 및 사용 사례에 대한 비용 효율성을 달성하기 위해 EKS 환경을 설계하고 최적화하는 데 도움이 됩니다.

포드 간 통신

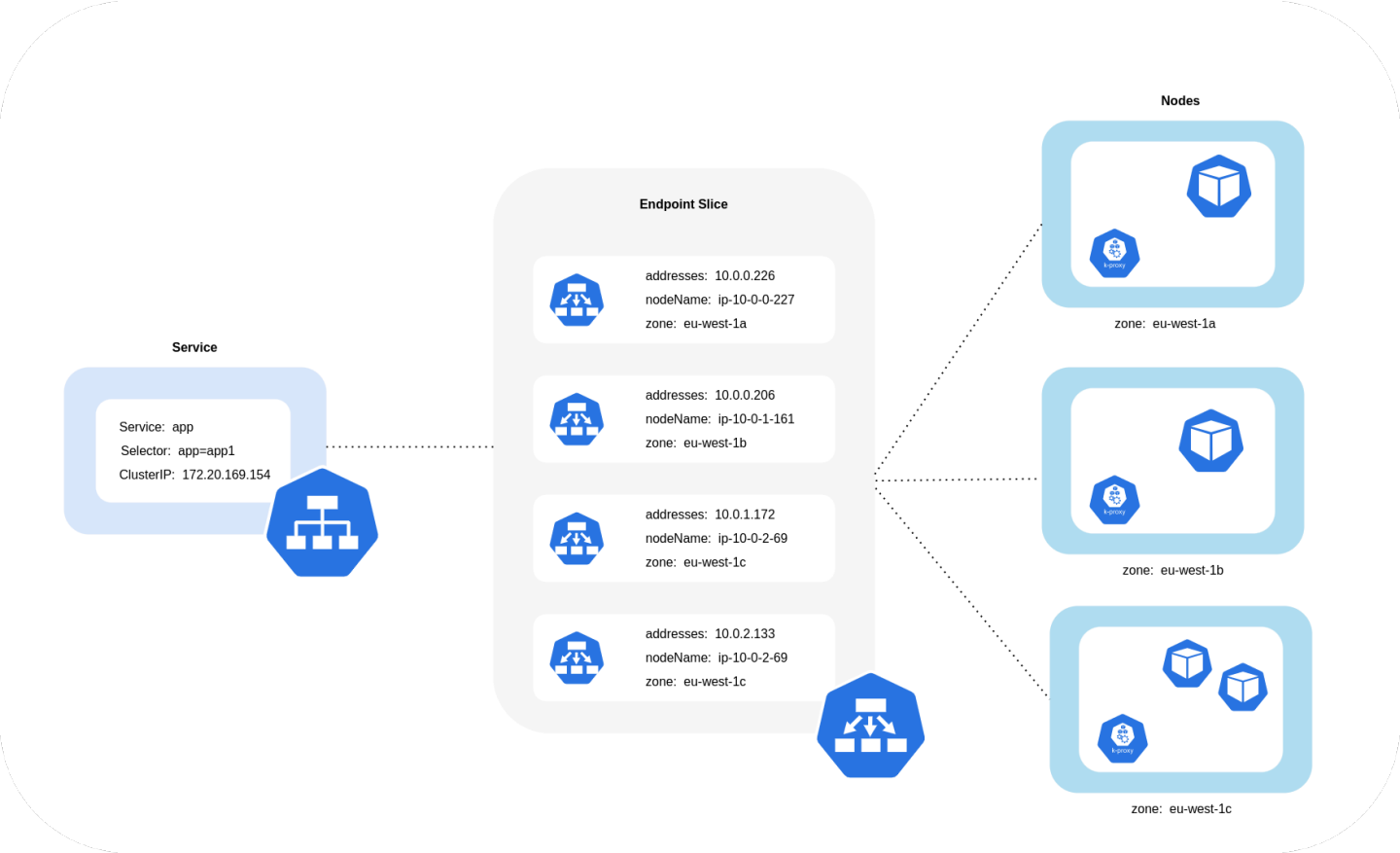
설정에 따라 포드 간 네트워크 통신 및 데이터 전송은 Amazon EKS 워크로드 실행의 전체 비용에 상당한 영향을 미칠 수 있습니다. 이 섹션에서는 고가용성(HA) 아키텍처, 애플리케이션 성능 및 복원력을 고려하면서 포드 간 통신과 관련된 비용을 완화하기 위한 다양한 개념과 접근 방식을 다룹니다.

가용 영역으로 트래픽 제한

Kubernetes 프로젝트는 초기에 노드에 할당된 kubernetes.io/hostname, topology.kubernetes.io/region 및 topology.kubernetes.io/zone 같은 레이블을 포함한 토폴로지 인식 구문을 개발하여 장애 도메인 및 토폴로지 인식 볼륨 프로비저너 간의 워크로드 분산과 같은 기능을 활성화하기 시작했습니다. Kubernetes 1.17을 졸업한 후 레이블을 활용하여 포드 간 통신을 위한 토폴로지 인식 라우팅 기능도 활성화했습니다.

다음은 비용을 절감하고 지연 시간을 최소화하기 위해 EKS 클러스터의 포드 간 AZ 간 트래픽 양을 제어하는 방법에 대한 몇 가지 전략입니다.

클러스터의 포드 간 AZ 간 트래픽 양(예: 바이트 단위로 전송된 데이터 양)을 세밀하게 확인하려면 이 게시물을 참조하세요.



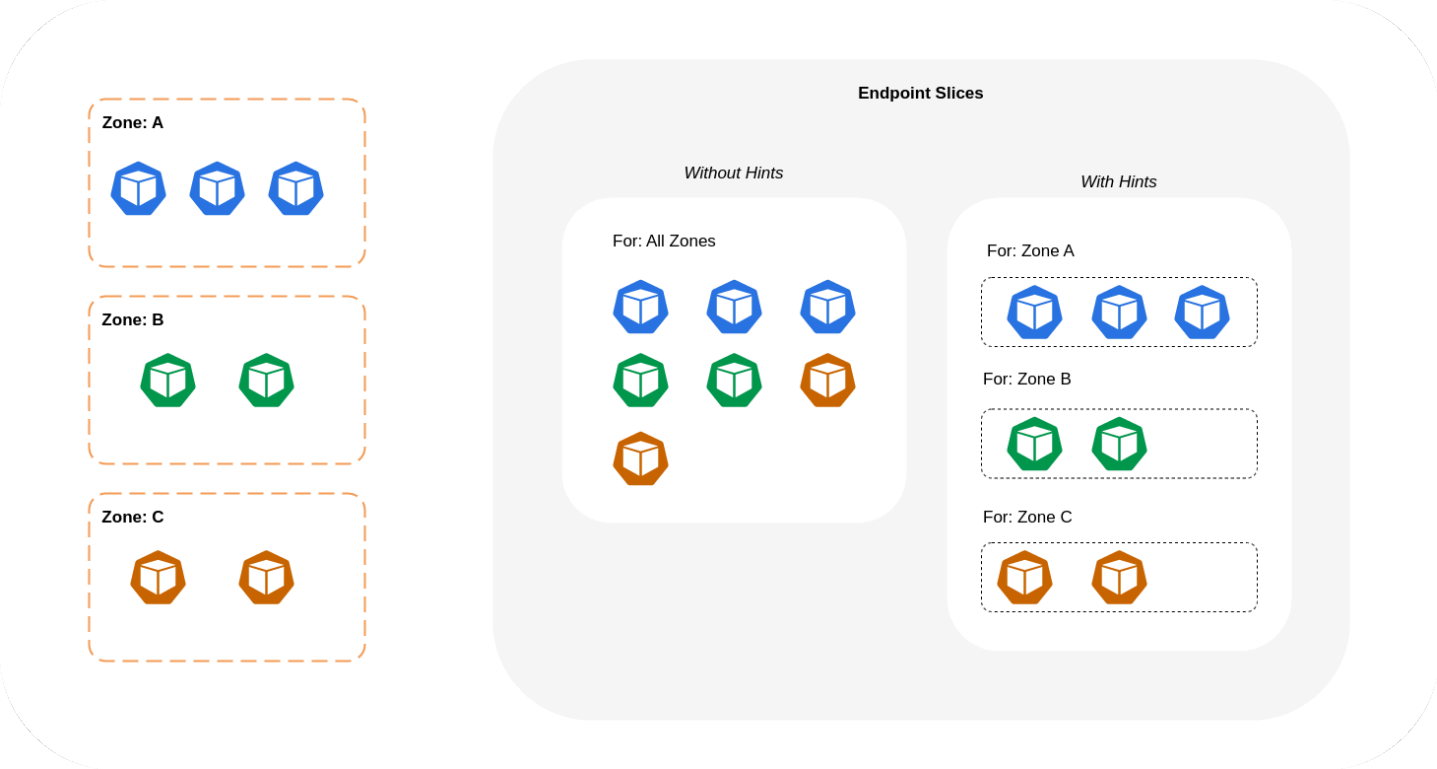
위 다이어그램에서 볼 수 있듯이 서비스는 포드로 향하는 트래픽을 수신하는 안정적인 네트워크 추상화 계층입니다. 서비스가 생성되면 여러 EndpointSlices가 생성됩니다. 각 EndpointSlice에는 실행 중인 노드와 함께 포드 주소의 하위 집합이 포함된 엔드포인트 목록과 추가 토폴로지 정보가 있습니다. Amazon VPC CNI를 사용하는 경우 모든 노드에서 실행되는 데몬 세트인 kube-proxy는 포드 통신 및 서비스 검색을 활성화하는 네트워크 규칙을 유지합니다(대체 eBPF 기반 CNIs kube-proxy를 사용하지 않지만 동등한 동작을 제공할 수 있음). 내부 라우팅의 역할을 수행하지만 생성된 EndpointSlices에서 소비하는 것에 따라 수행됩니다.

EKS에서 kube-proxy는 주로 노드 또는 AZ 배치에 관계없이 클러스터의 모든 포드에서 트래픽 분산을 위해 iptables NAT 규칙(또는 IPVS, NFTables를 대안으로 사용)을 사용합니다. 이 기본 배포로 인해 교차 AZ 트래픽 라우팅이 발생하여 대규모 배포에서 민감한 애플리케이션의 지연 시간이 증가하고 AZ 간 데이터 전송 요금이 증가할 수 있습니다.

토폴로지 인식 라우팅 사용(이전 명칭은 토폴로지 인식 힌트)

Kubernetes 서비스에서 토폴로지 인식 라우팅이 활성화되고 구현되면 EndpointSlice 컨트롤러는 클러스터가 분산된 여러 영역에 엔드포인트를 비례적으로 할당합니다. 이러한 각 엔드포인트에 대해 EndpointSlice 컨트롤러는 영역에 대한 힌트도 설정합니다. 힌트는 엔드포인트가 트래픽을 제공해야 하는 영역을 설명합니다. 그러면 kube-proxy는 적용된 힌트를 기반으로 영역에서 엔드포인트로 트래픽을 라우팅합니다.

아래 다이어그램은 힌트가 있는 EndpointSlices가 영역별 오리진 지점을 기반으로 어떤 대상으로 이동해야 하는지 알 kube-proxy 수 있는 방식으로 구성되는 방법을 보여줍니다. 힌트가 없으면 이러한 할당이나 조직이 없으며 트래픽은 출처에 관계없이 다양한 영역 대상으로 프록시됩니다.



경우에 따라 EndpointSlice 컨트롤러는 다른 영역에 힌트를 적용할 수 있습니다. 즉, 엔드포인트가 다른 영역에서 시작된 트래픽을 제공할 수 있습니다. 그 이유는 서로 다른 영역의 엔드포인트 간에 트래픽이 균등하게 분산되도록 하기 위한 것입니다.

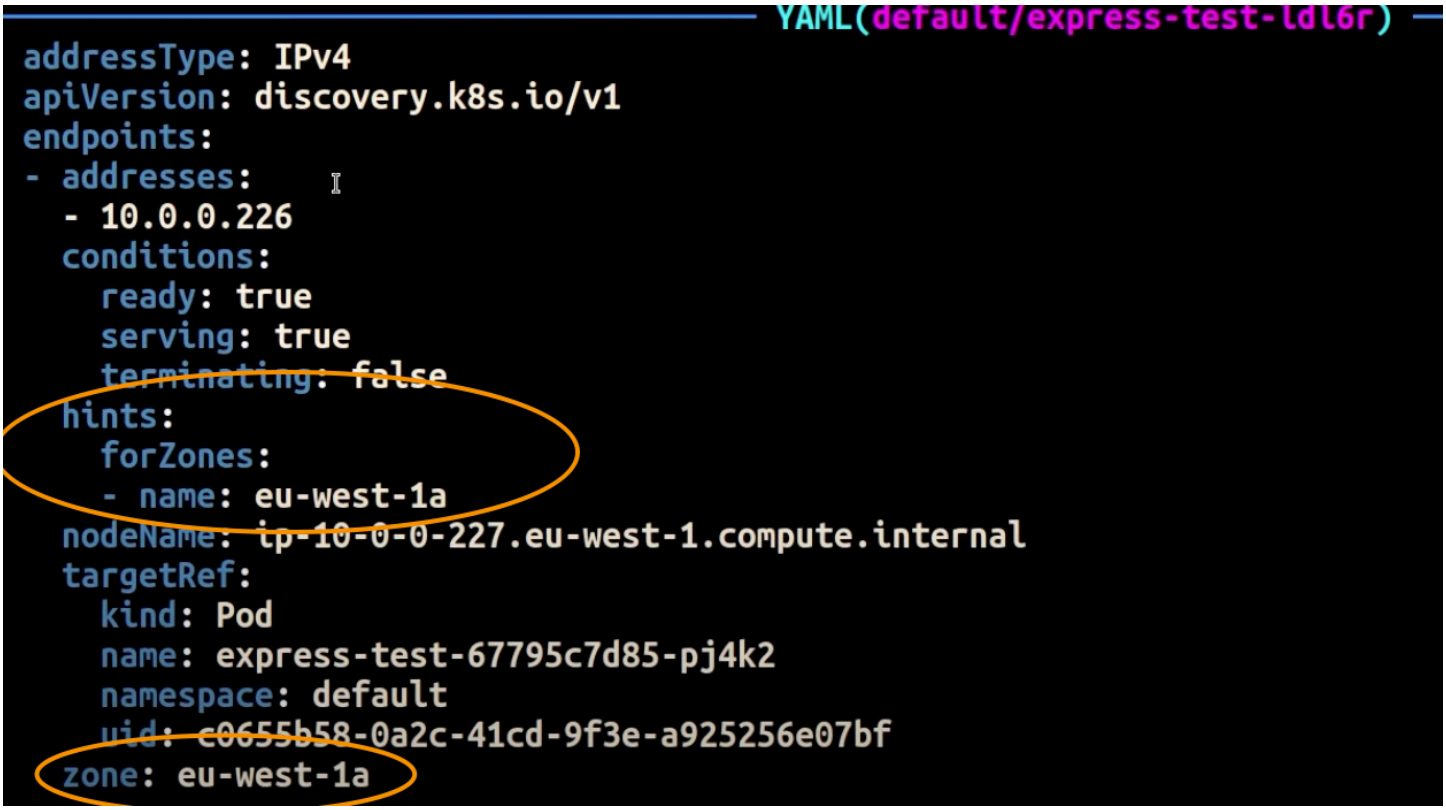
다음은 서비스에 대한 토폴로지 인식 라우팅을 활성화하는 방법에 대한 코드 조각입니다.

```
apiVersion: v1
kind: Service
metadata:
  name: orders-service
  namespace: ecommerce
  annotations:
```

```
    service.kubernetes.io/topology-mode: Auto
spec:
  selector:
    app: orders
  type: ClusterIP
  ports:

* protocol: TCP
port: 3003
targetPort: 3003
```

아래 스크린샷은 AZ에서 실행 중인 포드 복제본의 엔드포인트에 힌트를 성공적으로 적용한 EndpointSlice 컨트롤러의 결과를 보여줍니다`eu-west-1a`.



> **Note**
>
> 토폴로지 인식 라우팅은 여전히 베타 버전입니다. 이 기능은 컨트롤러가 영역 간에 엔드포인트를 비례적으로 할당하지만 영역의 노드 리소스가 너무 불균형하여 과도한 과부하를 피할 수 없는 경우 힌트 할당을 건너뛸 수 있으므로 클러스터 토폴로지 전체에 균등하게 분산된 워크로드에서 보다 예측 가능하게 작동합니다. 따라서 포드 토폴로지 분산 제약 조건과 같은 애플리케이션의 가용성을 높이는 예약 제약 조건과 함께 사용하는 것이 좋습니다. 비례 분포를 계산할

때 중단 또는 교체가 실시간으로 감지되지 않으므로 Amazon EC2 스팟 인스턴스를 사용할 때와 같이 영역 간에 용량이 변동하는 경우에도 힌트가 할당되지 않을 수 있습니다.

트래픽 배포 사용

Kubernetes 1.30에 도입되고 1.33에 정식으로 제공되는 트래픽 분산은 동일한 영역 트래픽 선호도를 위해 토폴로지 인식 라우팅에 대한 더 간단한 대안을 제공합니다. 토폴로지 인식 라우팅은 엔드포인트 과부하를 방지하기 위해 트래픽 라우팅에 대한 지능형 접근 방식을 사용하려고 시도하지만 예측할 수 없는 동작이 발생했습니다. 트래픽 분산은 대신 예측 가능성을 우선시합니다. PreferClose 옵션은 kube-proxy에 EndpointSlice Controller에서 설정한 영역 힌트를 기반으로 먼저 트래픽을 동일한 영역 엔드포인트로 라우팅하는 규칙을 생성하도록 지시합니다. 동일한 영역 엔드포인트를 사용할 수 없는 경우 서비스의 모든 클러스터 엔드포인트에 트래픽을 분산하는 것으로 돌아갑니다. 이 기능은 토폴로지 인식 라우팅이 제공하는 로드의 균일한 분포가 아닌 근접성을 위한 최적화의 장단점을 수용하는 워크로드를 위해 설계되었습니다.

다음은 서비스에 대한 트래픽 분산을 활성화하는 방법에 대한 코드 조각입니다.

```
apiVersion: v1
kind: Service
metadata:
  name: orders-service
  namespace: ecommerce
spec:
  trafficDistribution: PreferClose
  selector:
    app: orders
  type: ClusterIP
  ports:

* protocol: TCP
port: 3003
targetPort: 3003
```

트래픽 분산을 활성화하면 일반적인 문제가 발생합니다. 대부분의 트래픽이 동일한 영역에서 발생하는 경우 단일 AZ 내의 엔드포인트에 과부하가 걸릴 수 있습니다. 이 오버로드로 인해 다음과 같은 심각한 문제가 발생할 수 있습니다.

- 다중 AZ 배포를 관리하는 단일 Horizontal Pod Autoscaler(HPA)는 여러 AZs. 그러나이 작업은 영향을 받는 영역의 증가된 부하를 효과적으로 해결하지 못합니다.

- 이 상황은 결과적으로 리소스 비효율성을 초래할 수 있습니다. Karpenter와 같은 클러스터 오토스케일러가 여러 AZs에서 포드 스케일 아웃을 감지하면 영향을 받지 않는 AZs에 추가 노드를 프로비저닝하여 불필요한 리소스 할당이 발생할 수 있습니다.

이 문제를 해결하려면:

- 영역별로 HPAs가 있는 별도의 배포를 생성하여 서로 독립적으로 확장할 수 있습니다.
- 토폴로지 분산 제약 조건을 활용하여 클러스터 전반의 워크로드 분산을 보장함으로써 트래픽이 많은 영역에서 엔드포인트 과부하를 방지하는 데 도움이 됩니다.

Autoscaler 사용: 특정 AZ에 노드 프로비저닝

여러 AZs에서 고가용성 환경에서 워크로드를 실행하는 것이 좋습니다. 이렇게 하면 특히 AZ에 문제가 발생한 경우 애플리케이션의 신뢰성이 향상됩니다. 네트워크 관련 비용을 줄이기 위해 신뢰성을 희생하려는 경우 노드를 단일 AZ로 제한할 수 있습니다.

동일한 AZ에서 모든 포드를 실행하려면 동일한 AZ에서 작업자 노드를 프로비저닝하거나 동일한 AZ에서 실행되는 작업자 노드의 포드를 예약합니다. 단일 AZ 내에서 노드를 프로비저닝하려면 Cluster Autoscaler(CA)를 사용하여 동일한 AZ에 속하는 서브넷이 있는 노드 그룹을 정의합니다. Karpenter의 경우를 `topology.kubernetes.io/zone` 사용하고 작업자 노드를 생성할 AZ를 지정합니다. 예를 들어 아래 Karpenter 프로비저너 코드 조각은 us-west-2a AZ에 노드를 프로비저닝합니다.

Karpenter

```
apiVersion: karpenter.sh/v1
kind: Provisioner
metadata:
name: single-az
spec:
  requirements:

* key: "topology.kubernetes.io/zone"`
operator: In
values: ["us-west-2a"]
```

Cluster Autoscaler(CA)

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
```

```
  name: my-ca-cluster
  region: us-east-1
  version: "1.21"
availabilityZones:

* us-east-1a
managedNodeGroups:
* name: managed-nodes
labels:
  role: managed-nodes
instanceType: t3.medium
minSize: 1
maxSize: 10
desiredCapacity: 1
...
```

포드 할당 및 노드 선호도 사용

또는 여러 AZs에서 작업자 노드를 실행하는 경우 각 노드에는 AZ 값이 포함된 topology.kubernetes.io/zone 레이블이 있습니다(예: us-west-2a 또는 us-west-2b). `nodeSelector` 또는 `nodeAffinity`를 사용하여 단일 AZ의 노드에 대한 포드를 예약할 수 있습니다. 예를 들어 다음 매니페스트 파일은 AZ us-west-2a에서 실행되는 노드 내의 포드를 예약합니다.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  nodeSelector:
    topology.kubernetes.io/zone: us-west-2a
  containers:

* name: nginx
image: nginx
imagePullPolicy: IfNotPresent
```

노드로 트래픽 제한

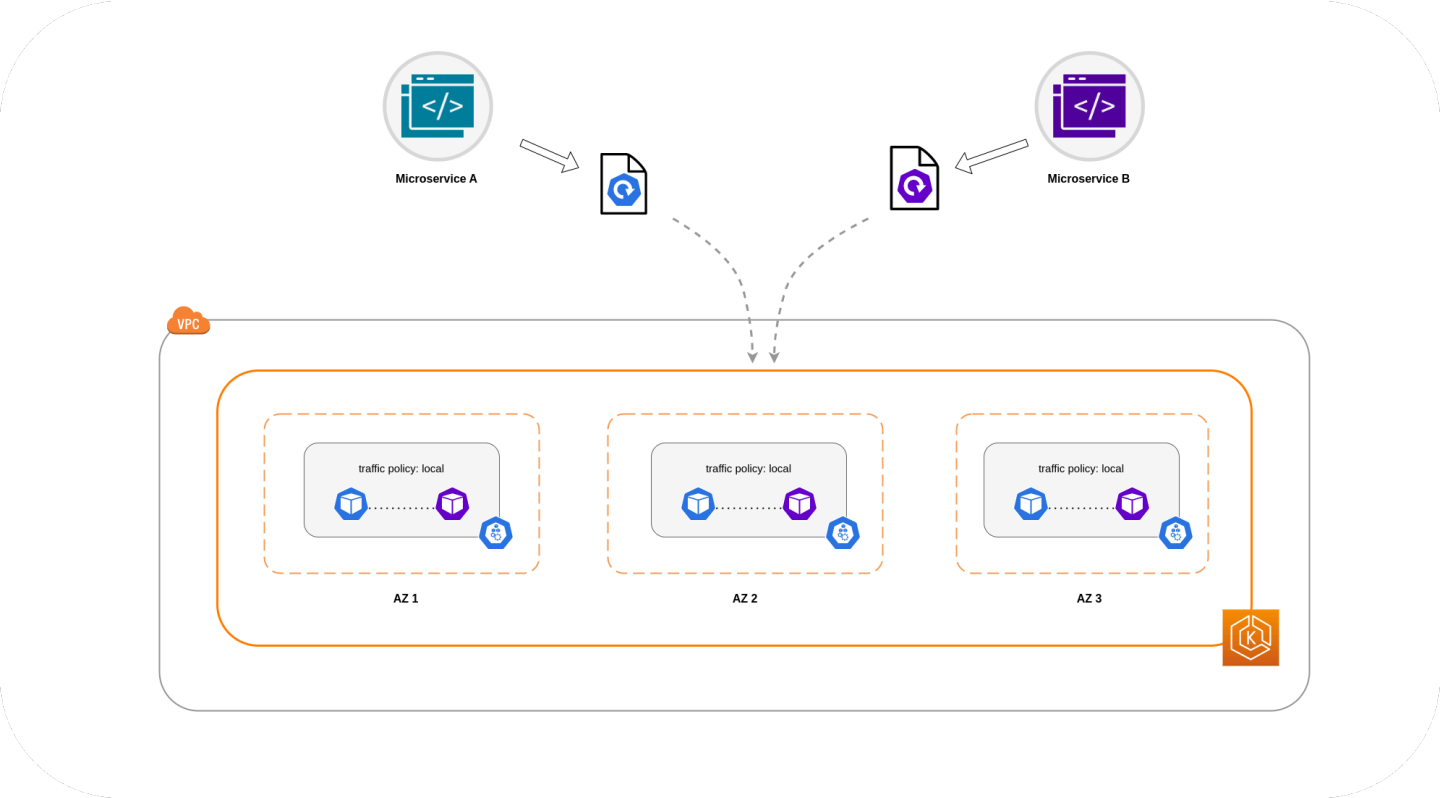
영역 수준에서 트래픽을 제한하는 것으로는 충분하지 않은 경우가 있습니다. 비용 절감 외에도 상호 통신이 빈번한 특정 애플리케이션 간의 네트워크 지연 시간을 줄여야 하는 요구 사항이 추가될 수 있습니

다. 최적의 네트워크 성능을 달성하고 비용을 절감하려면 트래픽을 특정 노드로 제한할 수 있는 방법이 필요합니다. 예를 들어 마이크로서비스 A는 고가용성(HA) 설정에서도 항상 노드 1의 마이크로서비스 B와 통신해야 합니다. 노드 1의 마이크로서비스 A가 노드 2의 마이크로서비스 B와 통신하면 특히 노드 2가 별도의 AZ에 있는 경우 이러한 특성의 애플리케이션에 대해 원하는 성능에 부정적인 영향을 미칠 수 있습니다.

서비스 내부 트래픽 정책 사용

포드 네트워크 트래픽을 노드로 제한하기 위해 서비스 내부 트래픽 정책 를 사용할 수 있습니다. 기본적으로 워크로드의 서비스로 전송된 트래픽은 생성된 여러 엔드포인트에 무작위로 분산됩니다. 따라서 HA 아키텍처에서는 마이크로서비스 A의 트래픽이 서로 다른 AZs. 그러나 서비스의 내부 트래픽 정책을 로 설정하면 `Local`트래픽이 시작된 노드의 엔드포인트로 트래픽이 제한됩니다. 이 정책은 노드-로컬 엔드포인트의 독점적 사용을 지시합니다. 즉, 해당 워크로드에 대한 네트워크 트래픽 관련 비용은 배포가 클러스터 전체인 경우보다 저렴합니다. 또한 지연 시간이 짧아져 애플리케이션의 성능이 향상됩니다.

> **Note**
>
> 이 기능은 Kubernetes의 토폴로지 인식 라우팅과 결합할 수 없습니다.



다음은 서비스에 대한 내부 트래픽 정책을 설정하는 방법에 대한 코드 조각입니다.

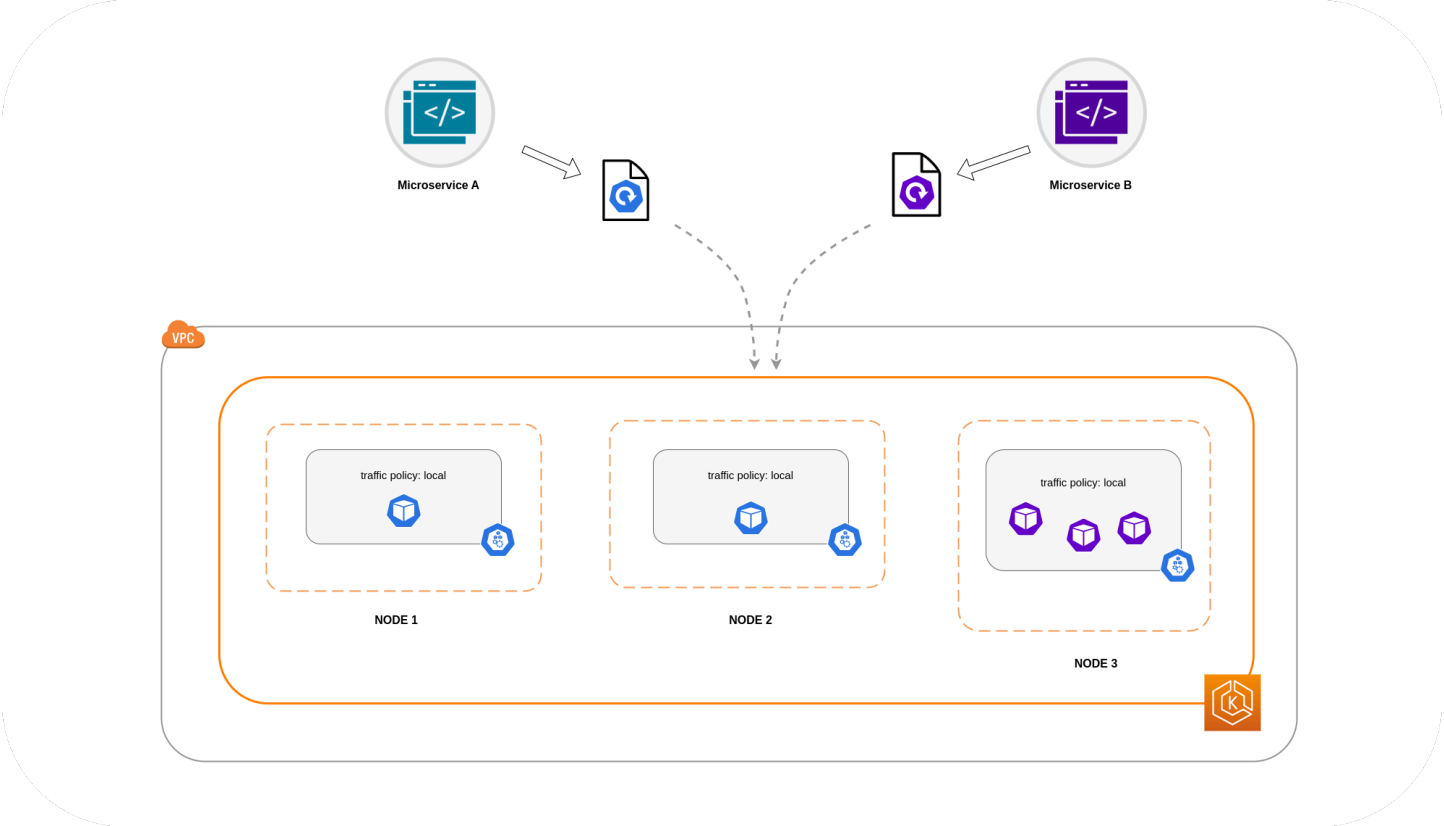
```
apiVersion: v1
kind: Service
metadata:
  name: orders-service
  namespace: ecommerce
spec:
  selector:
    app: orders
  type: ClusterIP
  ports:

* protocol: TCP
port: 3003
targetPort: 3003
  internalTrafficPolicy: Local
```

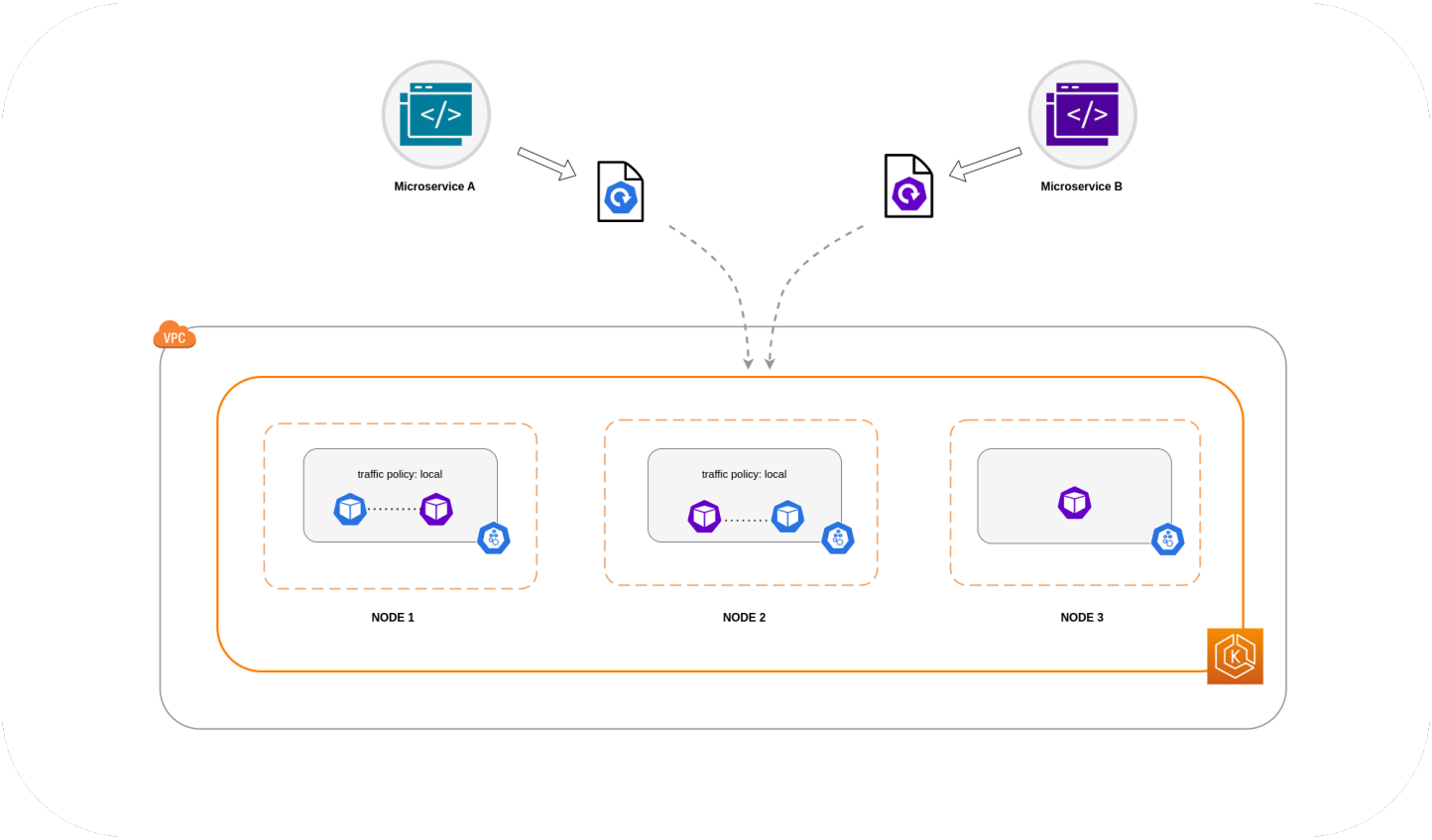
트래픽 감소로 인한 애플리케이션의 예기치 않은 동작을 방지하려면 다음 접근 방식을 고려해야 합니다.

- 각 통신 포드에 대해 충분한 복제본 실행
- 토폴로지 분산 제약 조건을 사용하여 포드를 비교적 균등하게 분산시킵니다.
- 포드 통신 코로케이션에 포드 선호도 규칙 사용

이 예제에서는 마이크로서비스 A 복제본 2개와 마이크로서비스 B 복제본 3개가 있습니다. 마이크로서비스 A의 복제본이 노드 1과 2 사이에 분산되어 있고 마이크로서비스 B의 노드 3에 복제본 3개가 모두 있는 경우 Local 내부 트래픽 정책으로 인해 통신할 수 없습니다. 사용 가능한 노드-로컬 엔드포인트가 없으면 트래픽이 삭제됩니다.



마이크로서비스 B의 노드 1과 2에 복제본 3개 중 2개가 있는 경우 피어 애플리케이션 간에 통신이 이루어집니다. 그러나 통신할 피어 복제본이 없는 마이크로서비스 B의 격리된 복제본이 여전히 있을 것입니다.



> **Note**
>
> 일부 시나리오에서는 위 다이어그램에 표시된 것과 같은 격리된 복제본이 여전히 목적(예: 외부 수신 트래픽의 요청 처리)을 제공하는 경우 문제가 되지 않을 수 있습니다.

토폴로지 분산 제약 조건과 함께 서비스 내부 트래픽 정책 사용

토폴로지 분산 제약 조건과 함께 내부 트래픽 정책을 사용하면 여러 노드에서 마이크로서비스를 통신하기 위한 적절한 수의 복제본이 있는지 확인하는 데 유용할 수 있습니다.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: express-test
spec:
  replicas: 6
  selector:
    matchLabels:
      app: express-test
```

```
  template:
    metadata:
      labels:
        app: express-test
        tier: backend
    spec:
      topologySpreadConstraints:
      - maxSkew: 1
        topologyKey: "topology.kubernetes.io/zone"
        whenUnsatisfiable: ScheduleAnyway
        labelSelector:
          matchLabels:
            app: express-test
```

포드 선호도 규칙과 함께 서비스 내부 트래픽 정책 사용

또 다른 접근 방식은 서비스 내부 트래픽 정책을 사용할 때 포드 선호도 규칙을 사용하는 것입니다. 포드 선호도를 사용하면 빈번한 통신으로 인해 스케줄러가 특정 포드를 공동 배치하도록 영향을 미칠 수 있습니다. 특정 포드에 엄격한 예약 제약 조건(requiredDuringSchedulingIgnoredDuringExecution)을 적용하면 스케줄러가 노드에 포드를 배치할 때 포드 코로케이션에 대한 더 나은 결과를 얻을 수 있습니다.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: graphql
  namespace: ecommerce
  labels:
    app.kubernetes.io/version: "0.1.6"
    ...
    spec:
      serviceAccountName: graphql-service-account
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - orders
            topologyKey: "kubernetes.io/hostname"
```

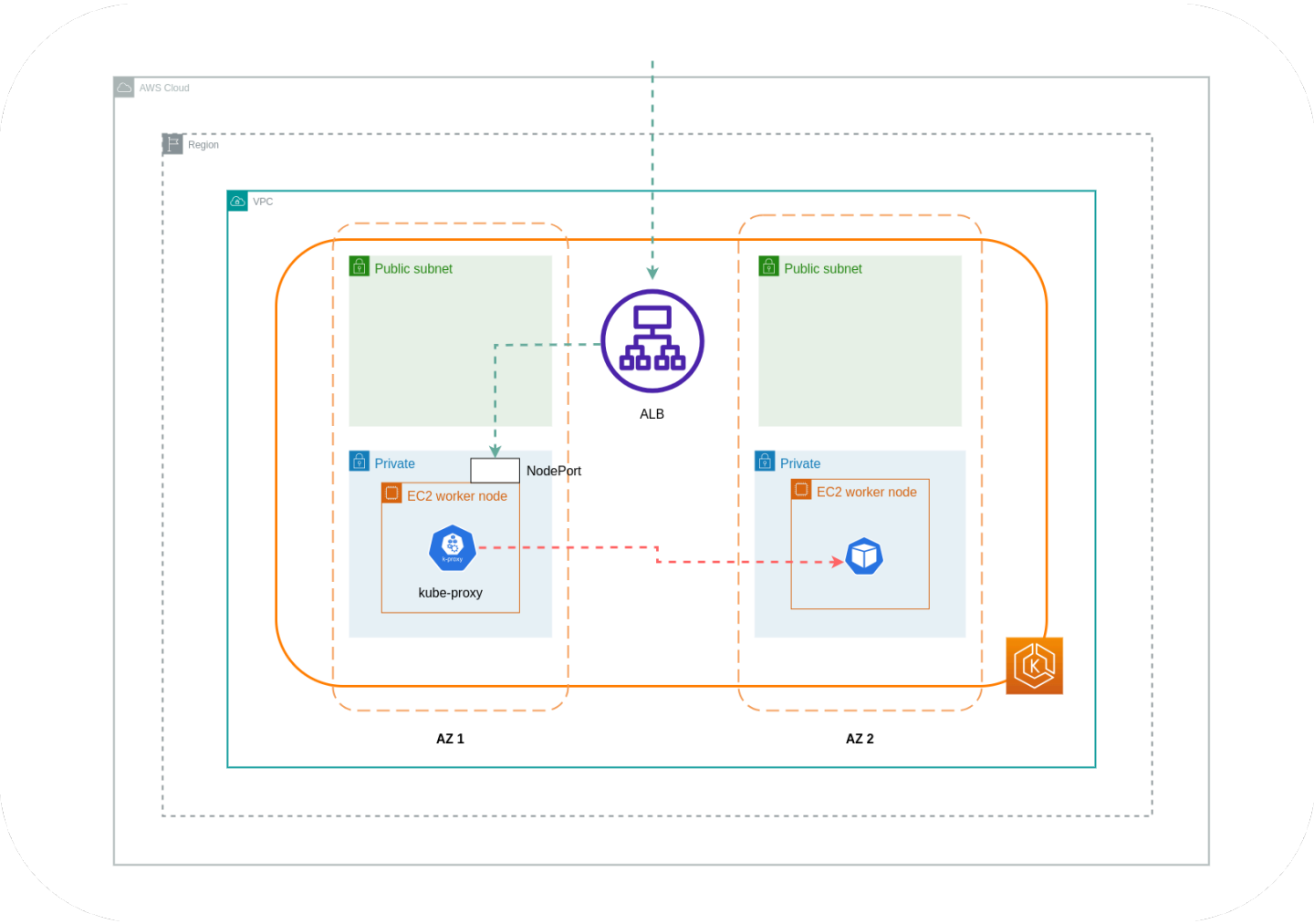
Load Balancer서와 포드 통신

EKS 워크로드는 일반적으로 EKS 클러스터의 관련 포드로 트래픽을 분산하는 로드 밸런서가 앞에 있습니다. 아키텍처는 내부 및/또는 외부 방향 로드 밸런서를 포함할 수 있습니다. 아키텍처 및 네트워크 트래픽 구성에 따라 로드 밸런서와 포드 간의 통신이 데이터 전송 요금에 상당한 영향을 미칠 수 있습니다.

AWS Load Balancer Controller를 사용하여 ELB 리소스(ALB 및 NLB) 생성을 자동으로 관리할 수 있습니다. 이러한 설정에서 발생하는 데이터 전송 요금은 네트워크 트래픽이 취하는 경로에 따라 달라집니다. AWS Load Balancer 컨트롤러는 인스턴스 모드와 IP 모드라는 두 가지 네트워크 트래픽 모드를 지원합니다.

인스턴스 모드를 사용하면 EKS 클러스터의 각 노드에서 NodePort가 열립니다. 그러면 로드 밸런서가 노드 간에 트래픽을 균등하게 프록시합니다. 노드에 대상 포드가 실행 중인 경우 데이터 전송 비용이 발생하지 않습니다. 그러나 대상 포드가 트래픽을 수신하는 NodePort와 다른 노드와 다른 AZ에 있는 경우 kube-proxy에서 대상 포드로 추가 네트워크 홉이 발생합니다. 이러한 시나리오에서는 교차 AZ 데이터 전송 요금이 발생합니다. 노드 간에 트래픽이 균등하게 분산되기 때문에 kube 프록시에서 관련 대상 포드로의 교차 영역 네트워크 트래픽 홉과 관련된 추가 데이터 전송 요금이 발생할 가능성이 높습니다.

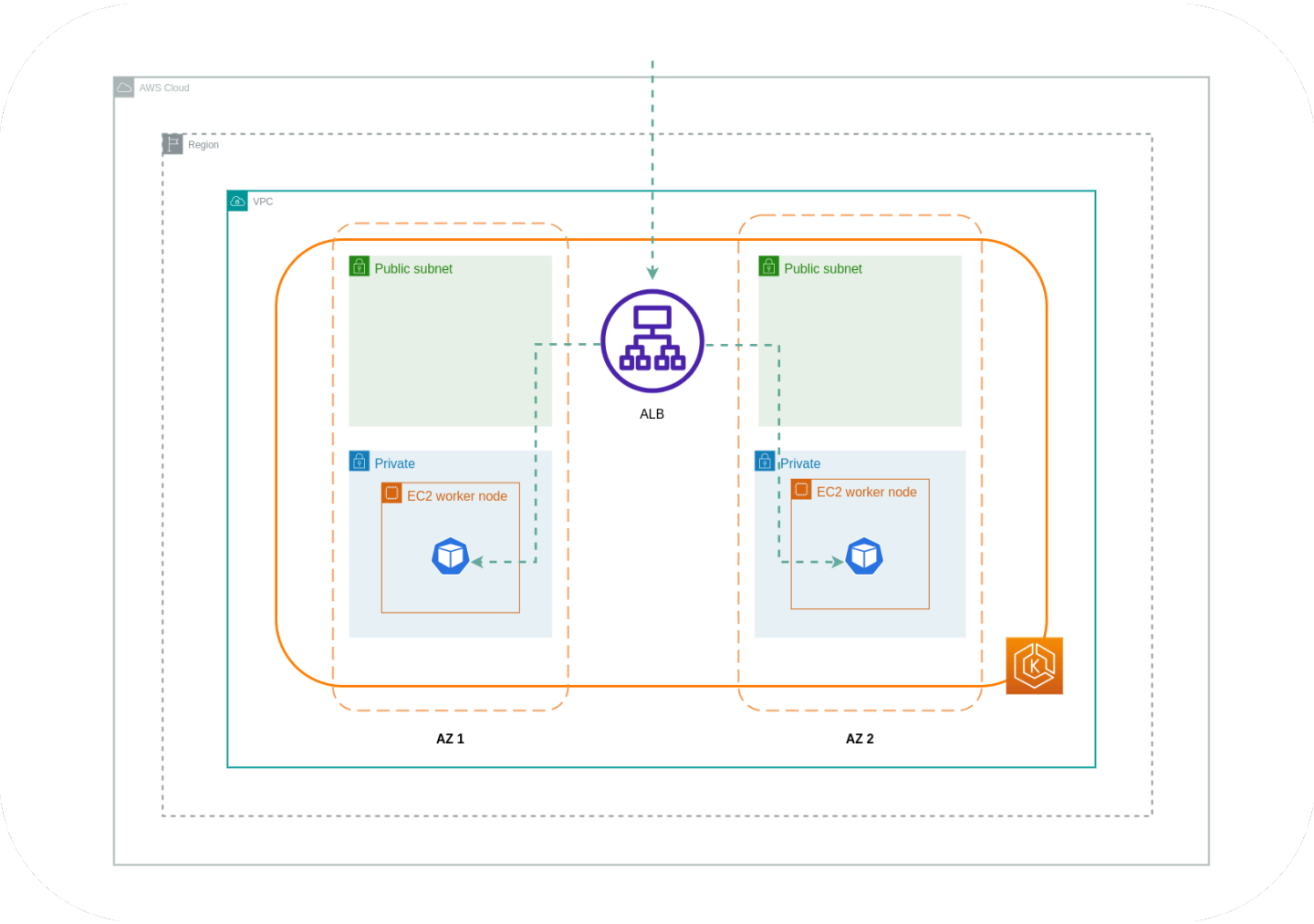
아래 다이어그램은 로드 밸런서에서 NodePort로, 이후에서 다른 AZ의 별도 노드에 있는 `kube-proxy` 대상 포드로 흐르는 트래픽의 네트워크 경로를 보여줍니다. 다음은 인스턴스 모드 설정의 예입니다.



IP 모드를 사용하는 경우 네트워크 트래픽은 로드 밸런서에서 대상 포드로 직접 프록시됩니다. 따라서 이 접근 방식에는 데이터 전송 요금이 부과되지 않습니다.

> **Note**
>
> 데이터 전송 요금을 줄이려면 로드 밸런서를 IP 트래픽 모드로 설정하는 것이 좋습니다. 이 설정의 경우 로드 밸런서가 VPC의 모든 서브넷에 배포되었는지 확인하는 것도 중요합니다.

아래 다이어그램은 네트워크 IP 모드에서 로드 밸런서에서 포드로 흐르는 트래픽의 네트워크 경로를 보여줍니다.



컨테이너 레지스트리에서 데이터 전송

Amazon ECR

Amazon ECR 프라이빗 레지스트리로의 데이터 전송은 무료입니다. 리전 내 데이터 전송에는 비용이 발생하지 않지만 인터넷 및 리전 간 데이터 전송에는 전송 양쪽의 인터넷 데이터 전송 요금이 부과됩니다.

ECRs 기본 제공 이미지 복제 기능을 활용하여 관련 컨테이너 이미지를 워크로드와 동일한 리전에 복제해야 합니다. 이렇게 하면 복제 요금이 한 번 청구되고 동일한 리전(리전 내) 이미지 풀은 모두 무료입니다.

인터페이스 VPC 엔드포인트를 사용하여 리전 내 ECR 리포지토리에 연결하여 ECR(데이터 전송)에서 이미지를 가져오는 것과 관련된 데이터 전송 비용을 추가로 줄일 수 있습니다. ECR의 퍼블릭 AWS 엔드포인트(NAT 게이트웨이 및 인터넷 게이트웨이를 통해)에 연결하는 대체 접근 방식은 데이터 처리 및

전송 비용을 높입니다. 다음 섹션에서는 워크로드와 AWS 서비스 간의 데이터 전송 비용 절감에 대해 자세히 설명합니다.

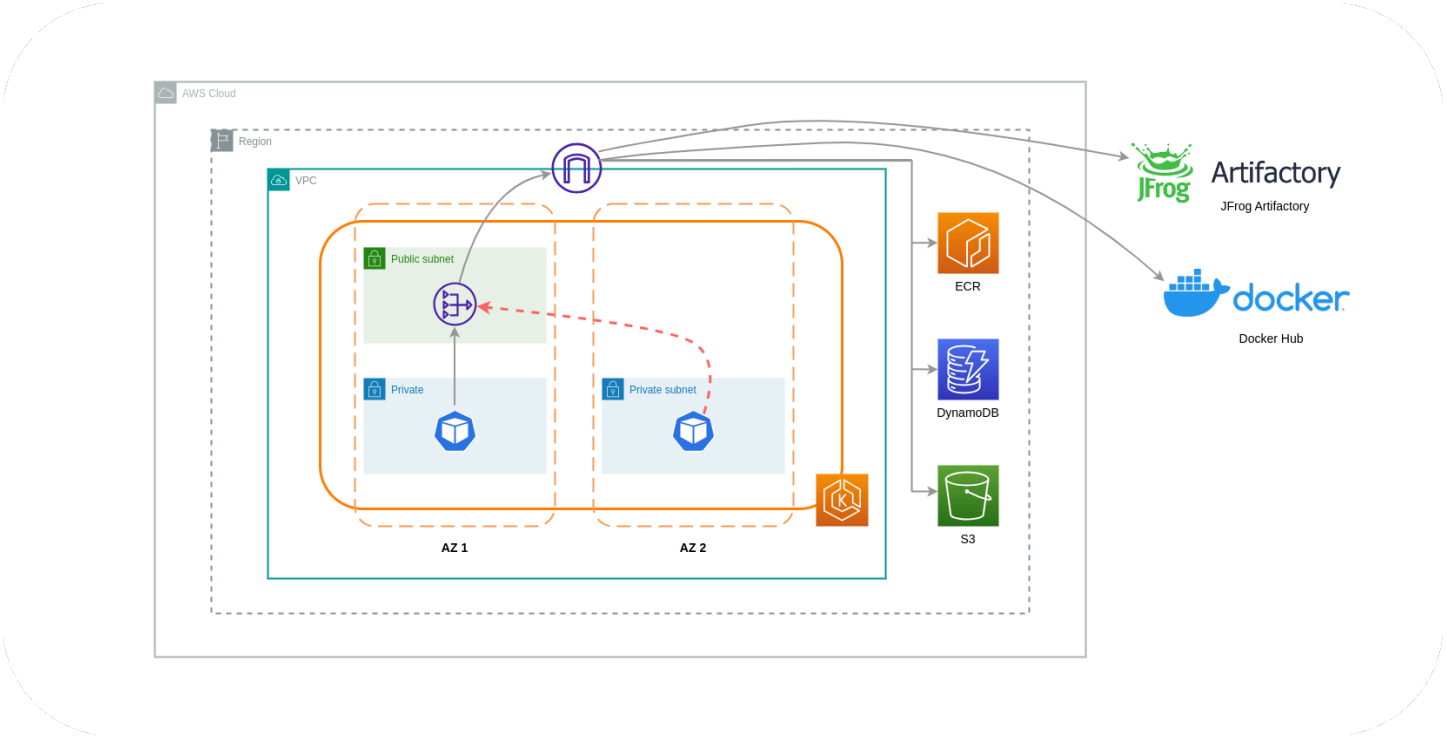
특히 큰 이미지로 워크로드를 실행하는 경우 사전 캐시된 컨테이너 이미지로 사용자 지정 Amazon Machine Image(AMIs)를 구축할 수 있습니다. 이렇게 하면 컨테이너 레지스트리에서 EKS 작업자 노드로 초기 이미지 가져오기 시간과 잠재적 데이터 전송 비용을 줄일 수 있습니다.

인터넷 및 AWS 서비스로 데이터 전송

인터넷을 통해 Kubernetes 워크로드를 다른 AWS 서비스 또는 타사 도구 및 플랫폼과 통합하는 것이 일반적인 방법입니다. 관련 대상과의 트래픽을 라우팅하는 데 사용되는 기본 네트워크 인프라는 데이터 전송 프로세스에서 발생하는 비용에 영향을 미칠 수 있습니다.

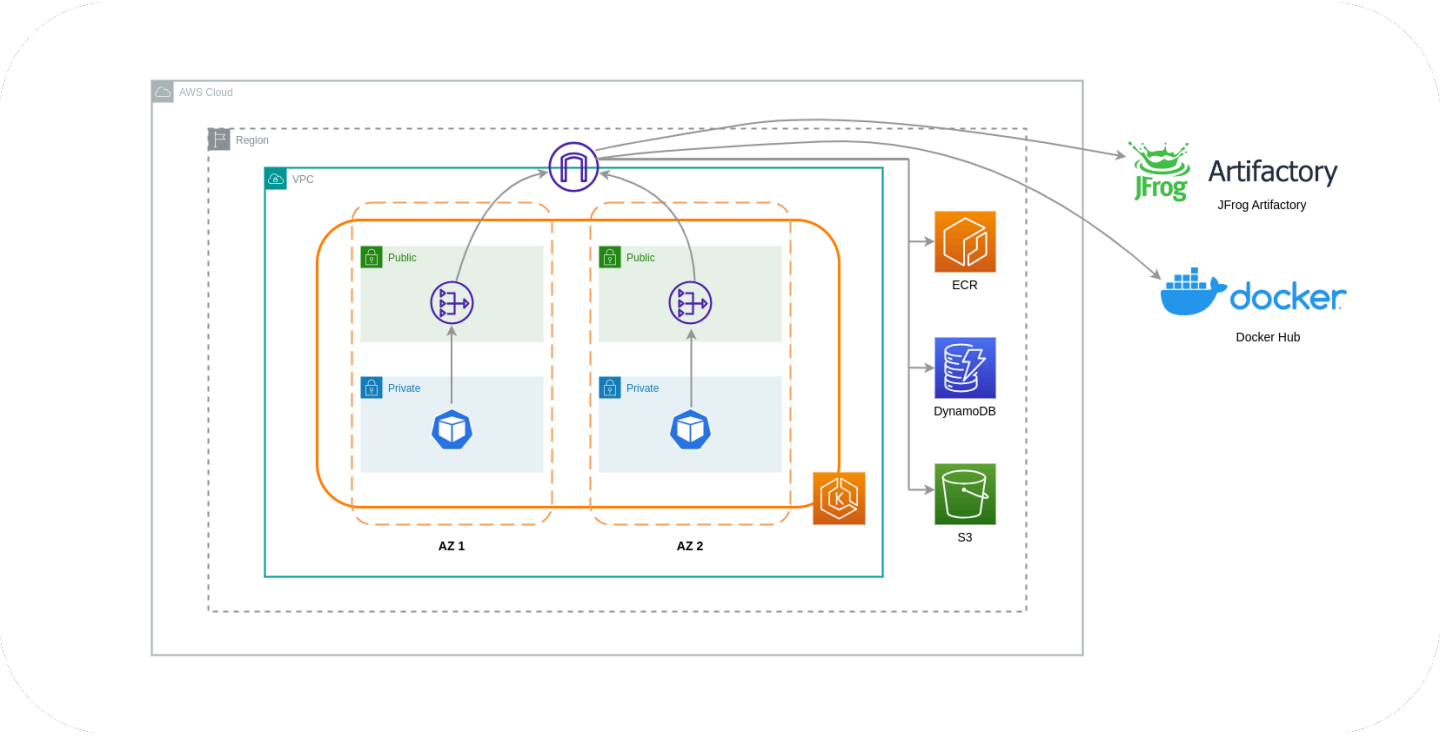
NAT 게이트웨이 사용

NAT 게이트웨이는 NAT(네트워크 주소 변환)를 수행하는 네트워크 구성 요소입니다. 아래 다이어그램은 다른 AWS 서비스(Amazon ECR, DynamoDB 및 S3) 및 타사 플랫폼과 통신하는 EKS 클러스터의 포드를 보여줍니다. 이 예제에서는 포드가 개별 AZs. 인터넷에서 트래픽을 보내고 받기 위해 NAT 게이트웨이가 하나의 AZ의 퍼블릭 서브넷에 배포되므로 프라이빗 IP 주소가 있는 모든 리소스가 인터넷에 액세스하기 위해 단일 퍼블릭 IP 주소를 공유할 수 있습니다. 이 NAT 게이트웨이는 인터넷 게이트웨이 구성 요소와 통신하므로 패킷을 최종 대상으로 전송할 수 있습니다.



이러한 사용 사례에 NAT 게이트웨이를 사용하는 경우 각 AZ에 NAT 게이트웨이를 배포하여 데이터 전송 비용을 최소화할 수 있습니다. 이렇게 하면 인터넷으로 라우팅되는 트래픽이 동일한 AZ의 NAT 게이트웨이를 통과하여 AZ 간 데이터 전송을 피할 수 있습니다. 그러나 AZ 간 데이터 전송 비용을 절감할 수 있지만 이 설정의 영향은 아키텍처에서 추가 NAT 게이트웨이 비용이 발생한다는 것입니다.

이 권장 접근 방식은 아래 다이어그램에 나와 있습니다.



VPC 종단점 사용

이러한 아키텍처의 비용을 더욱 줄이려면 VPC 엔드포인트를 사용하여 워크로드와 AWS 서비스 간의 연결을 설정해야 합니다. VPC 엔드포인트를 사용하면 인터넷을 통과하는 데이터/네트워크 패킷 없이 VPC 내에서 AWS 서비스에 액세스할 수 있습니다. 모든 트래픽은 내부 트래픽이며 AWS 네트워크 내에 유지됩니다. VPC 엔드포인트에는 인터페이스 VPC 엔드포인트(많은 AWS 서비스에서 지원)와 게이트웨이 VPC 엔드포인트(S3 및 DynamoDB에서만 지원)의 두 가지 유형이 있습니다.

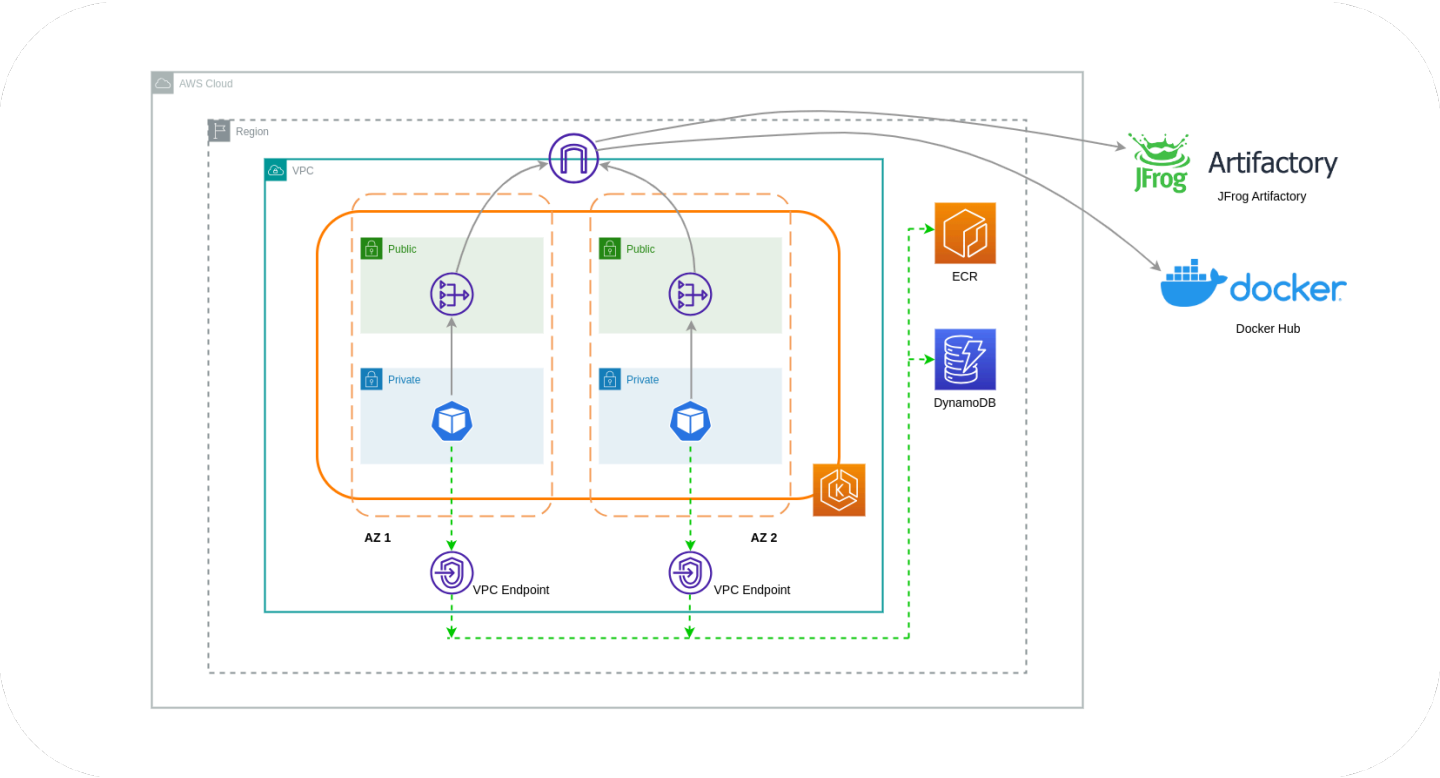
게이트웨이 VPC 엔드포인트

게이트웨이 VPC 엔드포인트와 관련된 시간당 또는 데이터 전송 비용은 없습니다. 게이트웨이 VPC 엔드포인트를 사용할 때는 VPC 경계를 넘어 확장할 수 없다는 점에 유의해야 합니다. VPC 피어링, VPN 네트워킹 또는 Direct Connect를 통해 사용할 수 없습니다.

인터페이스 VPC 엔드포인트

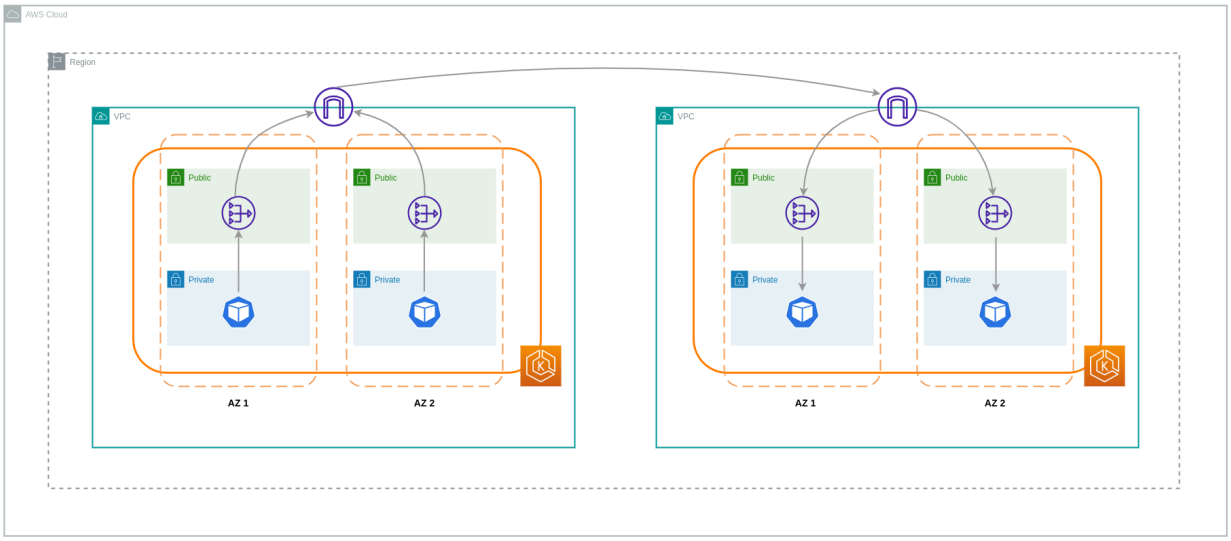
VPC 엔드포인트에는 시간당 요금이 부과되며 기본 ENI를 통한 데이터 처리와 관련된 추가 요금이 부과됩니다. AZ 간 데이터 전송은 [요금 부과되지 않음](https://aws.amazon.com/about-aws/whats-new/2022/04/aws-data-transfer-price-reduction-privatelink-transit-gateway-client-vpn-services/)입니다.

아래 다이어그램은 VPC 엔드포인트를 통해 AWS 서비스와 통신하는 포드를 보여줍니다.



VPCs 간 데이터 전송

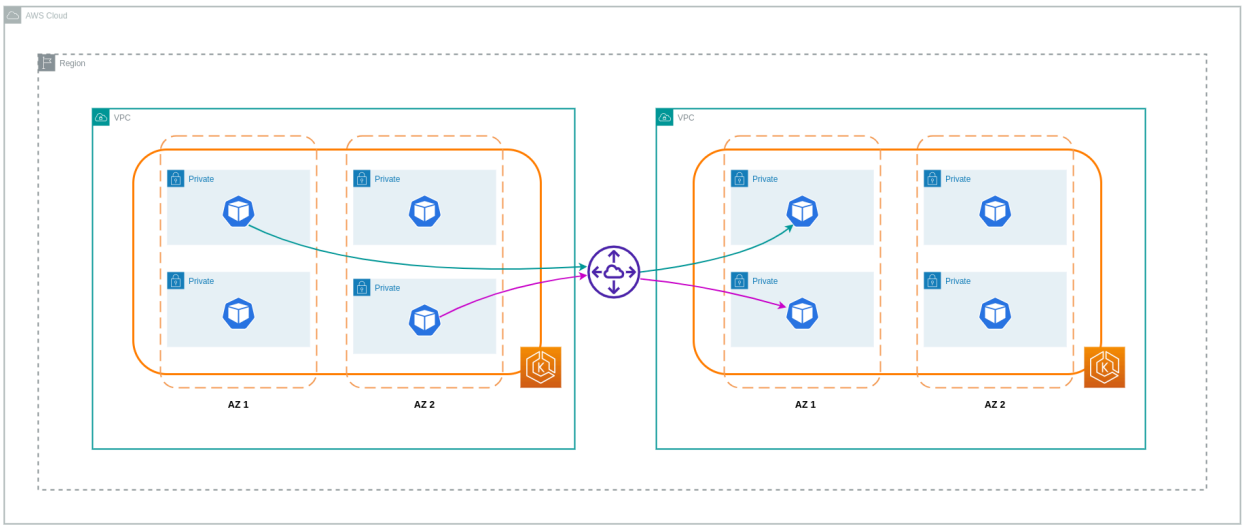
경우에 따라 서로 통신해야 하는 개별 VPCs(동일한 AWS 리전 내)에 워크로드가 있을 수 있습니다. 이는 트래픽이 각 VPCs. 이러한 통신은 퍼블릭 서브넷에 EC2 인스턴스, NAT 게이트웨이 또는 NAT 인스턴스와 같은 인프라 구성 요소를 배포하여 활성화할 수 있습니다. 그러나 이러한 구성 요소를 포함한 설정에는 VPCs 내부 및 외부의 데이터 처리/전송에 대한 요금이 발생합니다. 별도의 VPCs와 주고받는 트래픽이 AZs 간에 이동하는 경우 데이터 전송에 추가 요금이 부과됩니다. 아래 다이어그램은 NAT 게이트웨이와 인터넷 게이트웨이를 사용하여 서로 다른 VPCs의 워크로드 간에 통신을 설정하는 설정을 보여줍니다.



VPC 피어링 연결

이러한 사용 사례의 비용을 줄이기 위해 VPC 피어링을 사용할 수 있습니다. VPC 피어링 연결을 사용하면 동일한 AZ 내에 유지되는 네트워크 트래픽에 대한 데이터 전송 요금이 부과되지 않습니다. 트래픽이 AZs 통과하는 경우 비용이 발생합니다. 그러나 VPC 피어링 접근 방식은 동일한 AWS 리전 내의 개별 VPCs에 있는 워크로드 간의 비용 효율적인 통신에 권장됩니다. 그러나 전이적 네트워킹을 허용하지 않으므로 VPC 피어링은 주로 1:1 VPC 연결에 효과적입니다.

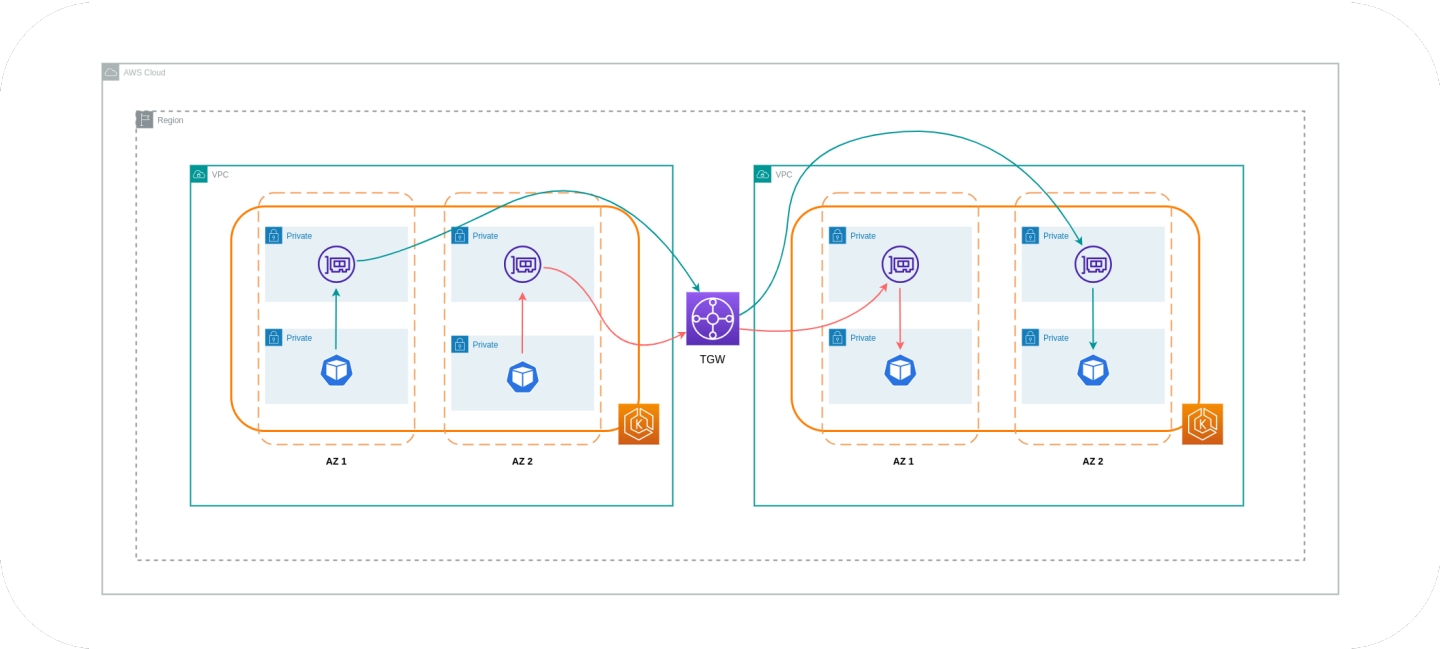
아래 다이어그램은 VPC 피어링 연결을 통한 워크로드 통신을 개괄적으로 나타낸 것입니다.



전이적 네트워킹 연결

이전 단원에서 언급했듯이 VPC 피어링 연결은 전이적 네트워킹 연결을 허용하지 않습니다. 3VPCs를 전이적 네트워킹 요구 사항에 연결하려면 Transit Gateway(TGW)를 사용해야 합니다. 이렇게 하면 VPC 피어링의 제한이나 여러 VPC 간에 여러 VPC 피어링 연결이 있는 것과 관련된 운영 오버헤드를 극복할 수 VPCs. 시간당 및 TGW로 전송된 데이터에 대한 요금이 청구됩니다. TGW를 통해 흐르는 AZ 간 트래픽과 관련된 대상 비용은 없습니다.

아래 다이어그램은 서로 다른 VPCs의 워크로드 간에 동일한 AWS 리전 내에서 TGW를 통해 흐르는 AZ 간 트래픽을 보여줍니다.



Service Mesh 사용

서비스 메시는 EKS 클러스터 환경에서 네트워크 관련 비용을 줄이는 데 사용할 수 있는 강력한 네트워킹 기능을 제공합니다. 그러나 채택할 경우 서비스 메시가 환경에 도입할 운영 작업과 복잡성을 신중하게 고려해야 합니다.

가용 영역으로 트래픽 제한

Istio의 지리 가중 분포 사용

Istio를 사용하면 라우팅이 발생한 후 트래픽에 네트워크 정책을 적용할 수 있습니다. 이는 지리 가중 분포와 같은 대상 규칙을 사용하여 수행됩니다. 이 기능을 사용하면 오리진에 따라 특정 대상으로 이동할 수 있는 트래픽의 가중치(백분율로 표시)를 제어할 수 있습니다. 이 트래픽의 소스는 외부(또는 퍼블릭)

로드 밸런서 또는 클러스터 자체 내의 포드에서 발생할 수 있습니다. 모든 포드 엔드포인트를 사용할 수 있게 되면 가중치 라운드 로빈 로드 밸런싱 알고리즘을 기반으로 로캘리티가 선택됩니다. 특정 엔드포인트가 비정상이거나 사용할 수 없는 경우 사용 가능한 엔드포인트의이 변경 사항을 반영하도록 로캘 가중치가 자동으로 조정됩니다.

> **Note**
>
> 로컬리티 가중치 기반 배포를 구현하기 전에 먼저 네트워크 트래픽 패턴과 대상 규칙 정책이 애플리케이션 동작에 미칠 수 있는 영향을 이해해야 합니다. 따라서 AWS X-Ray 또는 Jaeger와 같은 도구를 사용하여 분산 추적 메커니즘을 마련하는 것이 중요합니다.

위에 자세히 설명된 Istio 대상 규칙을 적용하여 로드 밸런서에서 EKS 클러스터의 포드로 가는 트래픽을 관리할 수도 있습니다. 고가용성 로드 밸런서(특히 Ingress Gateway)에서 트래픽을 수신하는 서비스에 로컬리티 가중치 기반 배포 규칙을 적용할 수 있습니다. 이러한 규칙을 사용하면 영역 오리진, 즉 이 경우 로드 밸런서에 따라 트래픽이 이동하는 양을 제어할 수 있습니다. 올바르게 구성하면 서로 다른 AZs의 포드 복제본에 트래픽을 균등하게 또는 무작위로 분산하는 로드 밸런서에 비해 송신 교차 영역 트래픽이 적게 발생합니다.

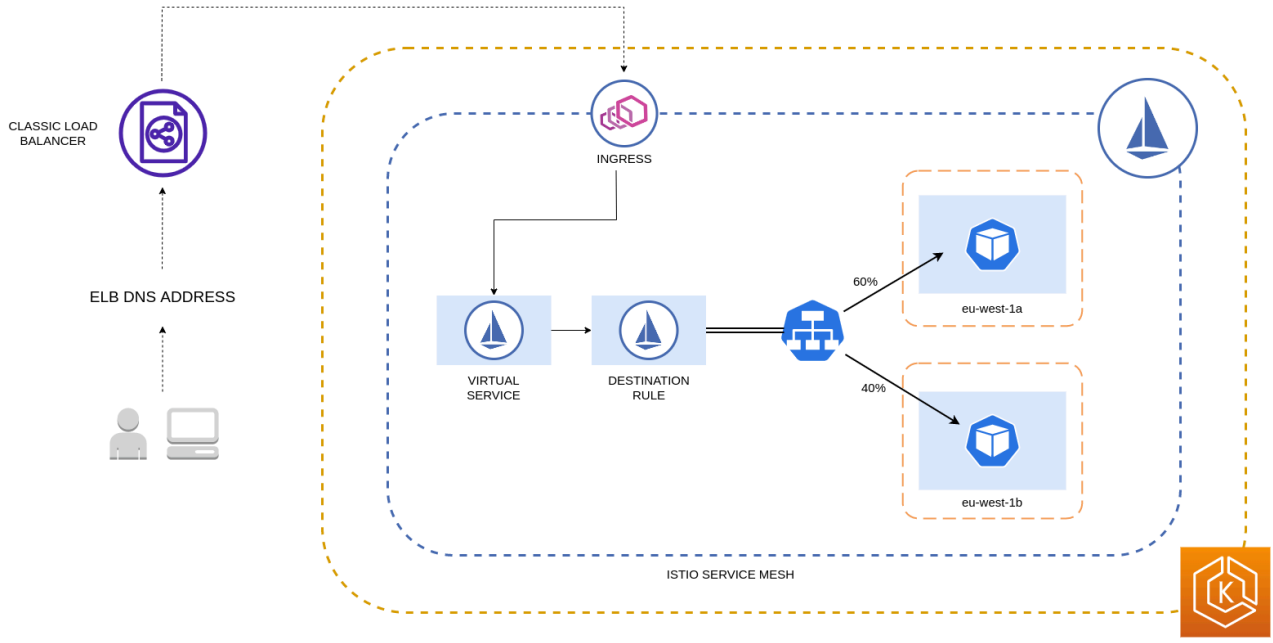
다음은 Istio에 있는 대상 규칙 리소스의 코드 블록 예제입니다. 아래에서 볼 수 있듯이이 리소스는 `eu-west-1` 리전에 있는 3개의 서로 다른 AZs에서 들어오는 트래픽에 대한 가중치 기반 구성을 지정합니다. 이러한 구성은 지정된 AZ에서 들어오는 트래픽의 대부분(이 경우 70%)이 시작된 동일한 AZ의 대상으로 프록시되어야 한다고 선언합니다.

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: express-test-dr
spec:
  host: express-test.default.svc.cluster.local
  trafficPolicy:
    loadBalancer:                    +
      localityLbSetting:
        distribute:
        - from: eu-west-1/eu-west-1a/  +
          to:
            "eu-west-1/eu-west-1a/_": 70
            "eu-west-1/eu-west-1b/_": 20
            "eu-west-1/eu-west-1c/_": 10
        - from: eu-west-1/eu-west-1b/_  +
```

```
        to:
          "eu-west-1/eu-west-1a/_": 20
          "eu-west-1/eu-west-1b/_": 70
          "eu-west-1/eu-west-1c/_": 10
      - from: eu-west-1/eu-west-1c/_  +
        to:
          "eu-west-1/eu-west-1a/_": 20
          "eu-west-1/eu-west-1b/_": 10
          "eu-west-1/eu-west-1c/*": 70**
  connectionPool:
    http:
      http2MaxRequests: 10
      maxRequestsPerConnection: 10
  outlierDetection:
    consecutiveGatewayErrors: 1
    interval: 1m
    baseEjectionTime: 30s
```

> **Note**
>
> 분산 대상일 수 있는 최소 가중치는 1%입니다. 그 이유는 기본 대상의 엔드포인트가 비정상이거나 사용할 수 없는 경우 장애 조치 리전과 영역을 유지하기 위한 것입니다.

아래 다이어그램은 eu-west-1 리전에 가용성이 높은 로드 밸런서가 있고 로캘 가중 분포가 적용되는 시나리오를 보여줍니다. 이 다이어그램의 대상 규칙 정책은 eu-west-1a에서 오는 트래픽의 60%를 동일한 AZ의 포드로 전송하도록 구성된 반면, eu-west-1a에서 오는 트래픽의 40%는 eu-west-1b의 포드로 이동해야 합니다.



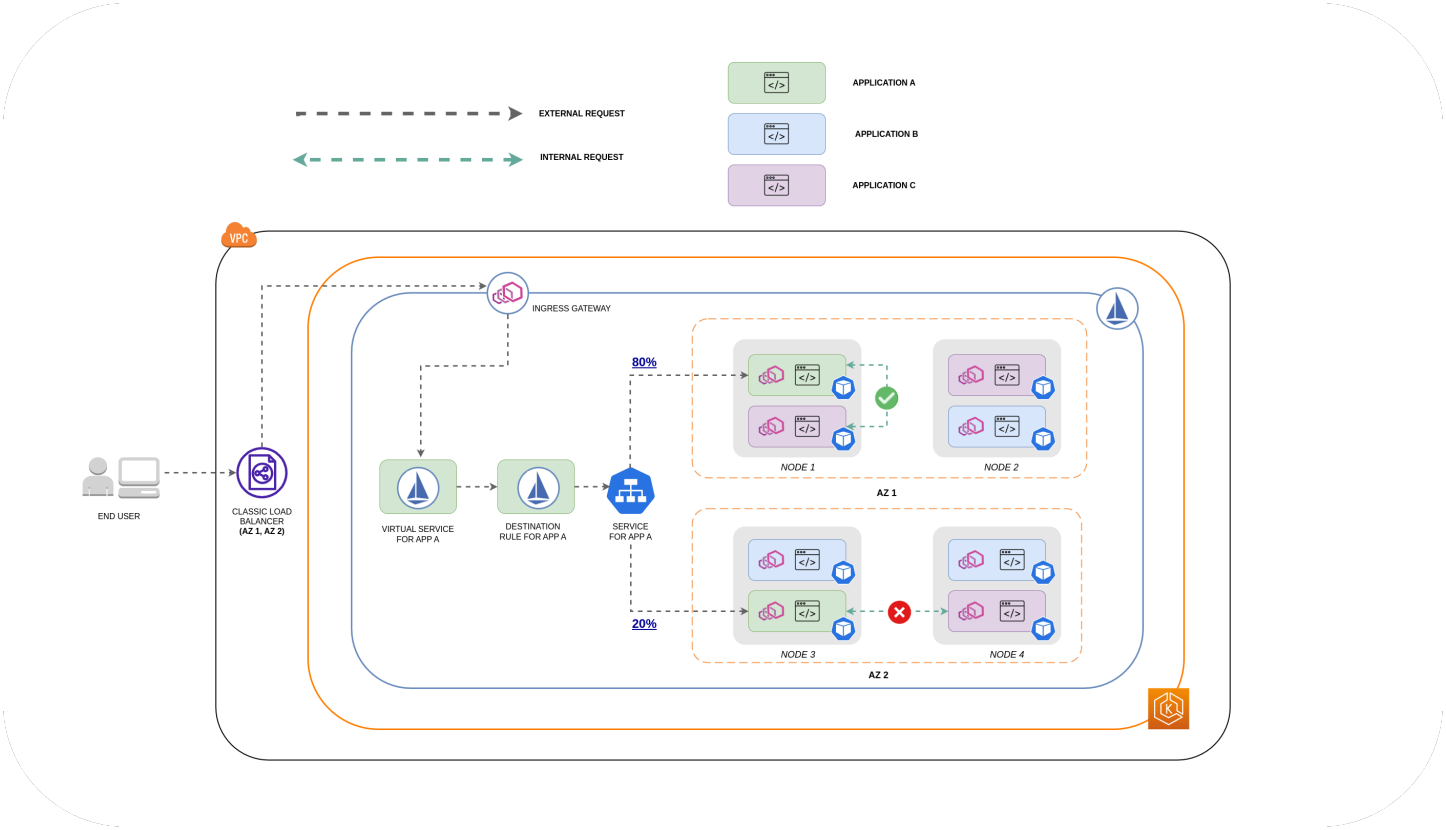
가용 영역 및 노드로 트래픽 제한

Istio에서 서비스 내부 트래픽 정책 사용

외부 수신 트래픽 및 포드 간 내부 트래픽과 관련된 네트워크 비용을 완화하기 위해 Istio의 대상 규칙과 Kubernetes Service 내부 트래픽 정책을 결합할 수 있습니다. Istio 대상 규칙을 서비스 내부 트래픽 정책과 결합하는 방법은 크게 3가지에 따라 달라집니다.

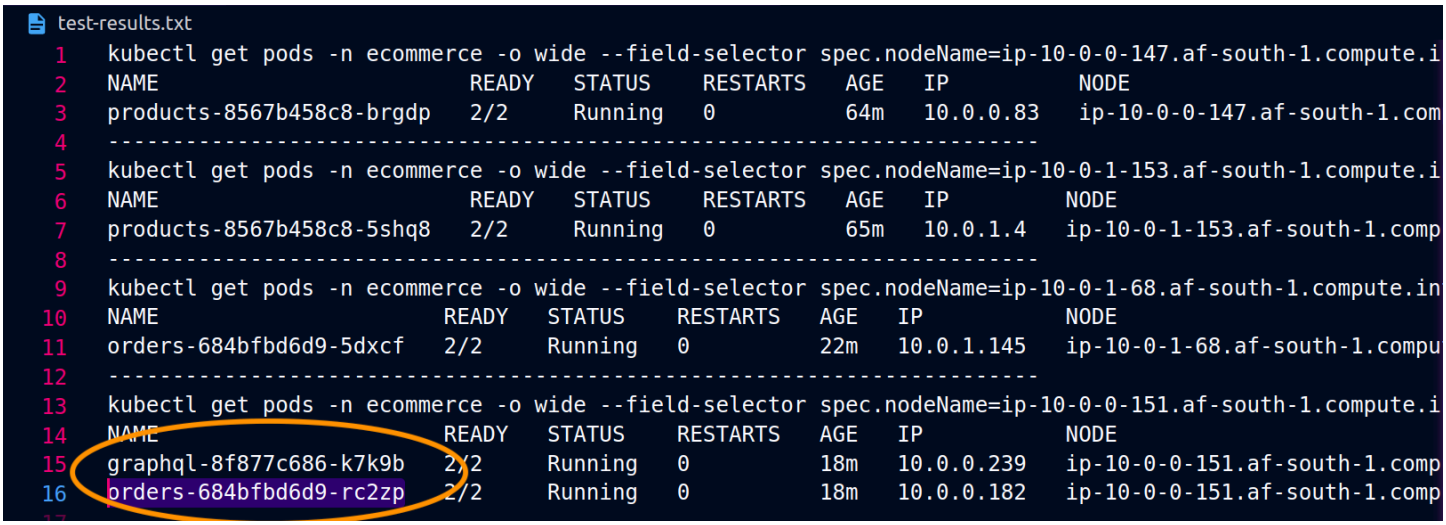
- 마이크로서비스의 역할
- 마이크로서비스 전반의 네트워크 트래픽 패턴
- Kubernetes 클러스터 토폴로지에 마이크로서비스를 배포하는 방법

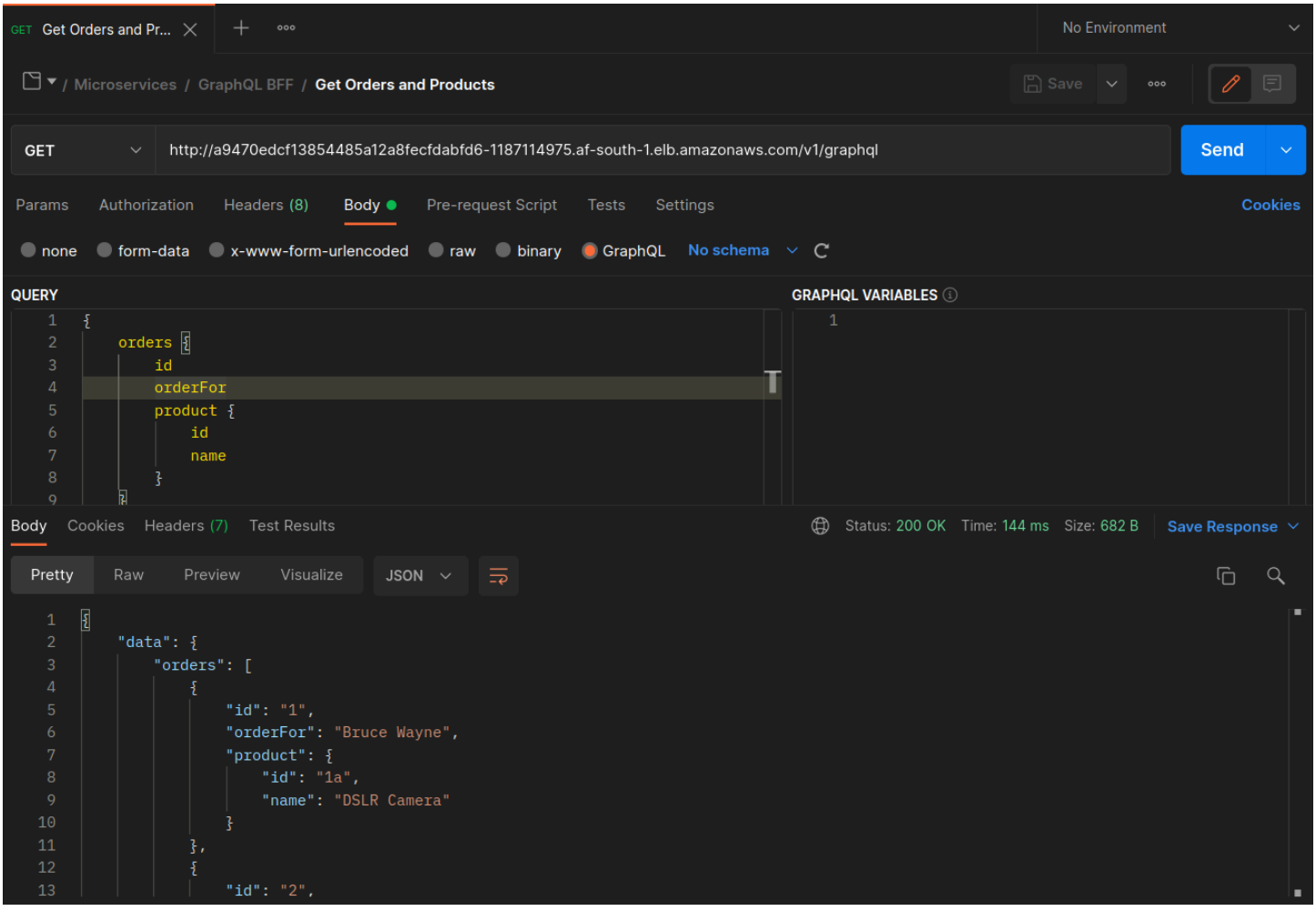
아래 다이어그램은 중첩된 요청의 경우 네트워크 흐름의 모습과 앞서 언급한 정책이 트래픽을 제어하는 방법을 보여줍니다.



1. 최종 사용자는 APP A에 요청을 하고, 그러면 APP C에 중첩된 요청을 합니다. 이 요청은 먼저 위의 다이어그램과 같이 AZ 1 및 AZ 2에 인스턴스가 있는 가용성이 높은 로드 밸런서로 전송됩니다.
2. 그런 다음 외부 수신 요청은 Istio Virtual Service에 의해 올바른 대상으로 라우팅됩니다.
3. 요청이 라우팅된 후 Istio 대상 규칙은 요청이 시작된 위치(AZs 따라 각 AZ로 전송되는 트래픽의 양을 제어합니다.
4. 그러면 트래픽이 APP A용 서비스로 이동하고 해당 포드 엔드포인트로 프록시됩니다. 다이어그램에서 볼 수 있듯이 수신 트래픽의 80%는 AZ 1의 포드 엔드포인트로 전송되고 수신 트래픽의 20%는 AZ 2로 전송됩니다.
5. 그런 다음 APP A는 APP C에 내부 요청을 합니다. APP C의 서비스에 내부 트래픽 정책이 활성화되어 있습니다(`internalTrafficPolicy`: Local`).
6. APP A(11월 1일)에서 APP C로의 내부 요청은 APP C에 사용할 수 있는 노드-로컬 엔드포인트로 인해 성공합니다.
7. APP A(NODE 3)에서 APP C로의 내부 요청은 APP C에 사용할 수 있는 노드-로컬 엔드포인트가 없기 때문에 실패합니다. 다이어그램에서 볼 수 있듯이 APP C에는 NODE 3에 복제본이 없습니다. **

아래 스크린샷은이 접근 방식의 라이브 예제에서 캡처됩니다. 첫 번째 스크린샷 세트는에 대한 성공적인 외부 요청graphql과에서 노드의 공동 위치 orders 복제본graphql으로의 성공적인 중첩 요청을 보여줍니다ip-10-0-0-151.af-south-1.compute.internal.





Istio를 사용하면 프록시가 인식하는 모든 [업스트림 클러스터](https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/intro/terminology) 및 엔드포인트의 통계를 확인하고 내보낼 수 있습니다. 이를 통해 네트워크 흐름과 워크로드 서비스 간의 분산 공유를 파악할 수 있습니다. 동일한 예제를 계속 진행하면서 `graphql` 프록시가 인식하는 `orders` 엔드포인트는 다음 명령을 사용하여 가져올 수 있습니다.

```
kubectl exec -it deploy/graphql -n ecommerce -c istio-proxy -- curl localhost:15000/clusters | grep orders
```

```
...
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**rq_error::0**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**rq_success::119**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**rq_timeout::0**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**rq_total::119**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**health_flags::healthy**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**region::af-south-1**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**zone::af-south-1b**
...
```

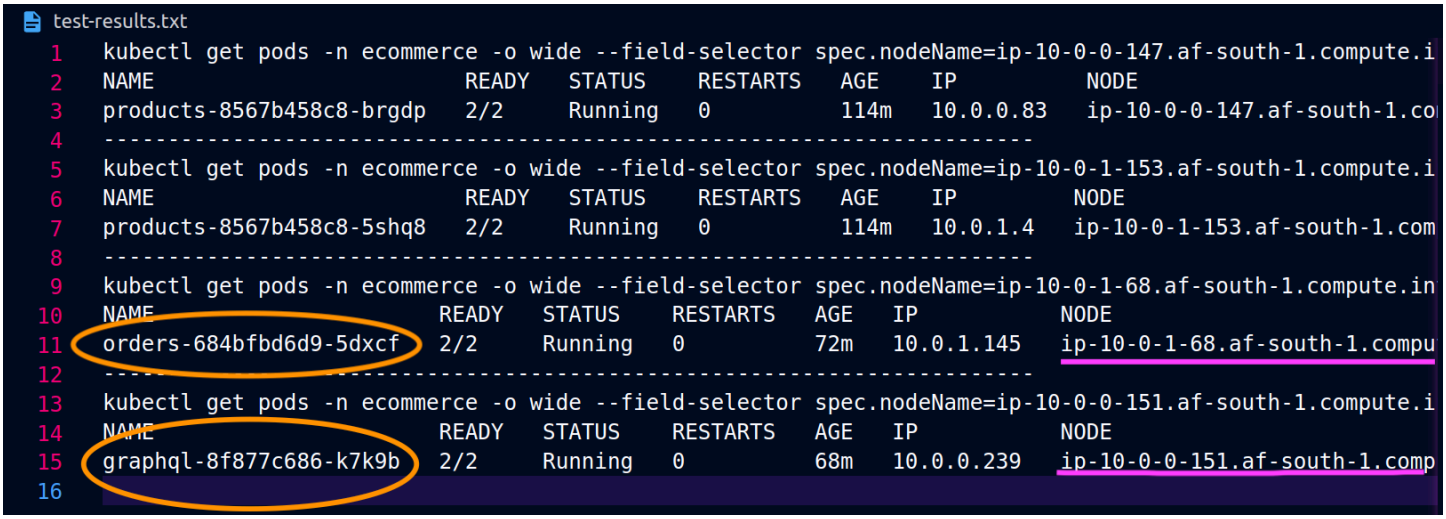
이 경우 `graphql` 프록시는 노드를 공유하는 복제본의 `orders` 엔드포인트만 인식합니다. 서비스 주문에서 `internalTrafficPolicy: Local` 설정을 제거하고 위와 같은 명령을 다시 실행하면 결과가 서로 다른 노드에 분산된 복제본의 모든 엔드포인트를 반환합니다. 또한 각 엔드포인트`rq_total`의를 검사하면 네트워크 배포가 비교적 균등하게 공유됩니다. 따라서 엔드포인트가 서로 다른 AZs에서 실행되는 업스트림 서비스와 연결된 경우 영역 간이 네트워크 배포로 인해 비용이 증가합니다.

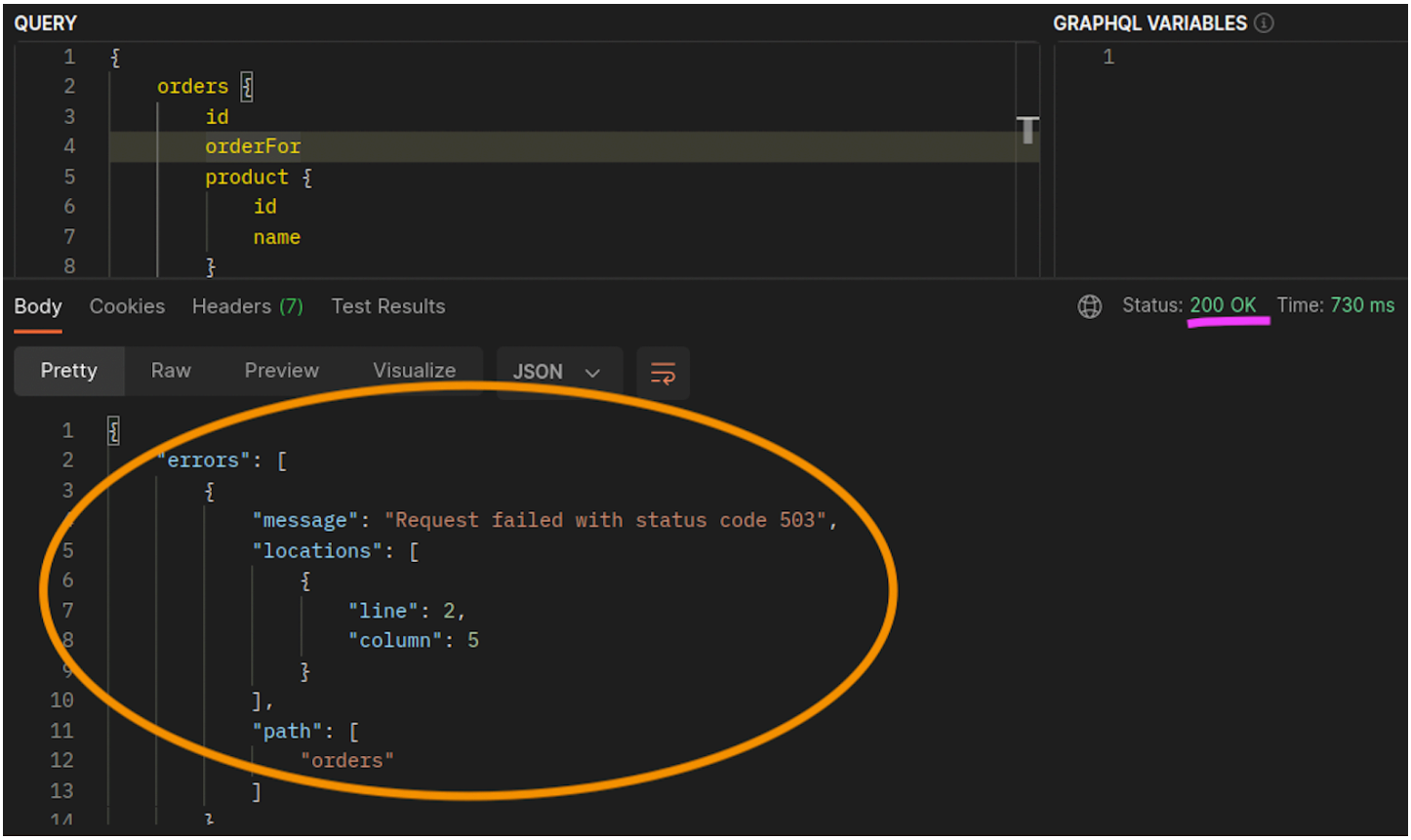
위의 이전 섹션에서 언급한 대로 포드 선호도를 사용하여 자주 통신하는 포드를 공동 배치할 수 있습니다.

```
...
spec:
...
  template:
    metadata:
      labels:
        app: graphql
        role: api
        workload: ecommerce
    spec:
      affinity:
```

```
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - orders
            topologyKey: "kubernetes.io/hostname"
      nodeSelector:
        managedBy: karpenter
        billing-team: ecommerce
...
```

graphql 및 orders 복제본graphql이 동일한 노드(ip-10-0-0-151.af-south-1.compute.internal)에 공존하지 않으면 아래 Postman 스크린샷의 200 response code에 명시된 대로에 대한 첫 번째 요청이 성공하는 반면,에 graphql 대한 두 번째 중첩 요청은 로 orders 실패합니다503 response code.





추가 리소스

- Istio를 사용하여 EKS의 지연 시간 및 데이터 전송 비용 해결
- Amazon Elastic Kubernetes Service에서 네트워크 트래픽에 대한 토폴로지 인식 힌트의 영향 탐색
- Amazon EKS Cross-AZ 포드에서 포드 네트워크 바이트로의 가시성 확보
- Istio를 사용하여 AZ 트래픽 최적화

- 토폴로지 인식 라우팅을 사용하여 AZ 트래픽 최적화
- 서비스 내부 트래픽 정책을 사용하여 Kubernetes 비용 및 성능 최적화
- Istio 및 Service 내부 트래픽 정책을 사용하여 Kubernetes 비용 및 성능 최적화
- Overview of Data Transfer Costs for Common Architectures
- AWS 컨테이너 서비스의 데이터 전송 비용 이해

스토리지

개요

단기 또는 장기적으로 데이터를 보존해야 하는 애플리케이션을 실행하려는 시나리오가 있습니다. 이러한 사용 사례의 경우 포드가 볼륨을 정의하고 탑재하여 컨테이너가 다양한 스토리지 메커니즘을 활용할 수 있도록 할 수 있습니다. Kubernetes는 임시 및 영구 스토리지를 위해 다양한 유형의 볼륨을 지원합니다. 스토리지 선택은 주로 애플리케이션 요구 사항에 따라 달라집니다. 각 접근 방식에는 비용에 영향을 미치며, 아래에 설명된 관행은 EKS 환경에서 일종의 스토리지가 필요한 워크로드에 대한 비용 효율성을 달성하는 데 도움이 됩니다.

임시 볼륨

임시 볼륨은 일시적인 로컬 볼륨이 필요하지만 재시작 후 데이터를 유지할 필요가 없는 애플리케이션을 위한 것입니다. 여기에는 스크래치 공간, 캐싱, 구성 데이터 및 보안 암호와 같은 읽기 전용 입력 데이터에 대한 요구 사항이 포함됩니다. Kubernetes 임시 볼륨에 대한 자세한 내용은 여기에서 확인할 수 있습니다. 대부분의 임시 볼륨(예: emptyDir, configMap, downwardAPI, secret, hostpath)은 로컬에 연결된 쓰기 가능한 디바이스(일반적으로 루트 디스크) 또는 RAM으로 지원되므로 가장 비용 효율적이고 성능이 뛰어난 호스트 볼륨을 선택하는 것이 중요합니다.

EBS 볼륨 사용

호스트 루트 볼륨으로 gp3로 시작하는 것이 좋습니다. Amazon EBS에서 제공하는 최신 범용 SSD 볼륨이며 gp2 볼륨에 비해 GB당 더 저렴한 가격(최대 20%)을 제공합니다.

Amazon EC2 인스턴스 스토어 사용

Amazon EC2 인스턴스 스토어는 EC2 인스턴스에 임시 블록 수준 스토리지를 제공합니다. EC2 인스턴스 스토어에서 제공하는 스토리지는 호스트에 물리적으로 연결된 디스크를 통해 액세스할 수 있습니다. Amazon EBS와 달리 인스턴스가 시작될 때만 인스턴스 스토어 볼륨을 연결할 수 있으며, 이러한 볼륨은 인스턴스 수명 동안에만 존재합니다. 분리하여 다른 인스턴스에 다시 연결할 수 없습니다.

Amazon EC2 인스턴스 스토어에 대한 자세한 내용은 여기에서 확인할 수 있습니다. 인스턴스 스토어 볼륨과 관련된 추가 요금은 없습니다. 이렇게 하면 EBS 볼륨이 큰 일반 EC2 인스턴스보다 (인스턴스 스토어 볼륨)가 더 비용 효율적입니다.

Kubernetes에서 로컬 스토어 볼륨을 사용하려면 볼륨을 포드 사양에서 HostPath로 마운트할 수 있도록 Amazon EC2 사용자 데이터를 사용하여 디스크를 분할, 구성 및 포맷해야 합니다. 또는 Local Persistent Volume Static Provisioner를 활용하여 로컬 스토리지 관리를 간소화할 수 있습니다. 로컬 영구 볼륨 정적 프로비저너를 사용하면 표준 Kubernetes PersistentVolumeClaim(PVC) 인터페이스를 통해 로컬 인스턴스 스토어 볼륨에 액세스할 수 있습니다. 또한 노드 선호도 정보가 포함된 PersistentVolumes(PVs)을 프로비저닝하여 포드를 올바른 노드로 예약합니다. Kubernetes PersistentVolumes를 사용하지만 EC2 인스턴스 스토어 볼륨은 본질적으로 일시적입니다. 임시 디스크에 기록된 데이터는 인스턴스 수명 동안에만 사용할 수 있습니다. 인스턴스가 종료되면 데이터도 마찬가지입니다. 자세한 내용은이 블로그를 참조하세요.

Amazon EC2 인스턴스 스토어 볼륨을 사용할 때 총 IOPS 한도는 호스트와 공유되며 특정 호스트에 포드를 바인딩합니다. Amazon EC2 인스턴스 스토어 볼륨을 채택하기 전에 워크로드 요구 사항을 철저히 검토해야 합니다.

영구 볼륨

Kubernetes는 일반적으로 상태 비저장 애플리케이션을 실행하는 것과 관련이 있습니다. 그러나 한 요청에서 다음 요청으로 영구 데이터 또는 정보를 보존해야 하는 마이크로서비스를 실행하려는 시나리오가 있습니다. 데이터베이스는 이러한 사용 사례의 일반적인 예입니다. 그러나 포드와 포드 내의 컨테이너 또는 프로세스는 본질적으로 일시적입니다. 포드의 수명을 초과하여 데이터를 유지하려면 PVs 사용하여 포드와 독립적인 특정 위치에서 스토리지에 대한 액세스를 정의할 수 있습니다. PVs와 관련된 비용은 사용 중인 스토리지 유형과 애플리케이션이 이를 사용하는 방식에 따라 크게 달라집니다.

여기에 나열된 Amazon EKS에서 Kubernetes PVs 지원하는 다양한 유형의 스토리지 옵션이 있습니다. 아래에서 다루는 스토리지 옵션은 Amazon EBS, Amazon EFS, Amazon FSx for Lustre, Amazon FSx for NetApp ONTAP입니다.

Amazon Elastic Block Store(EBS) 볼륨

Amazon EBS 볼륨을 Kubernetes PVs을 제공할 수 있습니다. 이는 임의 읽기 및 쓰기와 길고 지속적인 읽기 및 쓰기를 수행하는 처리량 집약적인 애플리케이션에 의존하는 데이터베이스에 적합합니다. Amazon Elastic Block Store 컨테이너 스토리지 인터페이스(CSI) 드라이버를 사용하면 Amazon EKS 클러스터가 영구 볼륨에 대한 Amazon EBS 볼륨의 수명 주기를 관리할 수 있습니다. 컨테이너 스토리지 인터페이스는 Kubernetes와 스토리지 시스템 간의 상호 작용을 활성화하고 용이하게 합니다. CSI 드라이버가 EKS 클러스터에 배포되면 영구 볼륨(PVs), 영구 볼륨 클레임(PVCs), 스토리지 클래스

(SCs. 이 링크는 Amazon EBS CSI 드라이버를 사용하여 Amazon EBS 볼륨과 상호 작용하는 방법에 대한 실제 예제를 제공합니다.

올바른 볼륨 선택

가격과 성능 간에 적절한 균형을 제공하므로 최신 세대의 블록 스토리지(gp3)를 사용하는 것이 좋습니다. 또한 추가 블록 스토리지 용량을 프로비저닝할 필요 없이 볼륨 크기와 관계없이 볼륨 IOPS 및 처리량을 조정할 수 있습니다. 현재 gp2 볼륨을 사용하는 경우 gp3 볼륨으로 마이그레이션하는 것이 좋습니다. 블로그 게시물 gp2에서 gp3 EBS 볼륨으로 Amazon EKS 클러스터 마이그레이션에서는 애플리케이션 가동 중지가 필요한 CSI 볼륨 스냅샷 기능을 사용하여 백업 및 복원을 통해 Amazon EKS 클러스터의 gp2에서 gp3로 마이그레이션하는 방법을 설명합니다.

Amazon EBS를 사용하면 볼륨 크기, IOPS 및 처리량과 같은 볼륨 특성을 온라인으로 변경할 수 있습니다. 이 기능을 사용하면이 블로그에 설명된 대로 EBS CSI 드라이버 v1.19.0+를 사용하거나 여기에 설명된 VolumeAttributesClass API를 사용하여 Amazon EKS v1.31 및 EBS CSI 드라이버 1.35로 시작하는 PVC 주석을 사용하여 애플리케이션 가동 중지 없이 gp3의 gp2에서 마이그레이션할 수 있습니다.

더 높은 성능이 필요하고 단일 gp3 볼륨이 지원할 수 있는 것보다 더 큰 볼륨이 필요한 애플리케이션이 있는 경우 io2 블록 익스프레스를 사용하는 것이 좋습니다. 이러한 유형의 스토리지는 SAP HANA 또는 지연 시간이 짧은 기타 대규모 데이터베이스와 같은 가장 크고 가장 많은 I/O 집약적이며 미션 크리티컬한 배포에 적합합니다. 인스턴스의 EBS 성능은 인스턴스의 성능 제한에 따라 제한되므로 모든 인스턴스가 io2 블록 Express 볼륨을 지원하는 것은 아닙니다. 이 문서에서 지원되는 인스턴스 유형 및 기타 고려 사항을 확인할 수 있습니다.

단일 gp3 볼륨은 최대 16,000 IOPS, 최대 1,000MiB/s 처리량, 최대 16TiB를 지원할 수 있습니다. 최대 256,000 IOPS, 4,000MiB/s, 처리량 및 64TiB를 제공하는 최신 세대의 프로비저닝된 IOPS SSD 볼륨입니다.

이러한 옵션 중에서 애플리케이션의 요구 사항에 맞게 스토리지 성능과 비용을 가장 잘 조정해야 합니다.

시간 경과에 따른 모니터링 및 최적화

애플리케이션의 기준 성능을 이해하고 선택한 볼륨에 대해 모니터링하여 애플리케이션이 요구 사항/기대치를 충족하는지 또는 과다 프로비저닝되었는지(예: 프로비저닝된 IOPS가 완전히 활용되지 않는 시나리오) 확인하는 것이 중요합니다.

대용량 볼륨을 처음부터 할당하는 대신 데이터를 누적할 때 볼륨 크기를 점진적으로 늘릴 수 있습니다. Amazon Elastic Block Store CSI 드라이버(aws-ebs-csi-driver)의 볼륨 크기 조정 기능을 사용하여 볼륨 크기를 동적으로 조정할 수 있습니다. EBS 볼륨 크기만 늘릴 수 있다는 점에 유의하세요.

매달린 EBS 볼륨을 식별하고 제거하려면 AWS 신뢰할 수 있는 어드바이저의 비용 최적화 범주를 사용할 수 있습니다. 이 기능을 사용하면 연결되지 않은 볼륨 또는 일정 기간 동안 쓰기 활동이 매우 적은 볼륨을 식별할 수 있습니다. 라이브 Kubernetes 클러스터를 스캔하고 배포된 리소스 및 구성과 관련된 잠재적 문제를 보고하는 Popeye라는 클라우드 네이티브 오픈 소스 읽기 전용 도구가 있습니다. 예를 들어 미사용 PVs 및 PVCs를 스캔하고 경계가 있는지 또는 볼륨 마운트 오류가 있는지 확인할 수 있습니다.

모니터링에 대한 자세한 내용은 EKS 비용 최적화 관찰성 가이드를 참조하세요.

고려할 수 있는 또 다른 옵션은 AWS Compute Optimizer Amazon EBS 볼륨 권장 사항입니다. 이 도구는 최적의 볼륨 구성과 필요한 올바른 성능 수준을 자동으로 식별합니다. 예를 들어 지난 14일 동안의 최대 사용률을 기반으로 프로비저닝된 IOPS, 볼륨 크기 및 EBS 볼륨 유형과 관련된 최적의 설정에 사용할 수 있습니다. 또한 권장 사항에서 파생된 잠재적 월별 비용 절감액도 정량화합니다. 자세한 내용은이 블로그를 참조하세요.

백업 보관 정책

point-in-time 스냅샷을 생성하여 Amazon EBS 볼륨에 데이터를 백업할 수 있습니다. Amazon EBS CSI 드라이버는 볼륨 스냅샷을 지원합니다. 여기에 설명된 단계를 사용하여 스냅샷을 생성하고 EBS PV를 복원하는 방법을 배울 수 있습니다.

후속 스냅샷은 증분 백업입니다. 즉, 가장 최근 스냅샷 이후에 변경된 디바이스의 블록만 저장됩니다. 그러면 스냅샷을 만드는 데 필요한 시간이 최소화되며 데이터를 복제하지 않으므로 스토리지 비용이 절약됩니다. 그러나 적절한 보존 정책 없이 이전 EBS 스냅샷 수를 늘리면 대규모로 운영할 때 예상치 못한 비용이 발생할 수 있습니다. AWS API를 통해 Amazon EBS 볼륨을 직접 백업하는 경우 Amazon Elastic Block Store(EBS) 스냅샷 및 EBS 지원 Amazon Machine Image(AMI)에 대한 자동화된 정책 기반 수명 주기 관리 솔루션을 제공하는 Amazon Data Lifecycle ManagerAMIs. 콘솔을 사용하면 EBS 스냅샷 및 AMIs.

> **Note**
>
> 현재 Amazon EBS CSI 드라이버를 통해 Amazon DLM을 사용할 수 있는 방법은 없습니다.

Kubernetes 환경에서는 Velero라는 오픈 소스 도구를 활용하여 EBS 영구 볼륨을 백업할 수 있습니다. 백업 만료 작업을 예약할 때 TTL 플래그를 설정할 수 있습니다. 다음은 Velero의 예제 가이드입니다.

Amazon Elastic File System(EFS)

Amazon Elastic File System(EFS)은 광범위한 워크로드 및 애플리케이션에 대해 표준 파일 시스템 인터페이스 및 파일 시스템 의미 체계를 사용하여 파일 데이터를 공유할 수 있는 서버리스의 완전 탄력적 파일 시스템입니다. 워크로드 및 애플리케이션의 예로는 Wordpress 및 Drupal, JIRA 및 Git과 같은 개발자 도구, Jupyter와 같은 공유 노트북 시스템, 홈 디렉터리 등이 있습니다.

Amazon EFS의 주요 이점 중 하나는 여러 노드와 여러 가용 영역에 분산된 여러 컨테이너에서 탑재할 수 있다는 것입니다. 또 다른 이점은 사용하는 스토리지에 대해서만 비용을 지불한다는 것입니다. 파일을 추가 및 제거하면 EFS 파일 시스템이 자동으로 확장 및 축소되므로 용량 계획이 필요하지 않습니다.

Kubernetes에서 Amazon EFS를 사용하려면 Amazon Elastic File System Container Storage Interface(CSI) 드라이버인 aws-efs-csi-driver 사용해야 합니다. 현재 드라이버는 액세스 포인트를 동적으로 생성할 수 있습니다. 그러나 Amazon EFS 파일 시스템을 먼저 프로비저닝하고 Kubernetes 스토리지 클래스 파라미터에 대한 입력으로 제공해야 합니다.

올바른 EFS 스토리지 클래스 선택

Amazon EFS는 네 가지 스토리지 클래스를 제공합니다.

두 가지 표준 스토리지 클래스:

- Amazon EFS Standard
- Amazon EFS Standard-Infrequent Access(EFS Standard-IA)

두 개의 단일 영역 스토리지 클래스:

- Amazon EFS One Zone
- Amazon EFS One Zone-Infrequent Access(EFS One Zone-IA)

Infrequent Access(IA) 스토리지 클래스는 매일 액세스하지 않는 파일에 대해 비용이 최적화됩니다. Amazon EFS 수명 주기 관리를 사용하면 수명 주기 정책 기간(7, 14, 30, 60 또는 90일) 동안 액세스하지 않은 파일을 IA 스토리지 클래스로 이동할 수 있으므로 EFS Standard 및 EFS One Zone 스토리지 클래스에 비해 스토리지 비용을 각각 최대 92% 절감할 수 있습니다.

EFS Intelligent-Tiering을 사용하면 수명 주기 관리가 파일 시스템의 액세스 패턴을 모니터링하고 파일을 가장 최적의 스토리지 클래스로 자동으로 이동합니다.

> **Note**
>
> aws-efs-csi-driver는 현재 스토리지 클래스 변경, 수명 주기 관리 또는 Intelligent-Tiering을 제어할 수 없습니다. AWS 콘솔 또는 EFS APIs.

> **Note**
>
> aws-efs-csi-driver는 창 기반 컨테이너 이미지와 호환되지 않습니다.

> **Note**
>
> 파일 시스템의 크기에 비례하는 양의 메모리를 소비하는 DiskUsage 함수로 인해 vol-metrics-opt-in(볼륨 지표를 내보내기 위해)이 활성화된 경우 알려진 메모리 문제가 있습니다. 현재는 메모리를 너무 많이 소비하지 않도록 대용량 파일 시스템에서 `--vol-metrics-opt-in` 옵션을 비활성화하는 것이 좋습니다.vol-metrics-opt-in 자세한 내용은 github 문제 링크를 참조하세요.

Amazon FSx for Lustre

Lustre는 최대 수백 GB/s의 처리량과 작업당 밀리초 미만의 지연 시간이 필요한 워크로드에 일반적으로 사용되는 고성능 병렬 파일 시스템입니다. 기계 학습 훈련, 금융 모델링, HPC 및 비디오 처리와 같은 시나리오에 사용됩니다. Amazon FSx for Lustre는 Amazon S3와 원활하게 통합된 확장성과 성능을 갖춘 완전관리형 공유 스토리지를 제공합니다.

Amazon EKS의 FSx for Lustre CSI 드라이버 또는 AWS의 자체 관리형 Kubernetes 클러스터를 사용하여 FSx for Lustre에서 지원하는 Kubernetes 영구 스토리지 볼륨을 사용할 수 있습니다. 자세한 내용과 예제는 Amazon EKS 설명서를 참조하세요.

Amazon S3에 대한 링크

Amazon S3에 있는 내구성이 뛰어난 장기 데이터 리포지토리를 FSx for Lustre 파일 시스템과 연결하는 것이 좋습니다. 연결된 대용량 데이터 세트는 필요에 따라 Amazon S3에서 FSx for Lustre 파일 시스템으로 지연 로드됩니다. 분석 및 결과를 S3로 다시 실행한 다음 Lustre 파일 시스템을 삭제할 수도 있습니다.

올바른 배포 및 스토리지 옵션 선택

FSx for Lustre는 다양한 배포 옵션을 제공합니다. 첫 번째 옵션은 스크래치라고 하며 데이터를 복제하지 않고, 두 번째 옵션은 영구적이라고 하며, 이름에서 알 수 있듯이 데이터가 유지됩니다.

첫 번째 옵션(스크래치)을 사용하여 일시적인 단기 데이터 처리 비용을 줄일 수 있습니다. 영구 배포 옵션은 AWS 가용 영역 내에서 데이터를 자동으로 복제하는 장기 스토리지를 위해 설계되었습니다. 또한 SSD 및 HDD 스토리지를 모두 지원합니다.

FSx for lustre 파일 시스템의 Kubernetes StorageClass의 파라미터에서 원하는 배포 유형을 구성할 수 있습니다. 다음은 샘플 템플릿을 제공하는 링크입니다.

> **Note**
>
> 지연 시간에 민감한 워크로드 또는 최고 수준의 IOPS/처리량이 필요한 워크로드의 경우 SSD 스토리지를 선택해야 합니다. 지연 시간에 민감하지 않은 처리량 중심 워크로드의 경우 HDD 스토리지를 선택해야 합니다.

데이터 압축 활성화

"LZ4"를 데이터 압축 유형으로 지정하여 파일 시스템에서 데이터 압축을 활성화할 수도 있습니다. 활성화되면 새로 작성된 모든 파일은 디스크에 기록되기 전에 FSx for Lustre에서 자동으로 압축되고 읽을 때 압축되지 않습니다. LZ4 데이터 압축 알고리즘은 손실이 없으므로 압축된 데이터에서 원본 데이터를 완전히 재구성할 수 있습니다.

FSx for lustre 파일 시스템의 Kubernetes StorageClass의 파라미터에서 데이터 압축 유형을 LZ4로 구성할 수 있습니다. 값이 기본값인 NONE으로 설정되면 압축이 비활성화됩니다. 이 링크는 샘플 템플릿을 제공합니다.

> **Note**
>
> Amazon FSx for Lustre는 창 기반 컨테이너 이미지와 호환되지 않습니다.

Amazon FSx for NetApp ONTAP

Amazon FSx for NetApp ONTAP은 NetApp의 ONTAP 파일 시스템에 구축된 완전 관리형 공유 스토리지입니다. FSx for ONTAP은 AWS 또는 온프레미스에서 실행되는 Linux, Windows 및 macOS 컴퓨팅

인스턴스에서 광범위하게 액세스할 수 있는 기능이 풍부하고 빠르고 유연한 공유 파일 스토리지를 제공합니다.

Amazon FSx for NetApp ONTAP은 1/프라이머리 티어와 2/용량 풀 티어의 두 가지 스토리지 티어를 지원합니다.

기본 계층은 지연 시간에 민감한 활성 데이터를 위한 프로비저닝된 고성능 SSD 기반 계층입니다. 완전 탄력적 용량 풀 계층은 자주 액세스하지 않는 데이터에 대해 비용 최적화되고, 데이터가 계층화되면 자동으로 확장되며, 사실상 무제한 페타바이트의 용량을 제공합니다. 용량 풀 스토리지에서 데이터 압축 및 중복 제거를 활성화하고 데이터가 소비하는 스토리지 용량을 추가로 줄일 수 있습니다. NetApp의 기본 정책 기반 FabricPool 기능은 데이터 액세스 패턴을 지속적으로 모니터링하여 스토리지 계층 간에 양방향으로 데이터를 자동으로 전송하여 성능과 비용을 최적화합니다.

NetApp의 Astra Trident는 Amazon EKS 클러스터가 Amazon FSx for NetApp ONTAP 파일 시스템이 지원하는 영구 볼륨 PVs의 수명 주기를 관리할 수 있도록 CSI 드라이버를 사용하여 동적 스토리지 오케스트레이션을 제공합니다. 시작하려면 Astra Trident 설명서의 Amazon FSx for NetApp ONTAP에서 Astra Trident 사용을 참조하세요.

기타 고려 사항

컨테이너 이미지 크기 최소화

컨테이너가 배포되면 컨테이너 이미지가 호스트에 여러 계층으로 캐시됩니다. 이미지 크기를 줄이면 호스트에 필요한 스토리지 양을 줄일 수 있습니다.

처음부터 스크래치 이미지 또는 무분별한 컨테이너 이미지(애플리케이션 및 런타임 종속성만 포함)와 같은 축소된 기본 이미지를 사용하면 공격 표면적 감소 및 이미지 풀 타임 단축과 같은 다른 보조 이점 외에도 스토리지 비용을 절감할 수 있습니다.

또한 Slim.ai 같은 오픈 소스 도구를 사용하면 최소한의 이미지를 쉽고 안전하게 생성할 수 있습니다.

패키지, 도구, 애플리케이션 종속성, 라이브러리의 여러 계층이 컨테이너 이미지 크기를 쉽게 팽창시킬 수 있습니다. 다단계 빌드를 사용하면 최종 이미지에서 필요하지 않은 모든 것을 제외하고 한 단계에서 다른 단계로 아티팩트를 선택적으로 복사할 수 있습니다. 여기에서 더 많은 이미지 빌드 모범 사례를 확인할 수 있습니다.

또 다른 고려 사항은 캐시된 이미지를 얼마나 오래 유지할지입니다. 일정량의 디스크를 사용할 때 이미지 캐시에서 오래된 이미지를 정리할 수 있습니다. 이렇게 하면 호스트 작업에 충분한 공간이 있는지 확인하는 데 도움이 됩니다. 기본적으로 kubelet은 5분마다 미사용 이미지에 대해 가비지 수집을 수행하고 1분마다 미사용 컨테이너에 대해 가비지 수집을 수행합니다.

미사용 컨테이너 및 이미지 가비지 수집에 대한 옵션을 구성하려면 구성 파일을 사용하여 kubelet을 조정하고 *KubeletConfiguration* 리소스 유형을 사용하여 가비지 수집과 관련된 파라미터를 변경합니다.

이에 대한 자세한 내용은 Kubernetes 설명서에서 확인할 수 있습니다.

관찰성

소개

관찰성 도구를 사용하면 워크로드를 효율적으로 감지, 해결 및 조사할 수 있습니다. EKS 사용이 증가함에 따라 원격 측정 데이터의 비용이 자연스럽게 증가합니다. 때로는 운영 요구 사항의 균형을 맞추고 비즈니스에 중요한 사항을 측정하며 관찰성 비용을 확인하는 것이 어려울 수 있습니다. 이 가이드는 관찰성의 세 가지 원칙인 로그, 지표 및 트레이스에 대한 비용 최적화 전략에 중점을 둡니다. 이러한 각 모범 사례는 조직의 최적화 목표에 맞게 독립적으로 적용할 수 있습니다.

로깅

로깅은 클러스터의 애플리케이션을 모니터링하고 문제를 해결하는 데 중요한 역할을 합니다. 로깅 비용을 최적화하는 데 사용할 수 있는 몇 가지 전략이 있습니다. 아래 나열된 모범 사례 전략에는 로그 보존 정책을 검사하여 로그 데이터 보관 기간에 대한 세분화된 제어를 구현하고, 중요도에 따라 로그 데이터를 다양한 스토리지 옵션으로 전송하고, 로그 필터링을 활용하여 저장된 로그 메시지의 유형을 좁히는 것이 포함됩니다. 로그 원격 측정을 효율적으로 관리하면 환경 비용을 절감할 수 있습니다.

EKS 제어 플레인

컨트롤 플레인 로그 최적화

Kubernetes 컨트롤 플레인은 클러스터를 관리하는 구성 요소 세트이며 이러한 구성 요소는 Amazon CloudWatch의 로그 그룹에 로그 스트림으로 다양한 유형의 정보를 전송합니다. 모든 컨트롤 플레인 로그 유형을 활성화하면 이점이 있지만 각 로그의 정보와 모든 로그 원격 측정을 저장하는 데 드는 관련 비용을 알고 있어야 합니다. 클러스터에서 Amazon CloudWatch Logs로 전송된 로그에 대한 표준 CloudWatch Logs 데이터 수집 및 스토리지 비용에 대한 요금이 부과됩니다. Amazon CloudWatch 활성화하기 전에 각 로그 스트림이 필요한지 평가합니다.

예를 들어 비프로덕션 클러스터에서는 API 서버 로그와 같은 특정 로그 유형을 선택적으로 분석용으로만 활성화하고 나중에 비활성화합니다. 그러나 이벤트를 재현할 수 없고 문제를 해결하려면 더 많은 로그 정보가 필요한 프로덕션 클러스터의 경우 모든 로그 유형을 활성화할 수 있습니다. 추가 컨트롤 플레인 비용 최적화 구현 세부 정보는이 블로그 게시물에 나와 있습니다.

S3로 로그 스트리밍

또 다른 비용 최적화 모범 사례는 CloudWatch Logs 구독을 통해 제어 영역 로그를 S3로 스트리밍하는 것입니다. CloudWatch Logs 구독을 활용하면 로그를 S3에 선택적으로 전달할 수 있으므로 CloudWatch에서 로그를 무기한 보관하는 것보다 더 비용 효율적인 장기 스토리지를 제공합니다. 예를 들어 프로덕션 클러스터의 경우 중요한 로그 그룹을 생성하고 구독을 활용하여 15일 후에 이러한 로그를 S3로 스트리밍할 수 있습니다. 이렇게 하면 분석을 위해 로그에 빠르게 액세스할 수 있을 뿐만 아니라 로그를 보다 비용 효율적인 스토리지로 이동하여 비용을 절감할 수 있습니다.

> **Important**
>
> 9/5/2023부터 EKS 로그는 Amazon CloudWatch Logs에서 벤딩 로그로 분류됩니다. 벤딩 로그는 AWS 서비스가 고객을 대신하여 기본적으로 게시하고 볼륨 할인 요금으로 사용할 수 있는 특정 AWS 서비스 로그입니다. Amazon CloudWatch 요금 페이지를 방문하여 벤딩 로그 요금에 대해 자세히 알아보세요.

EKS 데이터 영역

로그 보존

Amazon CloudWatch의 기본 보존 정책은 로그를 무기한으로 유지하고 만료되지 않도록 하여 AWS 리전에 적용되는 스토리지 비용을 발생시키는 것입니다. 스토리지 비용을 줄이기 위해 워크로드 요구 사항에 따라 각 로그 그룹에 대한 보존 정책을 사용자 지정할 수 있습니다.

개발 환경에서는 긴 보존 기간이 필요하지 않을 수 있습니다. 그러나 프로덕션 환경에서는 문제 해결, 규정 준수 및 용량 계획 요구 사항을 충족하도록 더 긴 보존 정책을 설정할 수 있습니다. 예를 들어, 성수기 동안 전자 상거래 애플리케이션을 실행하는 경우 시스템이 더 많은 부하를 받고 즉시 눈에 띄지 않을 수 있는 문제가 발생할 수 있는 경우 자세한 문제 해결 및 이벤트 후 분석을 위해 더 긴 로그 보존을 설정해야 합니다.

AWS CloudWatch 콘솔 또는 AWS API에서 각 로그 그룹에 따라 1일에서 10년까지 보존 기간을 구성할 수 있습니다. 유연한 보존 기간이 있으면 로그 스토리지 비용을 절감하는 동시에 중요한 로그를 유지할 수 있습니다.

로그 스토리지 옵션

스토리지는 관찰성 비용의 큰 동인이므로 로그 스토리지 전략을 최적화하는 것이 중요합니다. 전략은 성능과 확장성을 유지하면서 워크로드 요구 사항에 부합해야 합니다. 로그 저장 비용을 줄이기 위한 한 가지 전략은 AWS S3 버킷과 다양한 스토리지 티어를 활용하는 것입니다.

로그를 S3로 직접 전달

Cloudwatch 대신 개발 환경과 같이 덜 중요한 로그를 S3로 직접 전달하는 것이 좋습니다. 이는 로그 스토리지 비용에 즉각적인 영향을 미칠 수 있습니다. 한 가지 옵션은 Fluentbit을 사용하여 로그를 S3로 직접 전달하는 것입니다. FluentBit가 보관을 위해 컨테이너 로그를 전송하는 대상인 [OUTPUT] 섹션에서 이를 정의합니다. 여기에서 추가 구성 파라미터를 검토합니다.

```
[OUTPUT]
        Name eks_to_s3
        Match application.*
        bucket $S3_BUCKET name
        region us-east-2
        store_dir /var/log/fluentbit
        total_file_size 30M
        upload_timeout 3m
```

단기 분석을 위해서만 CloudWatch로 로그 전달

데이터에 대한 즉각적인 분석을 수행해야 할 수 있는 프로덕션 환경과 같은 더 중요한 로그의 경우 로그를 CloudWatch에 전달하는 것이 좋습니다. FluentBit가 보관을 위해 컨테이너 로그를 전송하는 대상인 [OUTPUT] 섹션에서 이를 정의합니다. 여기에서 추가 구성 파라미터를 검토합니다.

```
[OUTPUT]
        Name eks_to_cloudwatch_logs
        Match application.*
        region us-east-2
        log_group_name fluent-bit-cloudwatch
        log_stream_prefix from-fluent-bit-
        auto_create_group On
```

그러나 이는 비용 절감에 즉각적인 영향을 미치지 않습니다. 추가 비용 절감을 위해 이러한 로그를 Amazon S3로 내보내야 합니다.

CloudWatch에서 Amazon S3로 내보내기

Amazon CloudWatch logs를 장기간 저장하려면 Amazon EKS CloudWatch 로그를 Amazon Simple Storage Service(Amazon S3)로 내보내는 것이 좋습니다. 콘솔 또는 API를 통해 내보내기 작업을 생성하여 로그를 Amazon S3 버킷에 전달할 수 있습니다. 이렇게 하면 Amazon S3는 비용을 추가로 절감할 수 있는 다양한 옵션을 제공합니다. 자체 Amazon S3 수명 주기 규칙을 정의하여가 필요에 맞는 스토리지 클래스로 로그를 이동하거나 Amazon S3 Intelligent-Tiering 스토리지 클래스를 활용하여 AWS가 사용 패턴에 따라 데이터를 장기 스토리지로 자동으로 이동하도록 할 수 있습니다. 자세한 내용은이

블로그를 참조하세요. 예를 들어 프로덕션 환경의 경우 로그는 30일 이상 CloudWatch에 상주한 다음 Amazon S3 버킷으로 내보냅니다. 그런 다음 나중에 로그를 다시 참조해야 하는 경우 Amazon Athena를 사용하여 Amazon S3 버킷의 데이터를 쿼리할 수 있습니다.

로그 수준 감소

애플리케이션에 대한 선택적 로깅을 연습합니다. 애플리케이션과 노드 모두 기본적으로 로그를 출력합니다. 애플리케이션 로그의 경우 워크로드 및 환경의 중요도에 맞게 로그 수준을 조정합니다. 예를 들어 아래 Java 애플리케이션은 일반적인 기본 애플리케이션 구성인 INFO 로그를 출력하고 있으며, 코드에 따라 많은 양의 로그 데이터가 생성될 수 있습니다.

```
import org.apache.log4j.*;

public class LogClass {
   private static org.apache.log4j.Logger log = Logger.getLogger(LogClass.class);

public static void main(String[] args) {
      log.setLevel(Level.INFO);

   log.debug("This is a DEBUG message, check this out!");
   log.info("This is an INFO message, nothing to see here!");
   log.warn("This is a WARN message, investigate this!");
   log.error("This is an ERROR message, check this out!");
   log.fatal("This is a FATAL message, investigate this!");    } }
```

개발 환경에서는 로그 수준을 로 변경합니다. DEBUG이렇게 하면 문제를 디버깅하거나 프로덕션에 들어가기 전에 잠재적 문제를 포착하는 데 도움이 될 수 있습니다.

```
      log.setLevel(Level.DEBUG);
```

프로덕션 환경에서는 로그 수준을 ERROR 또는 로 수정하는 것이 좋습니다FATAL. 이렇게 하면 애플리케이션에 오류가 있는 경우에만 로그가 출력되어 로그 출력이 줄어들고 애플리케이션 상태에 대한 중요한 데이터에 집중할 수 있습니다.

```
      log.setLevel(Level.ERROR);
```

다양한 Kubernetes 구성 요소 로그 수준을 미세 조정할 수 있습니다. 예를 들어 Bottlerocket을 EKS 노드 운영 체제로 사용하는 경우 kubelet 프로세스 로그 수준을 조정할 수 있는 구성 설정이 있습니다. 이 구성 설정의 코드 조각은 다음과 같습니다. kubelet 프로세스의 로깅 세부 정보를 조정하는 기본 로그 수준 2를 기록해 둡니다.

```
[settings.kubernetes]
log-level = "2"
image-gc-high-threshold-percent = "85"
image-gc-low-threshold-percent = "80"
```

개발 환경의 경우 로그 수준을 2보다 크게 설정하여 추가 이벤트를 볼 수 있습니다. 이는 디버깅에 적합합니다. 프로덕션 환경의 경우 중요한 이벤트만 보기 위해 레벨을 0으로 설정할 수 있습니다.

필터 활용

기본 EKS Fluentbit 구성을 사용하여 컨테이너 로그를 Cloudwatch로 전송할 때 FluentBit는 아래 [INPUT] 구성 블록과 같이 Kubernetes 메타데이터가 풍부한 모든 애플리케이션 컨테이너 로그를 캡처하여 Cloudwatch로 전송합니다.

```
 [INPUT]
     Name                tail
     Tag                 application.*
     Exclude_Path        /var/log/containers/cloudwatch-agent*, /var/log/containers/
fluent-bit*, /var/log/containers/aws-node*, /var/log/containers/kube-proxy*
     Path                /var/log/containers/*.log
     Docker_Mode         On
     Docker_Mode_Flush   5
     Docker_Mode_Parser  container_firstline
     Parser              docker
     DB                  /var/fluent-bit/state/flb_container.db
     Mem_Buf_Limit       50MB
     Skip_Long_Lines     On
     Refresh_Interval    10
     Rotate_Wait         30
     storage.type        filesystem
     Read_from_Head      ${READ_FROM_HEAD}
```

위 [INPUT] 섹션에서는 모든 컨테이너 로그를 수집합니다. 이렇게 하면 필요 없는 대량의 데이터가 생성될 수 있습니다. 이 데이터를 필터링하면 CloudWatch로 전송되는 로그 데이터의 양이 줄어들어 비용이 절감됩니다. CloudWatch로 출력하기 전에 로그에 필터를 적용할 수 있습니다. Fluentbit은 [FILTER] 섹션에서 이를 정의합니다. 예를 들어 Kubernetes 메타데이터가 로그 이벤트에 추가되지 않도록 필터링하면 로그 볼륨이 감소할 수 있습니다.

```
    [FILTER]
        Name                nest
        Match               application.*
```

```
        Operation           lift
        Nested_under        kubernetes
        Add_prefix          Kube.

    [FILTER]
        Name                modify
        Match               application.*
        Remove              Kube.<Metadata_1>
        Remove              Kube.<Metadata_2>
        Remove              Kube.<Metadata_3>

    [FILTER]
        Name                nest
        Match               application.*
        Operation           nest
        Wildcard            Kube.*
        Nested_under        kubernetes
        Remove_prefix       Kube.
```

Metrics

지표는 시스템 성능에 대한 중요한 정보를 제공합니다. 중앙 위치에서 모든 시스템 관련 또는 사용 가능한 리소스 지표를 통합하면 성능 데이터를 비교하고 분석할 수 있습니다. 이 중앙 집중식 접근 방식을 사용하면 리소스 규모 조정 또는 규모 축소와 같은 정보에 입각한 전략적 결정을 내릴 수 있습니다. 또한 지표는 리소스의 상태를 평가하는 데 중요한 역할을 하므로 필요한 경우 사전 예방 조치를 취할 수 있습니다. 일반적으로 관찰성 비용은 원격 측정 데이터 수집 및 보존에 따라 조정됩니다. 다음은 지표 원격 측정 비용을 줄이기 위해 구현할 수 있는 몇 가지 전략입니다. 중요한 지표만 수집하고, 원격 측정 데이터의 카디널리티를 줄이고, 원격 측정 데이터 수집의 세밀도를 미세 조정하는 것입니다.

무엇이 중요한지 모니터링하고 필요한 것만 수집

첫 번째 비용 절감 전략은 수집하는 지표 수를 줄이고 결과적으로 보존 비용을 줄이는 것입니다.

1. 먼저 사용자 및/또는 이해관계자의 요구 사항에서 벗어나 가장 중요한 지표를 결정합니다. 성공 지표는 모든 사람에게 다릅니다! 좋은 모습이 무엇인지 파악하고 측정합니다.
2. 지원하는 워크로드에 대해 자세히 살펴보고 KPIs(Key Performance Indicator)를 식별해 보세요. 'Golden Signals'라고도 합니다. 이는 비즈니스 및 이해관계자 요구 사항에 부합해야 합니다. Amazon CloudWatch 및 지표 수학SLIs, SLOs 및 SLAs를 계산하는 것은 서비스 신뢰성을 관리하는 데 매우 중요합니다. EKS 환경의 성능을 효과적으로 모니터링하고 유지하려면이 가이드에 설명된 모범 사례를 따르세요.

3. 그런 다음 다양한 인프라 계층을 계속 진행하여 EKS 클러스터, 노드 및 추가 인프라 지표를 워크로드 KPIs에 연결하고 상호 연관시킵니다. 비즈니스 지표와 운영 지표를 함께 상호 연관시킬 수 있는 시스템에 저장하고 두 지표 모두에 대해 관찰된 영향을 기반으로 결론을 도출합니다.
4. EKS는 컨트롤 플레인, 클러스터 kube-state-metrics를 노출합니다. 이러한 모든 지표의 관련성은 요구 사항에 따라 다르지만 여러 계층에 걸쳐 모든 단일 지표가 필요한 것은 아닙니다. 이 EKS 필수 지표 가이드를 EKS 클러스터 및 워크로드의 전반적인 상태를 모니터링하기 위한 기준으로 사용할 수 있습니다.

다음은를 사용하여 kubelet 지표만 유지하고 모든 컨테이너 지표`relabel_config`를 `metric_relabel_config` 삭제하는 prometheus 스크레이프 구성의 예입니다.

```
  kubernetes_sd_configs:
  - role: endpoints
    namespaces:
      names:
      - kube-system
  bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  tls_config:
    insecure_skip_verify: true
  relabel_configs:
  - source_labels: [__meta_kubernetes_service_label_k8s_app]
    regex: kubelet
    action: keep

  metric_relabel_configs:
  - source_labels: [__name__]
    regex: container_(network_tcp_usage_total|network_udp_usage_total|tasks_state|
cpu_load_average_10s)
    action: drop
```

해당하는 경우 카디널리티 감소

카디널리티는 특정 지표 세트에 대한 차원(예: prometheus 레이블)과 함께 데이터 값의 고유성을 나타냅니다. 카디널리티 지표가 높을수록 차원이 많고 각 차원 지표 조합의 고유성이 높아집니다. 카디널리티가 높을수록 지표 원격 측정 데이터 크기와 스토리지 요구 사항이 커져 비용이 증가합니다.

아래 높은 카디널리티 예제에서 지표, 지연 시간은 차원, RequestID, CustomerID 및 서비스를 포함하고 각 차원에는 많은 고유 값이 있음을 확인할 수 있습니다. 카디널리티는 차원당 가능한 값 수의 조합에 대한 측정치입니다. Prometheus에서 각 고유 차원/레이블 세트는 새 지표로 간주되므로 카디널리티가 높을수록 지표가 더 많아집니다.

지표당 지표와 차원/레이블이 많은 EKS 환경(클러스터, 네임스페이스, 서비스, 포드, 컨테이너 등)에서는 카디널리티가 증가하는 경향이 있습니다. 비용을 최적화하려면 수집하는 지표의 카디널리티를 신중하게 고려하세요. 예를 들어 클러스터 수준에서 시각화를 위해 특정 지표를 집계하는 경우 네임스페이스 레이블과 같이 하위 계층에 있는 추가 레이블을 삭제할 수 있습니다.

prometheus에서 높은 카디널리티 지표를 식별하기 위해 다음 PROMQL 쿼리를 실행하여 지표 수가 가장 많은 스크레이프 대상(카디널리티)을 확인할 수 있습니다.

```
topk_max(5, max_over_time(scrape_samples_scraped[1h]))
```

및 다음 PROMQL 쿼리는 지표 이탈률이 가장 높은 스크레이프 대상(특정 스크레이프에서 생성된 새 지표 시리즈 수)을 결정하는 데 도움이 될 수 있습니다.

```
topk_max(5, max_over_time(scrape_series_added[1h]))
```

grafana를 사용하는 경우 Grafana Lab의 Mimirtool을 사용하여 grafana 대시보드 및 prometheus 규칙을 분석하여 사용하지 않는 높은 카디널리티 지표를 식별할 수 있습니다. `mimirtool analyze` 및 `mimirtool analyze prometheus` 명령을 사용하여 대시보드에서 참조되지 않는 활성 지표를 식별하는 방법에 대한 이 가이드를 따르세요.

지표 세분화 고려

매초 대 매분과 같이 더 세밀하게 지표를 수집하면 원격 측정 수집 및 저장량에 큰 영향을 미치므로 비용이 늘어날 수 있습니다. 일시적인 문제를 볼 수 있을 만큼 충분히 세분화되고 비용 효율적일 만큼 충분히 낮은 수준 사이에서 균형을 이루는 합리적인 스크레이프 또는 지표 수집 간격을 결정합니다. 용량 계획 및 더 큰 기간 분석에 사용되는 지표의 세분화를 줄입니다.

다음은 Opentelemetry용 AWS Distro(ADOT) EKS Addon Collector 구성의 기본 코드 조각입니다.

> **Important**
>
> 글로벌 prometheus 스크레이프 간격은 15초로 설정됩니다. 이 스크레이프 간격을 늘리면 prometheus에서 수집된 지표 데이터의 양이 감소할 수 있습니다.

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: my-collector-amp
```

```
...

config: |
    extensions:
      sigv4auth:
        region: "+++<YOUR_AWS_REGION>+++" service: "aps"+++</YOUR_AWS_REGION>+++

 receivers:
   #
   # Scrape configuration for the Prometheus Receiver
   # This is the same configuration used when Prometheus is installed using the
 community Helm chart
   #
   prometheus:
     config:
       global:   scrape_interval: 15s
         scrape_timeout: 10s
```

추적

추적과 관련된 기본 비용은 추적 스토리지 생성에서 발생합니다. 추적의 목표는 성능 측면을 진단하고 이해하기에 충분한 데이터를 수집하는 것입니다. 그러나 X-Ray 트레이스 비용은 X-Ray로 전달된 데이터를 기반으로 하므로 트레이스가 전달된 후 트레이스를 삭제해도 비용이 절감되지 않습니다. 적절한 분석을 수행할 수 있도록 데이터를 유지하면서 추적 비용을 낮추는 방법을 검토해 보겠습니다.

샘플링 규칙 적용

X-Ray 샘플링 속도는 기본적으로 보수적입니다. 수집하는 데이터의 양을 제어할 수 있는 샘플링 규칙을 정의합니다. 이렇게 하면 비용을 절감하면서 성능 효율성을 개선할 수 있습니다. 샘플링 속도를 줄이면 워크로드에 필요한 것만 요청에서 트레이스를 수집하는 동시에 비용 구조를 낮출 수 있습니다.

예를 들어 1개의 문제가 있는 경로에 대한 모든 요청의 트레이스를 디버깅하려는 Java 애플리케이션이 있습니다.

SDK를 통해 JSON 문서에서 샘플링 규칙을 로드하도록 구성

```
{
"version": 2,
  "rules": [
    {
"description": "debug-eks",
```

```
      "host": "*",
      "http_method": "PUT",
      "url_path": "/history/*",
      "fixed_target": 0,
      "rate": 1,
      "service_type": "debug-eks"
    }
  ],
  "default": {
"fixed_target": 1,
    "rate": 0.1
  }
}
```

콘솔을 통해

OpenTelemetry용 AWS Distro(ADOT)를 사용하여 테일 샘플링 적용

ADOT 테일 샘플링을 사용하면 서비스에 수집된 트레이스의 볼륨을 제어할 수 있습니다. 그러나 테일 샘플링을 사용하면 처음에 대신 요청의 모든 범위가 완료된 후 샘플링 정책을 정의할 수 있습니다. 이렇게 하면 CloudWatch로 전송되는 원시 데이터의 양이 추가로 제한되므로 비용이 절감됩니다.

예를 들어 랜딩 페이지로 가는 트래픽의 1%와 결제 페이지로 가는 요청의 10%를 샘플링하는 경우 30분 동안 300개의 트레이스가 남을 수 있습니다. 특정 오류를 필터링하는의 ADOT 테일 샘플링 규칙을 사용하면 200개의 트레이스가 남아 저장되는 트레이스 수가 줄어들 수 있습니다.

```
processors:
  groupbytrace:
    wait_duration: 10s
    num_traces: 300
    tail_sampling:
    decision_wait: 1s # This value should be smaller than wait_duration
    policies:
      - ..... # Applicable policies**
  batch/tracesampling:
    timeout: 0s # No need to wait more since this will happen in previous processors
    send_batch_max_size: 8196 # This will still allow us to limit the size of the
 batches sent to subsequent exporters

service:
  pipelines:
    traces/tailsampling:
      receivers: [otlp]
```

```
    processors: [groupbytrace, tail_sampling, batch/tracesampling]
    exporters: [awsxray]
```

Amazon S3 스토리지 옵션 활용

AWS S3 버킷과 다양한 스토리지 클래스를 활용하여 트레이스를 저장해야 합니다. 보존 기간이 만료되기 전에 S3로 트레이스를 내보냅니다. Amazon S3 수명 주기 규칙을 사용하여 추적 데이터를 요구 사항을 충족하는 스토리지 클래스로 이동합니다.

예를 들어 90일 전의 트레이스가 있는 경우 Amazon S3 Intelligent-Tiering은 사용 패턴에 따라 데이터를 장기 스토리지로 자동으로 이동할 수 있습니다. 나중에 추적을 다시 참조해야 하는 경우 Amazon Athena를 사용하여 Amazon S3의 데이터를 쿼리할 수 있습니다. 이렇게 하면 분산 추적 비용을 더욱 줄일 수 있습니다.

추가 리소스:

- 관찰성 모범 사례 가이드
- 모범 사례 지표 수집
- AWS re:Invent 2022 - Amazon의 관찰성 모범 사례(COP343)
- AWS re:Invent 2022 - Observability: 최신 애플리케이션을 위한 모범 사례(COP344)

Windows 모범 사례

이 가이드에서는 Windows 컨테이너 및 노드 실행에 대한 조언을 제공합니다.

주제

- Amazon EKS 최적화 Windows AMI 관리
- Windows 포드 및 컨테이너용 gMSA 구성
- Windows 작업자 노드 강화
- 컨테이너 이미지 스캔
- Windows Server 버전 및 라이선스
- 로깅
- 모니터링
- Windows 네트워킹
- OOM 오류 방지
- Windows Server 및 컨테이너 패치 적용
- 이기종 워크로드 실행
- 포드 보안 컨텍스트
- 영구 스토리지 옵션
- Windows 컨테이너 이미지 강화

Amazon EKS 최적화 Windows AMI 관리

Windows Amazon EKS 최적화 AMI는 Windows Server 2019 및 Windows Server 2022를 기반으로 빌드되었습니다. Amazon EKS 노드의 기본 이미지 역할을 하도록 구성되었습니다. 기본적으로 AMI에는 다음 구성 요소가 포함됩니다.

- kubelet
- kube-proxy
- Kubernetes용 AWS IAM Authenticator
- csi-proxy
- containerd

AWS Systems Manager Parameter Store API를 쿼리하여 Amazon EKS 최적화 AMIs) ID를 프로그래밍 방식으로 검색할 수 있습니다. 이 파라미터를 사용하면 Amazon EKS 최적화 AMI ID를 수동으로 조회할 필요가 없습니다. Systems Manager 파라미터 스토어 API에 대한 자세한 내용은 GetParameter 섹션을 참조하세요. Amazon EKS 최적화 AMI 메타데이터를 검색하려면 사용자 계정에 ssm:GetParameter IAM 권한이 있어야 합니다.

다음 예시에서는 Windows Server 2019 LTSC Core용 최신 Amazon EKS 최적화 AMI의 AMI ID를 검색합니다. AMI 이름에 나열된 버전 번호는 준비된 해당 Kubernetes 빌드와 관련이 있습니다.

```
aws ssm get-parameter --name /aws/service/ami-windows-latest/Windows_Server-2019-
English-Core-EKS_Optimized-1.21/image_id --region us-east-1 --query "Parameter.Value"
 --output text
```

출력 예시:

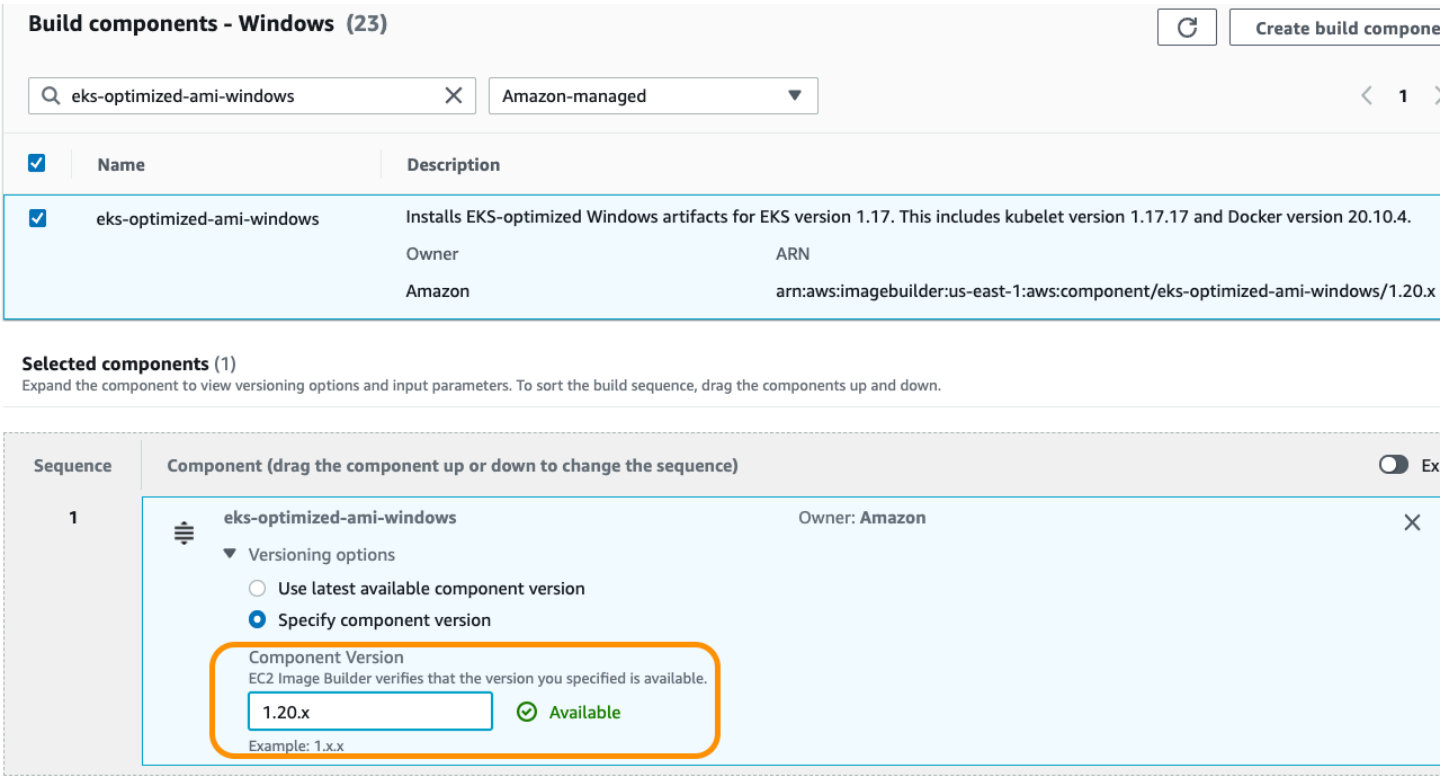
```
ami-09770b3eec4552d4e
```

자체 Amazon EKS 최적화 Windows AMI 관리

프로덕션 환경을 위한 필수 단계는 Amazon EKS 클러스터에서 동일한 Amazon EKS 최적화 Windows AMI 및 kubelet 버전을 유지하는 것입니다.

Amazon EKS 클러스터에서 동일한 버전을 사용하면 문제 해결 시간이 단축되고 클러스터 일관성이 향상됩니다. Amazon EC2 Image Builder는 Amazon EKS 클러스터에서 사용할 사용자 지정 Amazon EKS 최적화 Windows AMIs를 생성하고 유지 관리하는 데 도움이 됩니다.

Amazon EC2 Image Builder를 사용하여 Windows Server 버전, AWS Windows Server AMI 릴리스 날짜 및/또는 OS 빌드 버전 중에서 선택합니다. 빌드 구성 요소 단계에서는 기존 EKS 최적화 Windows 아티팩트와 kubelet 버전 중에서 선택할 수 있습니다. 자세한 내용은 https://docs.aws.amazon.com/eks/latest/userguide/eks-custom-ami-windows.html 참조하십시오.



참고: 기본 이미지를 선택하기 전에 릴리스 채널 업데이트와 관련된 중요한 세부 정보는 Windows Server 버전 및 라이선스 섹션을 참조하세요.

사용자 지정 EKS 최적화 AMIs 대한 더 빠른 시작 구성

사용자 지정 Windows Amazon EKS 최적화 AMI를 사용하는 경우 빠른 시작 기능을 활성화하여 Windows 작업자 노드를 최대 65% 더 빠르게 시작할 수 있습니다. 이 기능은 Sysprep 전문화, OOBE(Windows Out of Box Experience) 단계 및 필요한 재부팅이 이미 완료된 사전 프로비저닝된 스냅샷 세트를 유지합니다. 그런 다음 이러한 스냅샷은 후속 시작 시 사용되므로 노드를 스케일 아웃하거나 교체하는 데 걸리는 시간이 줄어듭니다. 빠른 시작은 EC2 콘솔 또는 AWS CLI를 통해 소유AMIs에 대해서만 활성화할 수 있으며 유지 관리되는 스냅샷 수는 구성할 수 있습니다.

참고: Fast Launch는 Amazon에서 제공하는 기본 EKS 최적화 AMI와 호환되지 않으므로 활성화하기 전에 위와 같이 사용자 지정 AMI를 생성합니다.

자세한 내용: AWS Windows AMIs- 더 빠른 시작을 위해 AMI 구성

사용자 지정 AMIs에서 Windows 기본 계층 캐싱

Windows 컨테이너 이미지는 Linux 컨테이너 이미지보다 큽니다. 컨테이너화된 .NET Framework 기반 애플리케이션을 실행하는 경우 평균 이미지 크기는 약 8.24GB입니다. 포드 예약 중에 포드가 실행 중 상태에 도달하기 전에 컨테이너 이미지를 완전히 가져와 디스크에서 추출해야 합니다.

이 프로세스 중에 컨테이너 런타임(컨테이너드)은 디스크에서 전체 컨테이너 이미지를 가져와 추출합니다. 풀 작업은 병렬 프로세스입니다. 즉, 컨테이너 런타임은 컨테이너 이미지 계층을 병렬로 가져옵니다. 반대로 추출 작업은 순차적 프로세스에서 수행되며 I/O 집약적입니다. 따라서 컨테이너 이미지를 완전히 추출하고 컨테이너 런타임(컨테이너드)에서 사용할 준비가 되는 데 8분 이상 걸릴 수 있으므로 포드 시작 시간은 몇 분 정도 걸릴 수 있습니다.

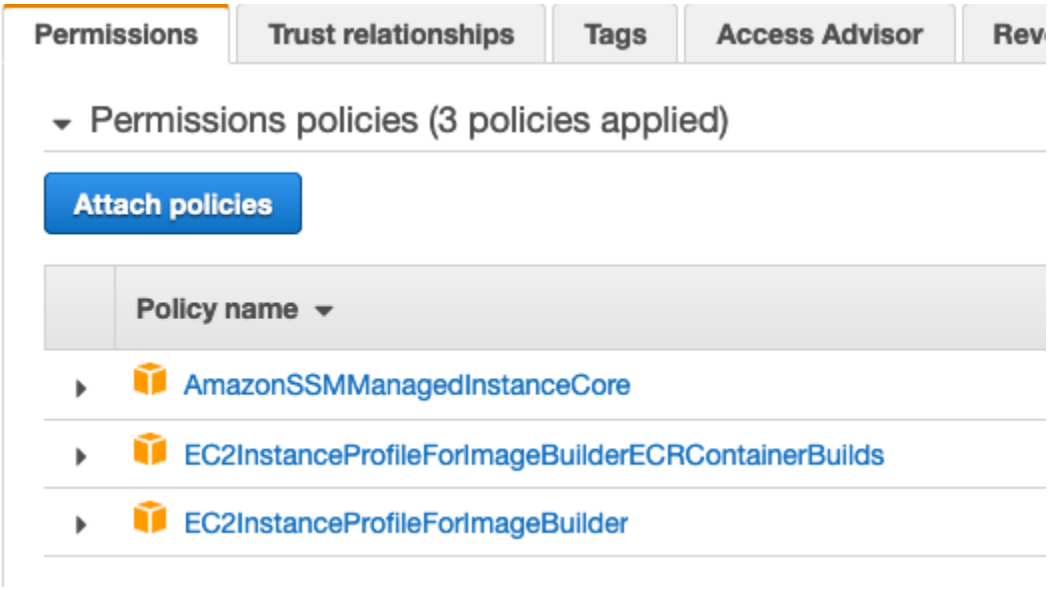
Windows Server 및 컨테이너 패치 적용 주제에서 언급했듯이 EKS를 사용하여 사용자 지정 AMI를 빌드하는 옵션이 있습니다. AMI 준비 중에 EC2 Image Builder 구성 요소를 추가하여 필요한 모든 Windows 컨테이너 이미지를 로컬에서 가져온 다음 AMI를 생성할 수 있습니다. 이 전략은 포드가 실행 중 상태에 도달하는 시간을 크게 줄입니다.

Amazon EC2 Image Builder에서 구성 요소를 생성하여 필요한 이미지를 다운로드하고 이미지 레시피에 연결합니다. 다음 예시에서는 ECR 리포지토리에서 특정 이미지를 가져옵니다.

```
name: ContainerdPull
description: This component pulls the necessary containers images for a cache strategy.
schemaVersion: 1.0

phases:
  - name: build
    steps:
      - name: containerdpull
        action: ExecutePowerShell
        inputs:
          commands:
            - Set-ExecutionPolicy Unrestricted -Force
            - (Get-ECRLoginCommand).Password | docker login --username AWS --password-
stdin 111000111000.dkr.ecr.us-east-1.amazonaws.com
            - ctr image pull mcr.microsoft.com/dotnet/framework/aspnet:latest
            - ctr image pull 111000111000.dkr.ecr.us-east-1.amazonaws.com/
myappcontainerimage:latest
```

다음 구성 요소가 예상대로 작동하는지 확인하려면 EC2 Image 빌더(EC2InstanceProfileForImageBuilder)에서 사용하는 IAM 역할에 연결된 정책이 있는지 확인합니다.



블로그 게시물

다음 블로그 게시물에서는 사용자 지정 Amazon EKS Windows AMIs에 캐싱 전략을 구현하는 방법에 대한 단계별 정보를 제공합니다.

EC2 이미지 빌더 및 이미지 캐시 전략을 사용하여 Windows 컨테이너 시작 시간 단축

Windows 포드 및 컨테이너용 gMSA 구성

gMSA 계정이란?

.NET 애플리케이션과 같은 Windows 기반 애플리케이션은 종종 Active Directory를 자격 증명 공급자로 사용하여 NTLM 또는 Kerberos 프로토콜을 사용하여 권한 부여/인증을 제공합니다.

Kerberos 티켓을 Active Directory와 교환하는 애플리케이션 서버는 도메인에 조인되어야 합니다. Windows 컨테이너는 도메인 조인을 지원하지 않으며 컨테이너가 임시 리소스이기 때문에 Active Directory RID 풀에 부담을 주지 않습니다.

그러나 관리자는 gMSA Active Directory 계정을 활용하여 Windows 컨테이너, NLB 및 서버 팜과 같은 리소스에 대한 Windows 인증을 협상할 수 있습니다.

Windows 컨테이너 및 gMSA 사용 사례

Windows 인증을 활용하고 Windows 컨테이너로 실행되는 애플리케이션은 Windows 노드가 컨테이너를 대신하여 Kerberos 티켓을 교환하는 데 사용되므로 gMSA의 이점을 누릴 수 있습니다. gMSA 통합을 지원하도록 Windows 작업자 노드를 설정하는 데 사용할 수 있는 두 가지 옵션이 있습니다.

1 - 도메인에 조인된 Windows 작업자 노드

이 설정에서 Windows 작업자 노드는 Active Directory 도메인에 도메인 조인되고, Windows 작업자 노드의 AD 컴퓨터 계정은 Active Directory에 대해 인증하고 포드에 사용할 gMSA 자격 증명을 검색하는 데 사용됩니다.

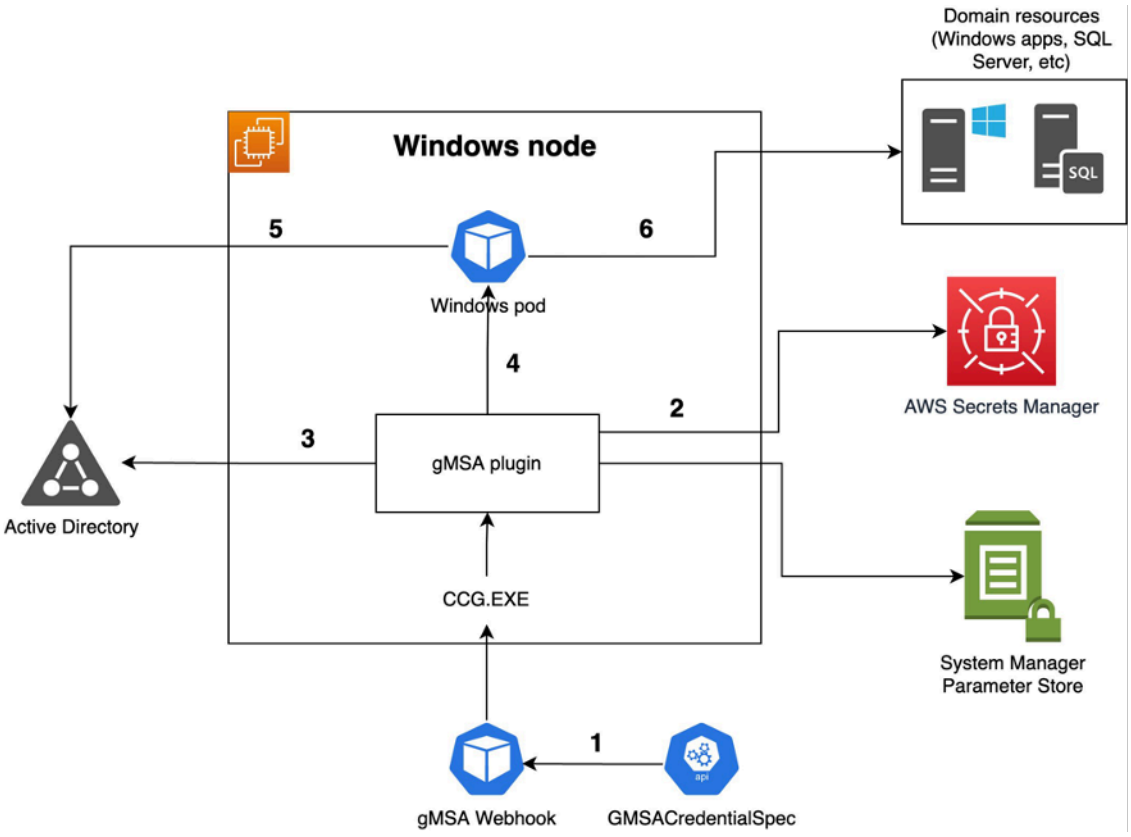
도메인 조인 접근 방식에서는 기존 Active Directory GPOs를 사용하여 Windows 작업자 노드를 쉽게 관리하고 강화할 수 있지만, Kubernetes 클러스터에서 Windows 작업자 노드에 조인하는 동안 추가 운영 오버헤드와 지연이 발생합니다. Kubernetes 클러스터가 노드를 종료한 후 노드 시작 및 Active Directory 차고 정리 중에 추가 재부팅이 필요하기 때문입니다.

다음 블로그 게시물에서는 도메인에 조인된 Windows 작업자 노드 접근 방식을 구현하는 방법에 대한 step-by-step 내용을 확인할 수 있습니다.

[Amazon EKS Windows 포드의 Windows 인증]

2 - 도메인 없는 Windows 작업자 노드

이 설정에서 Windows 작업자 노드는 Active Directory 도메인에 조인되지 않으며, "휴대용" 자격 증명(사용자/암호)은 Active Directory에 대해 인증하고 포드에 사용할 gMSA 자격 증명을 검색하는 데 사용됩니다.



휴대용 자격 증명은 Active Directory 사용자입니다. 자격 증명(사용자/암호)은 AWS Secrets Manager 또는 AWS System Manager Parameter Store에 저장되며, ccg_plugin이라는 AWS에서 개발한 플러그인을 사용하여 AWS Secrets Manager 또는 AWS System Manager Parameter Store에서이 자격 증명을 검색하고 컨테이너로 전달하여 gMSA 자격 증명을 검색하고 포드에 사용할 수 있도록 합니다.

이 도메인 없는 접근 방식에서는 gMSA를 사용하고 Active Directory 관리자의 운영 오버헤드를 줄일 때 Windows 작업자 노드 시작 중에 Active Directory 상호 작용이 없는 이점을 누릴 수 있습니다.

다음 블로그 게시물에서는 Domainless Windows 작업자 노드 접근 방식을 구현하는 방법에 대한 step-by-step 내용을 확인할 수 있습니다.

Amazon EKS Windows 포드에 대한 도메인 없는 Windows 인증

중요 정보

포드가 gMSA 계정을 사용할 수 있더라도 Windows 인증을 지원하도록 애플리케이션 또는 서비스를 적절히 설정해야 합니다. 예를 들어 Windows 인증을 지원하도록 Microsoft IIS를 설정하려면 dockerfile을 통해 준비해야 합니다.

```
RUN Install-WindowsFeature -Name Web-Windows-Auth -IncludeAllSubFeature
RUN Import-Module WebAdministration; Set-ItemProperty 'IIS:\AppPools\SiteName' -name
 processModel.identityType -value 2
RUN Import-Module WebAdministration; Set-WebConfigurationProperty -Filter '/
system.webServer/security/authentication/anonymousAuthentication' -Name Enabled -Value
 False -PSPath 'IIS:\' -Location 'SiteName'
RUN Import-Module WebAdministration; Set-WebConfigurationProperty -Filter '/
system.webServer/security/authentication/windowsAuthentication' -Name Enabled -Value
 True -PSPath 'IIS:\' -Location 'SiteName'
```

Windows 작업자 노드 강화

OS 강화는 시스템을 잠그고 공격 표면을 줄이는 것을 목표로 하는 불필요한 소프트웨어 패키지의 OS 구성, 패치 적용 및 제거의 조합입니다. 회사에서 요구하는 강화 구성을 사용하여 자체 EKS 최적화 Windows AMI를 준비하는 것이 가장 좋습니다.

AWS는 최신 Windows Server 보안 패치가 포함된 새로운 EKS 최적화 Windows AMI를 매월 제공합니다. 그러나 자체 관리형 또는 관리형 노드 그룹을 사용하는지 여부에 관계없이 필요한 OS 구성을 적용하여 AMI를 강화하는 것은 여전히 사용자의 책임입니다.

Microsoft는 보안 정책 요구 사항에 따라 강화하는 데 도움이 되는 Microsoft 보안 규정 준수 도구 키트 및 보안 기준과 같은 다양한 도구를 제공합니다. CIS 벤치마크도 사용할 수 있으며 프로덕션 환경을 위한 Amazon EKS 최적화 Windows AMI를 기반으로 구현해야 합니다.

Windows Server Core를 사용하여 공격 표면 줄이기

Windows Server Core는 EKS 최적화 Windows AMI의 일부로 사용할 수 있는 최소 설치 옵션입니다. Windows Server Core 배포에는 몇 가지 이점이 있습니다. 먼저 디스크 공간이 비교적 작으며, Server Core의 경우 6GB이고 데스크톱 환경이 있는 Windows Server의 경우 10GB입니다. 둘째, 더 작은 코드 베이스와 사용 가능한 APIs.

AWS는 Amazon EKS 지원 버전과 관계없이 최신 Microsoft 보안 패치가 포함된 새로운 Amazon EKS 최적화 Windows AMIs를 고객에게 매월 제공합니다. 모범 사례로 Windows 작업자 노드는 최신 Amazon EKS 최적화 AMI를 기반으로 새 노드로 교체해야 합니다. 업데이트 또는 노드 교체 없이 45일 이상 실행 중인 모든 노드에는 보안 모범 사례가 없습니다.

RDP 연결 방지

RDP(원격 데스크톱 프로토콜)는 네트워크를 통해 다른 Windows 컴퓨터에 연결할 수 있는 그래픽 인터페이스를 사용자에게 제공하기 위해 Microsoft에서 개발한 연결 프로토콜입니다.

가장 좋은 방법은 Windows 작업자 노드를 임시 호스트인 것처럼 취급하는 것입니다. 즉, 관리 연결, 업데이트 및 문제 해결이 없습니다. 모든 수정 및 업데이트는 새 사용자 지정 AMI로 구현하고 Auto Scaling 그룹을 업데이트하여 대체해야 합니다. Windows Server 및 컨테이너 패치 및 Amazon EKS 최적화 Windows AMI 관리를 참조하세요.

아래 예제와 같이 ssh 속성에 false 값을 전달하여 배포 중에 Windows 노드에서 RDP 연결을 비활성화합니다.

```
nodeGroups:
- name: windows-ng
  instanceType: c5.xlarge
  minSize: 1
  volumeSize: 50
  amiFamily: WindowsServer2019CoreContainer
  ssh:
    allow: false
```

Windows 노드에 액세스해야 하는 경우 AWS System Manager 세션 관리자를 사용하여 AWS 콘솔 및 SSM 에이전트를 통해 안전한 PowerShell 세션을 설정합니다. 솔루션을 구현하는 방법을 확인하려면 AWS Systems Manager 세션 관리자를 사용하여 Windows 인스턴스에 안전하게 액세스하기

System Manager Session Manager를 사용하려면 Windows 작업자 노드를 시작하는 데 사용되는 IAM 역할에 추가 IAM 정책을 적용해야 합니다. 다음은 AmazonSSMManagedInstanceCore가 `eksctl` 클러스터 매니페스트에 지정되는 예입니다.

```
 nodeGroups:
- name: windows-ng
  instanceType: c5.xlarge
  minSize: 1
  volumeSize: 50
  amiFamily: WindowsServer2019CoreContainer
  ssh:
    allow: false
  iam:
    attachPolicyARNs:
      - arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy
      - arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy
      - arn:aws:iam::aws:policy/ElasticLoadBalancingFullAccess
      - arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly
      - arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore
```

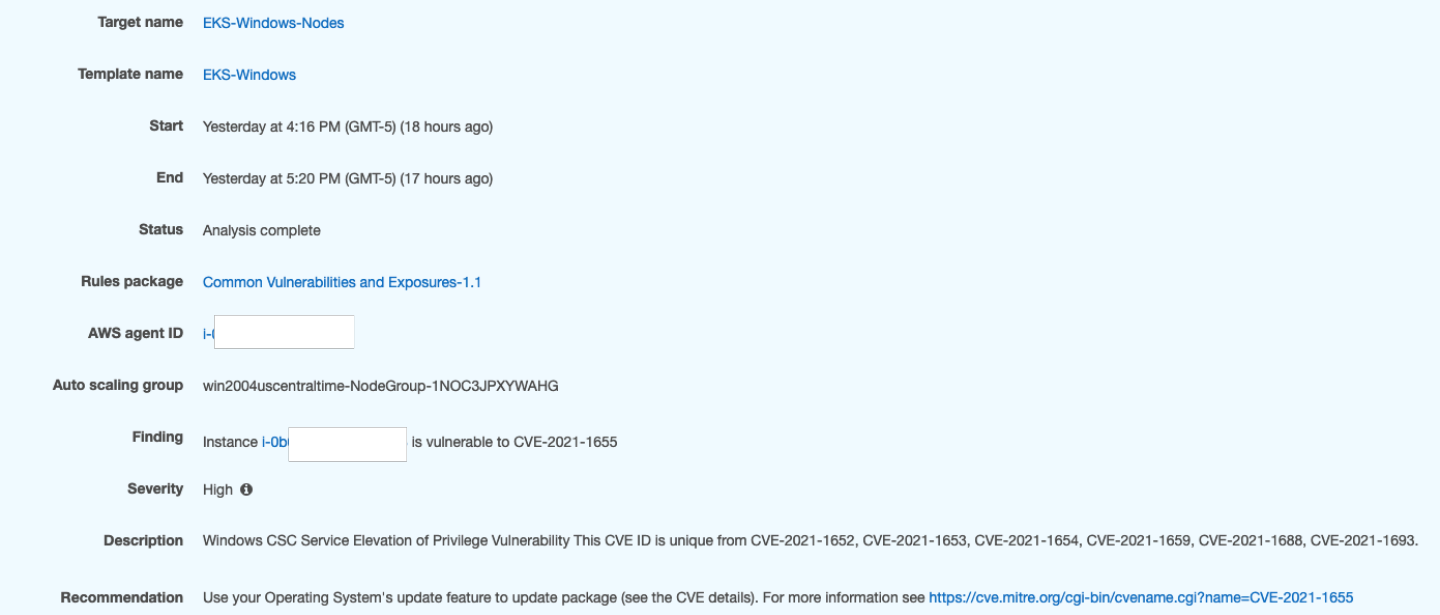
Amazon Inspector –

> Amazon Inspector는 AWS에 배포된 애플리케이션의 보안 및 규정 준수를 개선하는 데 도움이 되는 자동화된 보안 평가 서비스입니다. Amazon Inspector는 애플리케이션의 노출, 취약성 및 모범 사례와의 편차를 자동으로 평가합니다. 평가를 수행한 후, Amazon Inspector는 상세한 보안 평가 결과 목록을 제공하며, 이 목록은 심각도 수준에 따라 구성되어 있습니다. 이러한 결과는 직접 검토하거나 Amazon Inspector 콘솔 또는 API를 통해 사용할 수 있는 세부 평가 보고서의 일부로 검토할 수 있습니다.

Amazon Inspector는 Windows 작업자 노드에서 CIS 벤치마크 평가를 실행하는 데 사용할 수 있으며 다음 작업을 수행하여 Windows Server Core에 설치할 수 있습니다.

1. .exe 파일을 다운로드합니다. https://inspector-agent.amazonaws.com/windows/installer/latest/AWSAgentInstall.exe
2. 에이전트를 Windows 작업자 노드로 전송합니다.
3. PowerShell에서 다음 명령을 실행하여 Amazon Inspector 에이전트를 설치합니다. `.\AWSAgentInstall.exe /install`

다음은 첫 번째 실행 후 출력입니다. 보시다시피 CVE 데이터베이스를 기반으로 조사 결과를 생성했습니다. 이를 사용하여 작업자 노드를 강화하거나 강화된 구성을 기반으로 AMI를 생성할 수 있습니다.



Amazon Inspector 에이전트를 설치하고, CIS 벤치마크 평가를 설정하고, 보고서를 생성하는 방법을 포함하여 Amazon Inspector에 대한 자세한 내용은 Amazon Inspector로 Windows 워크로드의 보안 및 규정 준수 개선 동영상을 참조하세요.

Amazon GuardDuty

> Amazon GuardDuty는 악의적인 활동 및 무단 동작을 지속적으로 모니터링하여 Amazon S3에 저장된 AWS 계정, 워크로드 및 데이터를 보호하는 위협 탐지 서비스입니다. 클라우드를 사용하면 계정 및 네트워크 활동의 수집 및 집계가 간소화되지만 보안 팀이 잠재적 위협에 대한 이벤트 로그 데이터를 지속적으로 분석하는 데 시간이 많이 걸릴 수 있습니다.

Amazon GuardDuty를 사용하면 RDP 무차별 대입 및 포트 프로브 공격과 같은 Windows 작업자 노드에 대한 악의적인 활동을 감시할 수 있습니다.

Amazon GuardDuty를 사용하여 Windows 워크로드에 대한 위협 탐지 비디오를 보고 최적화된 EKS Windows AMI에서 CIS 벤치마크를 구현하고 실행하는 방법을 알아봅니다.

Windows용 Amazon EC2의 보안

Amazon EC2 Windows 인스턴스의 보안 모범 사례를 읽고 모든 계층에서 보안 제어를 구현하세요.

컨테이너 이미지 스캔

이미지 스캔은 애플리케이션 컨테이너 이미지에 광범위한 운영 체제 취약성이 있는지 스캔하여 보안을 개선하는 데 도움이 되는 자동화된 취약성 평가 기능입니다.

현재 Amazon Elastic Container Registry(ECR)는 Linux 컨테이너 이미지의 취약성만 스캔할 수 있습니다. 하지만 Windows 컨테이너 이미지 스캔을 위해 기존 CI/CD 파이프라인과 통합할 수 있는 타사 도구가 있습니다.

- 앵커
- PaloAlto Prisma 클라우드
- Trend Micro - Deep Security Smart Check

이러한 솔루션을 Amazon Elastic Container Repository(ECR)와 통합하는 방법에 대해 자세히 알아보려면 다음을 확인하세요.

- 고정, Amazon Elastic Container Registry(ECR)에서 이미지 스캔
- PaloAlto, Amazon Elastic Container Registry(ECR)에서 이미지 스캔
- TrendMicro, Amazon Elastic Container Registry(ECR)에서 이미지 스캔

Windows Server 버전 및 라이선스

Windows Server 버전

Amazon EKS 최적화 Windows AMI는 장기 서비스 채널(LTSC)의 Windows Server 2019 및 2022 Datacenter 에디션을 기반으로 합니다. 데이터 센터 버전은 작업자 노드에서 실행되는 컨테이너 수에 제한이 없습니다. 자세한 내용은 https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/faq 참조하십시오.

장기 서비스 채널(LTSC)

이전에는 "장기 서비스 브랜치"라고 했던이 릴리스 모델은 이미 익숙한 릴리스 모델로, 2~3년마다 새로운 메이저 버전의 Windows Server가 릴리스됩니다. 사용자는 5년간의 메인스트림 지원과 5년간의 추가 지원을 받을 수 있습니다.

라이선싱

Windows Server 기반 AMI를 사용하여 Amazon EC2 인스턴스를 시작할 때 Amazon은 라이선스 비용과 라이선스 규정 준수를 보장합니다.

로깅

컨테이너화된 애플리케이션은 일반적으로 애플리케이션 로그를 STDOUT으로 전달합니다. 컨테이너 런타임은 이러한 로그를 트래핑하고 이를 사용하여 일반적으로 파일에 씁니다. 이러한 파일이 저장되는 위치는 컨테이너 런타임 및 구성에 따라 달라집니다.

Windows 포드의 한 가지 기본적인 차이점은 STDOUT을 생성하지 않는다는 것입니다. LogMonitor를 실행하여 실행 중인 Windows 컨테이너 및 파이프 형식의 로그 출력에서 STDOUT으로 ETW(Windows용 이벤트 추적), Windows 이벤트 로그 및 기타 애플리케이션별 로그를 검색할 수 있습니다. 그런 다음 fluent-bit를 사용하여 이러한 로그를 스트리밍하거나 Amazon CloudWatch와 같은 원하는 대상으로 유창하게 스트리밍할 수 있습니다.

로그 수집 메커니즘은 Kubernetes 포드에서 STDOUT/STDERR 로그를 검색합니다. DaemonSet는 컨테이너에서 로그를 수집하는 일반적인 방법입니다. 애플리케이션과 독립적으로 로그 routing/filtering/enrichment을 관리할 수 있습니다. fluentd DaemonSet를 사용하여 이러한 로그와 기타 애플리케이션에서 생성한 로그를 원하는 로그 집계자로 스트리밍할 수 있습니다.

Windows 워크로드에서 CloudWatch로의 로그 스트리밍에 대한 자세한 내용은 여기에서 설명합니다.

로깅 권장 사항

Kubernetes에서 Windows 워크로드를 운영할 때 일반적인 로깅 모범 사례는 다르지 않습니다.

- 이러한 구조화된 형식에 대해 미리 작성된 파서가 많기 때문에 로그 항목을 더 쉽게 처리할 수 있도록 항상 구조화된 로그 항목(JSON/SYSLOG)을 로깅합니다.
- 로그 중앙 집중화 - 전용 로깅 컨테이너를 사용하여 모든 컨테이너에서 대상으로 로그 메시지를 수집하고 전달할 수 있습니다.
- 디버깅할 때를 제외하고 로그 세부 정보를 낮게 유지합니다. 세부 정보는 로깅 인프라에 많은 스트레스를 주며 노이즈에서 중요한 이벤트가 손실될 수 있습니다.
- 추적성을 위해 항상 트랜잭션/요청 ID와 함께 애플리케이션 정보를 기록합니다. Kubernetes 객체는 애플리케이션 이름을 포함하지 않으므로 예를 들어 포드 이름은 로그를 디버깅할 때 의미를 갖지 않을 `windows-twryrqyw` 수 있습니다. 이렇게 하면 집계된 로그를 사용하여 애플리케이션을 추적하고 문제를 해결하는 데 도움이 됩니다.

이러한 트랜잭션/상관성 ID를 생성하는 방법은 프로그래밍 구문에 따라 다릅니다. 그러나 매우 일반적인 패턴은 다음과 같이 MDC(매핑된 진단 컨텍스트)를 사용하여 모든 수신 요청에 고유한 트랜잭션/상관성 ID를 주입할 수 있는 로깅 Aspect/Interceptor를 사용하는 것입니다.

```
import org.slf4j.MDC;
import java.util.UUID;
Class LoggingAspect { //interceptor

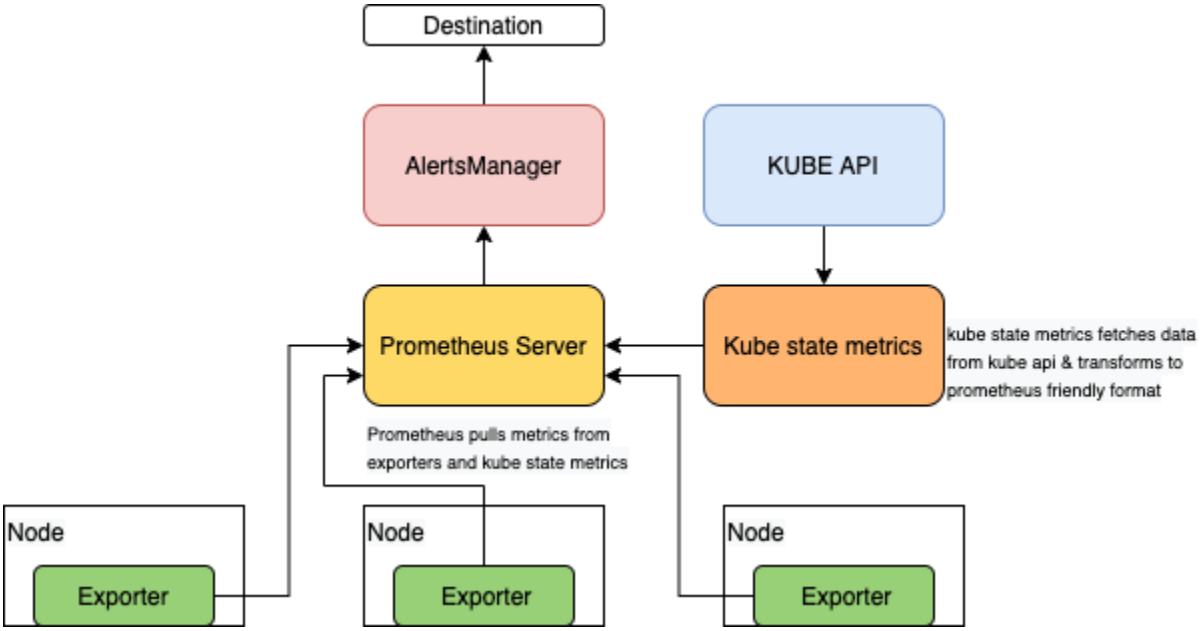
    @Before(value = "execution(* *.*(..))")
    func before(...) {
        transactionId = generateTransactionId();
        MDC.put(CORRELATION_ID, transactionId);
    }

    func generateTransactionId() {
        return UUID.randomUUID().toString();
    }
}
```

모니터링

졸업된 CNCF 프로젝트인 Prometheus는 Kubernetes에 기본적으로 통합되는 가장 인기 있는 모니터링 시스템입니다. Prometheus는 컨테이너, 포드, 노드 및 클러스터에 대한 지표를 수집합니다. 또한 Prometheus는 AlertsManager를 활용하여 클러스터에 문제가 있는 경우 경고를 프로그래밍할 수 있습니다. Prometheus는 지표 데이터를 지표 이름 및 키/값 페어로 식별되는 시계열 데이터로 저장합니다. Prometheus에는 Prometheus 쿼리 언어의 약어인 PromQL이라는 언어를 사용한 원격 쿼리가 포함되어 있습니다.

Prometheus 지표 컬렉션의 상위 수준 아키텍처는 다음과 같습니다.



Prometheus는 풀 메커니즘을 사용하고 내보내기를 사용하는 대상과 kube 상태 지표를 사용하는 Kubernetes API에서 지표를 스크레이프합니다. 즉, 애플리케이션 및 서비스는 Prometheus 형식의 지표가 포함된 HTTP(S) 엔드포인트를 노출해야 합니다. 그러면 Prometheus는 구성에 따라 이러한 HTTP(S) 엔드포인트에서 지표를 주기적으로 가져옵니다.

내보내기를 사용하면 타사 지표를 Prometheus 형식의 지표로 사용할 수 있습니다. Prometheus 내보내기는 일반적으로 각 노드에 배포됩니다. 내보내기의 전체 목록은 Prometheus 내보내기를 참조하세요. 노드 내보내기는 Linux 노드의 호스트 하드웨어 및 OS 지표를 내보내는 데 적합하지만 Windows 노드에서는 작동하지 않습니다.

안정적인 Prometheus helm 차트를 사용하는 경우 Windows 노드가 있는 혼합 노드 EKS 클러스터에서 이 내보내기는 Windows용이 아니므로 Windows 노드에 실패한 포드가 표시됩니다. Windows 작업자 풀을 별도로 취급하고 대신 Windows 작업자 노드 그룹에 Windows 내보내기를 설치해야 합니다.

Windows 노드에 대한 Prometheus 모니터링을 설정하려면 Windows 서버 자체에 WMI 내보내기를 다운로드하여 설치한 다음 Prometheus 구성 파일의 스크레이프 구성 내에서 대상을 설정해야 합니다. 릴리스 페이지에서는 사용 가능한 모든 .msi 설치 프로그램과 각 기능 세트 및 버그 수정을 제공합니다. 설치 관리자는 Windows_exporter를 Windows 서비스로 설정하고 Windows 방화벽에서 예외를 생성합니다. 설치 관리자가 파라미터 없이 실행되는 경우, 활성화된 수집기, 포트 등에 대한 기본 설정으로 내보내기가 실행됩니다.

노드를 부트스트랩하거나 원하는 구성 관리 도구(예: Chef, Ansible, SSM 등)를 사용하여 Windows 노드에 Windows Exporter를 설치하는 동안 taints/tolerations 또는 RuntimeClass를 사용하여 linux 노드에만 선택적으로 배포하는 방법을 제안하는이 가이드의 예약 모범 사례 섹션을 확인할 수 있습니다.

노드 내보내기가 데몬 세트 로 설치된 Linux 노드와 달리 Windows 노드에서는 WMI 내보내기가 호스트 자체에 설치됩니다. 내보내기 도구는 CPU 사용량, 메모리 및 디스크 I/O 사용량과 같은 지표를 내보내고 IIS 사이트 및 애플리케이션, 네트워크 인터페이스 및 서비스를 모니터링하는 데 사용할 수도 있습니다.

windows_exporter는 기본적으로 활성화된 수집기의 모든 지표를 노출합니다. 이는 오류를 방지하기 위해 지표를 수집하는 데 권장되는 방법입니다. 그러나 고급 사용의 경우 windows_exporter는 지표를 필터링하기 위한 선택적 수집기 목록을 전달할 수 있습니다. Prometheus 구성에서 collect[] 파라미터를 사용하면 됩니다.

Windows의 기본 설치 단계에는 필터링하려는 수집기와 같은 인수를 사용하여 부트스트래핑 프로세스 중에 서비스로 내보내기를 다운로드하고 시작하는 작업이 포함됩니다.

```
> Powershell Invoke-WebRequest https://github.com/prometheus-community/
windows_exporter/releases/download/v0.13.0/windows_exporter-0.13.0-amd64.msi -OutFile
 <DOWNLOADPATH>

> msiexec /i <DOWNLOADPATH>
 ENABLED_COLLECTORS="cpu,cs,logical_disk,net,os,system,container,memory"
```

기본적으로 포트 9182의 /metrics 엔드포인트에서 지표를 스크레이프할 수 있습니다. 이때 Prometheus는 Prometheus 구성에 다음 scrape_config를 추가하여 지표를 사용할 수 있습니다.

```
scrape_configs:
    - job_name: "prometheus"
      static_configs:
        - targets: ['localhost:9090']
    ...
    - job_name: "wmi_exporter"
      scrape_interval: 10s
      static_configs:
        - targets: ['<windows-node1-ip>:9182', '<windows-node2-ip>:9182', ...]
```

를 사용하여 Prometheus 구성이 다시 로드됩니다.

```
> ps aux | grep prometheus
> kill HUP <PID>
```

대상을 추가하는 더 나은 권장 방법은 ServiceMonitor에 대한 정의를 제공하는 ServiceMonitor 연산자] 라는 사용자 지정 리소스 정의와 정의한 ServiceMonitors를 활성화하고 필요한 Prometheus 구성을 자동으로 빌드하는 컨트롤러를 사용하는 것입니다.

Kubernetes 서비스 그룹을 모니터링하는 방법을 선언적으로 지정하는 ServiceMonitor는 Kubernetes 내에서 지표를 스크레이프하려는 애플리케이션을 정의하는 데 사용됩니다. ServiceMonitor 내에서 운영자가 Kubernetes 서비스를 식별하는 데 사용할 수 있는 Kubernetes 레이블을 지정합니다. 그러면 모니터링하려는 포드가 식별됩니다.

ServiceMonitor를 활용하려면 특정 Windows 대상, 헤드리스 서비스 및 Windows 노드용 ServiceMontor를 가리키는 엔드포인트 객체를 생성합니다.

```
apiVersion: v1
kind: Endpoints
metadata:
  labels:
    k8s-app: wmiexporter
  name: wmiexporter
  namespace: kube-system
subsets:
- addresses:
  - ip: NODE-ONE-IP
    targetRef:
      kind: Node
      name: NODE-ONE-NAME
  - ip: NODE-TWO-IP
    targetRef:
      kind: Node
      name: NODE-TWO-NAME
  - ip: NODE-THREE-IP
    targetRef:
      kind: Node
      name: NODE-THREE-NAME
  ports:
  - name: http-metrics
    port: 9182
    protocol: TCP

---
apiVersion: v1
kind: Service ##Headless Service
metadata:
  labels:
    k8s-app: wmiexporter
  name: wmiexporter
  namespace: kube-system
spec:
```

```
  clusterIP: None
  ports:
  - name: http-metrics
    port: 9182
    protocol: TCP
    targetPort: 9182
  sessionAffinity: None
  type: ClusterIP

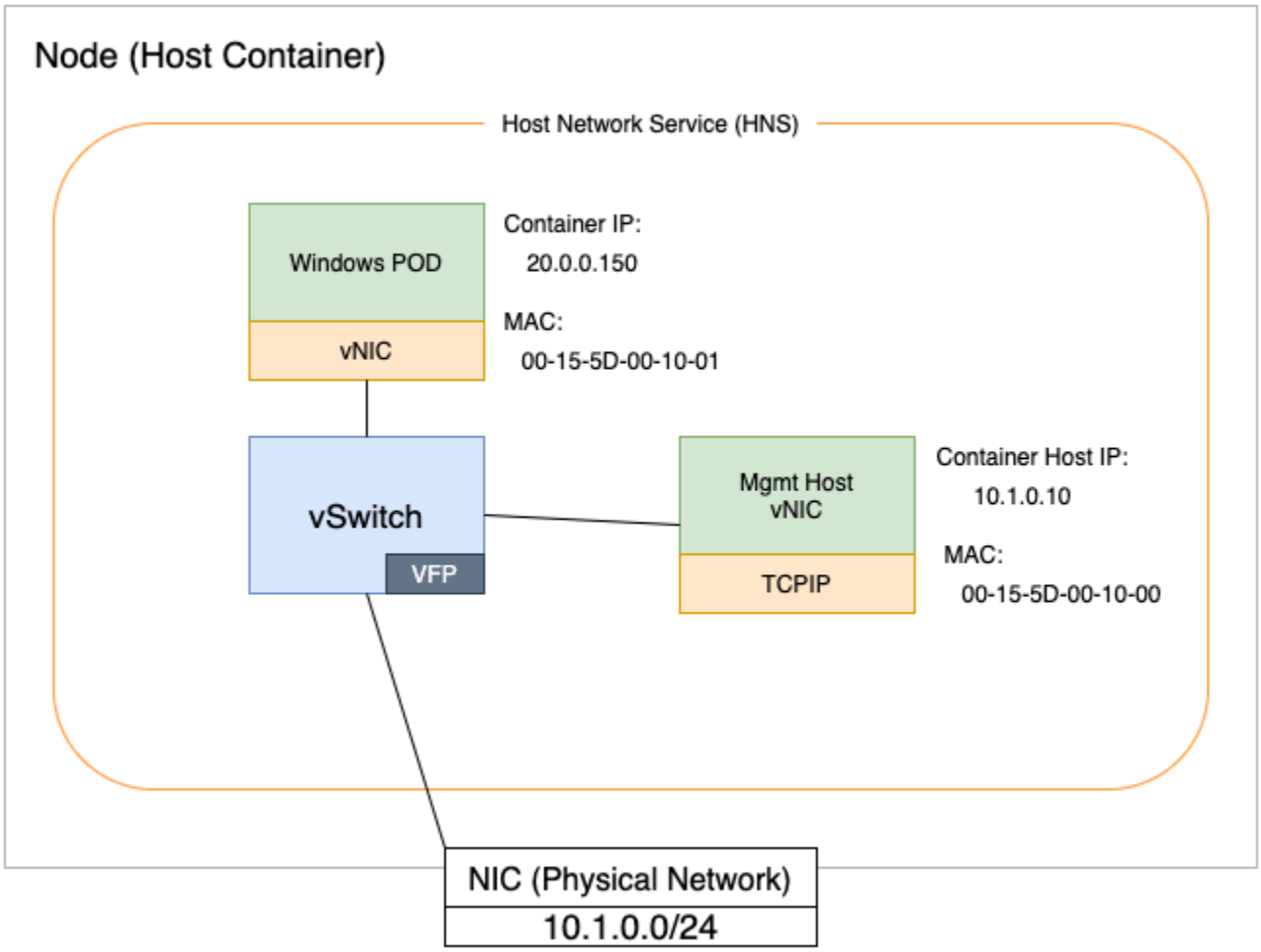
---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor ##Custom ServiceMonitor Object
metadata:
  labels:
    k8s-app: wmiexporter
  name: wmiexporter
  namespace: monitoring
spec:
  endpoints:
  - interval: 30s
    port: http-metrics
  jobLabel: k8s-app
  namespaceSelector:
    matchNames:
    - kube-system
  selector:
    matchLabels:
      k8s-app: wmiexporter
```

운영자 및 ServiceMonitor 사용에 대한 자세한 내용은 공식 운영자 설명서를 확인하세요. Prometheus는 많은 서비스 검색 옵션을 사용하여 동적 대상 검색을 지원합니다.

Windows 네트워킹

Windows 컨테이너 네트워킹 개요

Windows 컨테이너는 Linux 컨테이너와 근본적으로 다릅니다. Linux 컨테이너는 네임스페이스, 유니온 파일 시스템 및 cgroup과 같은 Linux 구문을 사용합니다. Windows에서는 이러한 구문이 호스트 컴퓨팅 서비스(HCS)에 의해 컨테이너화된에서 추출됩니다. HCS는 Windows에서 컨테이너 구현 위에 있는 API 계층 역할을 합니다. Windows 컨테이너는 노드에서 네트워크 토폴로지를 정의하는 호스트 네트워크 서비스(HNS)도 활용합니다.



네트워킹 관점에서 HCS와 HNS는 Windows 컨테이너가 가상 머신처럼 작동하도록 합니다. 예를 들어 각 컨테이너에는 위의 다이어그램과 같이 Hyper-V 가상 스위치(vSwitch)에 연결된 가상 네트워크 어댑터(vNIC)가 있습니다.

IP 주소 관리

Amazon EKS의 노드는 ENI(Elastic Network Interface)를 사용하여 AWS VPC 네트워크에 연결합니다. 현재 Windows 작업자 노드당 단일 ENI만 지원됩니다. Windows 노드의 IP 주소 관리는 컨트롤 플레인에서 실행되는 VPC 리소스 컨트롤러에서 수행됩니다. Windows 노드의 IP 주소 관리를 위한 워크플로에 대한 자세한 내용은 여기에서 확인할 수 있습니다.

Windows 작업자 노드가 지원할 수 있는 포드 수는 노드의 크기와 사용 가능한 IPv4 주소 수에 따라 결정됩니다. 노드에서 사용할 수 있는 IPv4 주소를 다음과 같이 계산할 수 있습니다.

- 기본적으로 보조 IPv4 주소만 ENI에 할당됩니다. 이러한 경우:

```
Total IPv4 addresses available for Pods = Number of supported IPv4 addresses in the
 primary interface - 1
```

하나의 IPv4 주소가 ENI의 기본 주소로 사용되므로 포드에 할당할 수 없으므로 총 개수에서 하나를 뺍니다.

- 접두사 위임 기능을 활성화하여 클러스터가 높은 포드 밀도에 맞게 구성된 경우

```
Total IPv4 addresses available for Pods = (Number of supported IPv4 addresses in the
 primary interface - 1) * 16
```

여기서는 보조 IPv4 주소를 할당하는 대신 VPC 리소스 컨트롤러가 할당/28 `prefixes`하므로 사용 가능한 전체 IPv4 주소 수가 16회 부스팅됩니다.

위의 공식을 사용하여 다음과 같이 m5.large 인스턴스를 기반으로 노드된 Windows 작업자의 최대 포드를 계산할 수 있습니다.

- 기본적으로 보조 IP 모드에서 실행되는 경우 -

```
10 secondary IPv4 addresses per ENI - 1 = 9 available IPv4 addresses
```

- 사용 시 `prefix delegation`-

```
(10 secondary IPv4 addresses per ENI - 1) * 16 = 144 available IPv4 addresses
```

인스턴스 유형이 지원할 수 있는 IP 주소 수에 대한 자세한 내용은 인스턴스 유형별 네트워크 인터페이스당 IP 주소를 참조하세요.

또 다른 주요 고려 사항은 네트워크 트래픽의 흐름입니다. Windows를 사용하면 서비스가 100개 이상인 노드에서 포트가 고갈될 위험이 있습니다. 이 조건이 발생하면 노드에서 다음 메시지와 함께 오류가 발생하기 시작합니다.

“정책 생성 실패: Win32에서 hcnCreateLoadBalancer 실패: 지정된 포트가 이미 있습니다.”

이 문제를 해결하기 위해 DSR(Direct Server Return)을 활용합니다. DSR은 비대칭 네트워크 로드 분산을 구현한 것입니다. 즉, 요청 및 응답 트래픽은 서로 다른 네트워크 경로를 사용합니다. 이 기능은 포

드 간 통신 속도를 높이고 포트 소진 위험을 줄입니다. 따라서 Windows 노드에서 DSR을 활성화하는 것이 좋습니다.

DSR은 Windows Server SAC EKS 최적화 AMIs에서 기본적으로 활성화됩니다. Windows Server 2019 LTSC EKS 최적화 AMIs 경우 아래 스크립트를 사용하고 Windows Server 2019 Full 또는 Core를 `eksctl` nodeGroup의 amiFamily로 사용하여 인스턴스 프로비저닝 중에 활성화해야 합니다. 자세한 내용은 eksctl 사용자 지정 AMI를 참조하세요.

```
nodeGroups:
- name: windows-ng
  instanceType: c5.xlarge
  minSize: 1
  volumeSize: 50
  amiFamily: WindowsServer2019CoreContainer
  ssh:
    allow: false
```

Windows Server 2019 이상에서 DSR을 활용하려면 인스턴스 시작 중에 다음 kube-proxy 플래그를 지정해야 합니다. 자체 관리형 노드 그룹 시작 템플릿과 연결된 사용자 데이터 스크립트를 조정하여이 작업을 수행할 수 있습니다.

```
<powershell>
[string]$EKSBinDir = "$env:ProgramFiles\Amazon\EKS"
[string]$EKSBootstrapScriptName = 'Start-EKSBootstrap.ps1'
[string]$EKSBootstrapScriptFile = "$EKSBinDir\$EKSBootstrapScriptName"
(Get-Content $EKSBootstrapScriptFile).replace('"--proxy-mode=kernelspace",', '"--proxy-
mode=kernelspace", "--feature-gates WinDSR=true", "--enable-dsr",') | Set-Content
 $EKSBootstrapScriptFile
& $EKSBootstrapScriptFile -EKSClusterName "eks-windows" -APIServerEndpoint "https://
<REPLACE-EKS-CLUSTER-CONFIG-API-SERVER>" -Base64ClusterCA "<REPLACE-EKSCLUSTER-
CONFIG-DETAILS-CA>" -DNSClusterIP "172.20.0.10" -KubeletExtraArgs "--node-
labels=alpha.eksctl.io/cluster-name=eks-windows,alpha.eksctl.io/nodegroup-name=windows-
ng-ltsc2019 --register-with-taints=" 3>&1 4>&1 5>&1 6>&1
</powershell>
```

DSR 활성화는 Microsoft 네트워킹 블로그 및 AWS Lab의 Windows 컨테이너의 지침에 따라 확인할 수 있습니다.

```
PS C:\temp> (Get-Content C:\temp\pmsx1eau.0ia\policy.txt | ConvertFrom-JSON).Policies


ExternalPort : 53
InternalPort : 53
IsDSR        : True
Protocol     : 17
Type         : ELB
VIPs         : {172.20.0.10}

ExternalPort : 53
InternalPort : 53
IsDSR        : True
Protocol     : 6
Type         : ELB
VIPs         : {172.20.0.10}

ExternalPort : 443
InternalPort : 443
IsDSR        : True
Protocol     : 6
Type         : ELB
VIPs         : {172.20.0.1}
```

서브넷에 사용 가능한 IPv4 주소를 보존하고 낭비를 최소화하는 것이 중요한 경우 일반적으로 Windows용 접두사 모드 - 피해야 할 경우에 언급된 대로 접두사 위임 모드를 사용하지 않는 것이 좋습니다. 접두사 위임을 계속 사용하려는 경우 서브넷의 IPv4 주소 사용률을 최적화하는 단계를 수행할 수 있습니다. IPv4 주소 요청 및 할당 프로세스를 미세 조정하는 방법에 대한 자세한 지침은 접두사 위임을 위한 파라미터 구성을 참조하세요. 이러한 구성을 조정하면 IPv4 주소 보존과 접두사 위임의 포드 밀도 이점 간에 균형을 맞출 수 있습니다.

보조 IPv4 주소 할당의 기본 설정을 사용하는 경우 현재 VPC 리소스 컨트롤러가 IPv4 주소를 요청하고 할당하는 방법을 조작하는 데 지원되는 구성이 없습니다. 보다 구체적으로, `minimum-ip-target` 및 `warm-ip-target`는 접두사 위임 모드에서만 지원됩니다. 또한 보조 IP 모드에서는 인터페이스에서 사용 가능한 IP 주소에 따라 VPC 리소스 컨트롤러가 일반적으로 노드에 미사용 IPv4 주소 3개를 할당하여 포드 시작 시간을 단축하기 위해 웜 IPs 유지합니다. 사용하지 않는 웜 IP 주소의 IP 낭비를 최소화하려면 ENI의 IP 주소 용량을 최대한 많이 사용하도록 지정된 Windows 노드에서 더 많은 포드를 예약하는 것이 좋습니다. 더 명시적으로 말하면 ENI의 모든 IPs 주소가 노드에서 이미 사용 중이고 포드를 실행하는 경우 웜 미사용 IP를 사용하지 않도록 할 수 있습니다. 서브넷의 IP 주소 가용성 제약 조건을 해결하는 데 도움이 되는 또 다른 해결 방법은 서브넷 크기를 늘리거나 Windows 노드를 전용 서브넷으로 분리하는 것입니다.

또한 IPv6는 현재 Windows 노드에서 지원되지 않는다는 점에 유의해야 합니다.

컨테이너 네트워크 인터페이스(CNI) 옵션

AWSVPC CNI는 Windows 및 Linux 작업자 노드용 사실상의 CNI 플러그인입니다. AWSVPC CNI는 많은 고객의 요구 사항을 충족하지만 IP 소진을 방지하기 위해 오버레이 네트워크와 같은 대안을 고려해야 하는 경우가 있을 수 있습니다. 이러한 경우 AWSVPC CNI 대신 Calico CNI를 사용할 수 있습니다. Project Calico는 Tigera에서 개발한 오픈 소스 소프트웨어입니다. 이 소프트웨어에는 EKS와 함께 작동하는 CNI가 포함되어 있습니다. EKS에 Calico CNI를 설치하는 지침은 Project Calico EKS 설치 페이지에서 확인할 수 있습니다.

네트워크 정책

Kubernetes 클러스터의 포드 간 개방형 통신의 기본 모드에서 네트워크 정책에 따라 액세스를 제한하는 것으로 변경하는 것이 모범 사례로 간주됩니다. 오픈 소스 Project Calico는 Linux 및 Windows 노드 모두에서 작동하는 네트워크 정책을 강력하게 지원합니다. 이 기능은 별개이며 Calico CNI 사용에 종속되지 않습니다. 따라서 Calico를 설치하고 네트워크 정책 관리에 사용하는 것이 좋습니다.

EKS에 Calico를 설치하는 지침은 Amazon EKS에 Calico 설치 페이지에서 확인할 수 있습니다.

또한 Amazon EKS 보안 모범 사례 가이드 - 네트워크 섹션에 제공된 조언은 Windows 작업자 노드가 있는 EKS 클러스터에도 동일하게 적용되지만, "포드용 보안 그룹"과 같은 일부 기능은 현재 Windows에서 지원되지 않습니다.

OOM 오류 방지

Windows에는 Linux처럼 out-of-memory 프로세스 킬러가 없습니다. Windows는 항상 모든 사용자 모드 메모리 할당을 가상으로 취급하며 페이지 파일은 필수입니다. 순 효과는 Windows가 Linux와 동일한 방식으로 메모리 조건을 벗어나지 않는다는 것입니다. 프로세스는 메모리 부족(OOM) 종료되는 대신 디스크로 페이징됩니다. 메모리가 과다 프로비저닝되고 모든 물리적 메모리가 소진되면 페이징으로 인해 성능이 저하될 수 있습니다.

시스템 및 kubelet 메모리 예약

kubelet, 컨테이너 런타임 등과 같은 kubernetes 시스템 데몬에 대한 리소스 예약을 `--kubelet-reserve` 캡처하고 sshd, udev 등과 같은 OS 시스템 데몬에 대한 리소스 예약을 `--system-reserve` 캡처하는 Linux와 다릅니다. Windows에서 이러한 플래그는 노드에서 실행되는 kubelet 또는 프로세스에 대한 메모리 제한을 캡처하고 설정하지 않습니다.

그러나 이러한 플래그를 결합하여 NodeAllocatable을 관리하여 포드 매니페스트 메모리 리소스 제한으로 노드의 용량을 줄여 포드당 메모리 할당을 제어할 수 있습니다. 이 전략을 사용하면 Windows

노드에서 out-of-memory(OOM)을 최소화하는 메커니즘뿐만 아니라 메모리 할당을 더 잘 제어할 수 있습니다.

Windows 노드에서 모범 사례는 OS 및 프로세스를 위해 최소 2GB의 메모리를 예약하는 것입니다. `--kubelet-reserve` 및/또는 `--system-reserve`를 사용하여 NodeAllocatable을 줄입니다.

Amazon EKS 자체 관리형 Windows 노드 설명서에 따라 CloudFormation 템플릿을 사용하여 kubelet 구성을 사용자 지정하여 새 Windows 노드 그룹을 시작합니다. CloudFormation에는와 `BootstrapArguments` 동일한 라는 요소가 있습니다`KubeletExtraArgs`. 다음 플래그 및 값과 함께를 사용합니다.

```
--kube-reserved memory=0.5Gi,ephemeral-storage=1Gi --system-
reserved memory=1.5Gi,ephemeral-storage=1Gi --eviction-hard
 memory.available<200Mi,nodefs.available<10%"
```

eksctl이 배포 도구인 경우 다음 설명서를 확인하여 kubelet 구성을 사용자 지정합니다. https://eksctl.io/usage/customizing-the-kubelet/

Windows 컨테이너 메모리 요구 사항

Microsoft 설명서에 따르면 NANO용 Windows Server 기본 이미지에는 최소 30MB가 필요한 반면 Server Core에는 45MB가 필요합니다. .NET Framework, Web Services as IIS 및 애플리케이션과 같은 Windows 구성 요소를 추가하면 이러한 수가 증가합니다.

Windows 컨테이너 이미지, 즉 기본 이미지와 애플리케이션 계층에 필요한 최소 메모리 양을 알고 포드 사양에서 컨테이너의 리소스/요청으로 설정해야 합니다. 또한 애플리케이션 문제가 발생할 경우 포드가 사용 가능한 모든 노드 메모리를 소비하지 않도록 제한을 설정해야 합니다.

아래 예제에서 Kubernetes 스케줄러가 노드에 포드를 배치하려고 하면 포드의 요청을 사용하여 예약에 사용할 수 있는 리소스가 충분한 노드를 결정합니다.

```
spec:
 - name: iis
   image: mcr.microsoft.com/windows/servercore/iis:windowsservercore-ltsc2019
   resources:
     limits:
       cpu: 1
       memory: 800Mi
     requests:
       cpu: .1
       memory: 128Mi
```

결론

이 접근 방식을 사용하면 메모리 소진 위험을 최소화하지만 메모리 소진을 방지하지는 않습니다. Amazon CloudWatch 지표를 사용하면 메모리 소진 발생 시 알림 및 문제 해결을 설정할 수 있습니다.

Windows Server 및 컨테이너 패치 적용

Windows Server 패치 적용은 Windows 관리자를 위한 표준 관리 작업입니다. 이는 Amazon System Manager - Patch Manager, WSUS, System Center Configuration Manager 등과 같은 다양한 도구를 사용하여 수행할 수 있습니다. 그러나 Amazon EKS 클러스터의 Windows 노드는 일반 Windows 서버로 취급해서는 안 됩니다. 변경 불가능한 서버로 취급해야 합니다. 간단히 말해서 기존 노드를 업데이트하지 말고 새로 업데이트된 AMI를 기반으로 새 노드를 시작하면 됩니다.

EC2 Image Builder를 사용하면 레시피를 생성하고 구성 요소를 추가하여 AMIs 빌드를 자동화할 수 있습니다.

다음 예제에서는 AWS에서 구축한 기존 구성 요소(Amazon 관리형)와 사용자가 생성한 구성 요소(내 소유)일 수 있는 구성 요소를 보여줍니다. update-windows라는 Amazon 관리형 구성 요소에 주의하세요. 그러면 EC2 Image Builder 파이프라인을 통해 AMI를 생성하기 전에 Windows Server가 업데이트됩니다.

Associated components

Search components

1

Component name	Version	Description	Date created	Platform	Owner	ARN
eks-optimized-ami-windows	1.20.0	Installs EKS-optimized Windows artifacts for EKS version 1.20. This includes kubelet version 1.20.4 and Docker version 20.10.6.	Aug 18, 2021 6:33 PM	Windows	Amazon	arn:aws:imagebuilder:us-east-1:aws:component/eks-optimized-ami-windows/1.20.0/1
update-windows	1.0.1	Updates Windows with the latest security updates.	Aug 18, 2021 6:33 PM	Windows	Amazon	arn:aws:imagebuilder:us-east-1:aws:component/update-windows/1.0.1/1
Docker Pull	1.0.1	-	Dec 22, 2021 12:08 PM	Windows	[redacted]	arn:aws:imagebuilder:us-east-[redacted]component/docker-pull/1.0.1/1
simple-boot-test-windows	1.0.0	Executes a simple boot test.	Nov 30, 2019 11:39 PM	Windows	Amazon	arn:aws:imagebuilder:us-east-1:aws:component/simple-boot-test-windows/1.0.0/1

EC2 Image Builder를 사용하면 Amazon Managed Public AMI를 기반으로 AMIs 빌드하고 비즈니스 요구 사항에 맞게 사용자 지정할 수 있습니다. 그런 다음 이러한 AMIs 시작 템플릿과 연결하여 새 AMI를 EKS 노드 그룹에서 생성한 Auto Scaling 그룹에 연결할 수 있습니다. 완료되면 기존 Windows 노드 종료를 시작할 수 있으며 업데이트된 새 AMI를 기반으로 새 노드가 시작됩니다.

Windows 이미지 푸시 및 가져오기

Amazon은 캐시된 두 개의 Windows 컨테이너 이미지가 포함된 EKS 최적화 AMIs를 게시합니다.

```
mcr.microsoft.com/windows/servercore
```

```
mcr.microsoft.com/windows/nanoserver
```

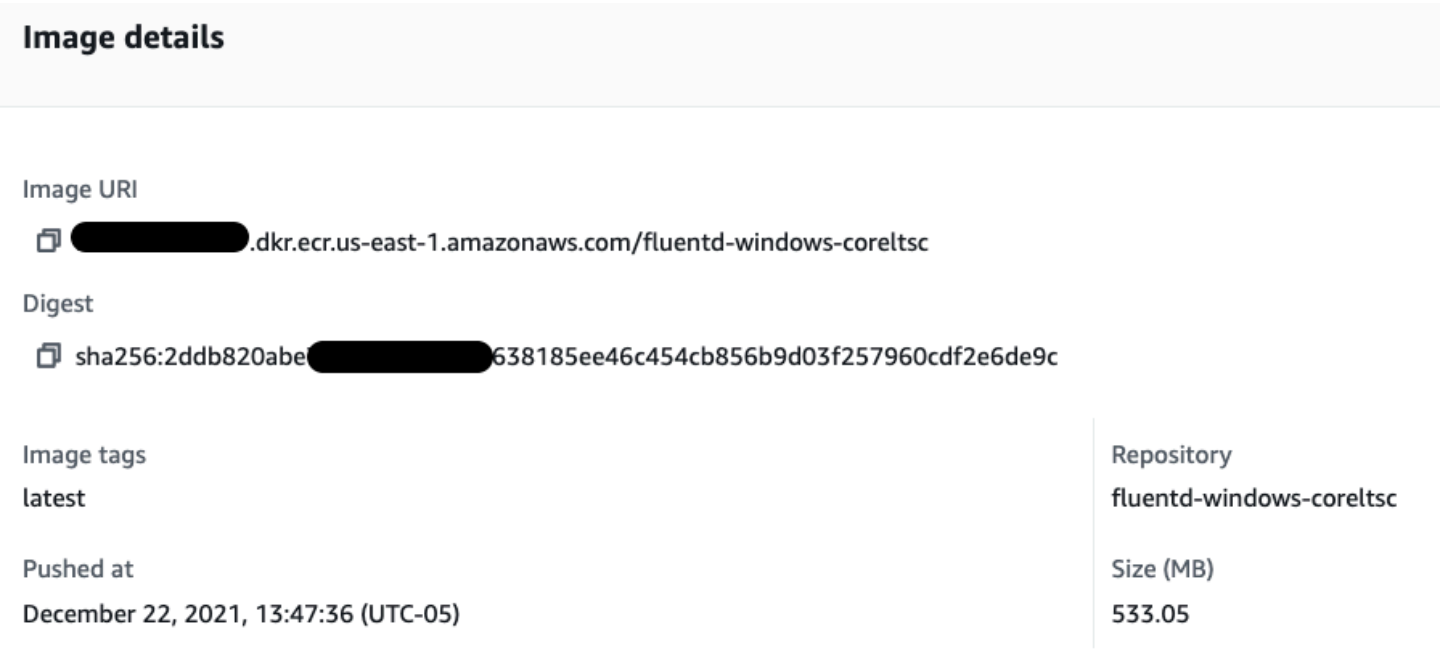
```
REPOSITORY                                                                TAG                      IMAGE ID       CREATED         SIZE
mcr.microsoft.com/windows/servercore                                      ltsc2019                 617dd5ead5b8   6 weeks ago     5.74GB
mcr.microsoft.com/windows/nanoserver                                      1809                     8334af2fa7ba   6 weeks ago     257MB
[redacted].dkr.ecr.us-east-1.amazonaws.com/fluentd-windows-coreltsc       latest                   721afca2c725   7 weeks ago     6.96GB
fluent/fluentd                                                            v1.14-windows-ltsc2019-1 721afca2c725   7 weeks ago     6.96GB
amazonaws.com/eks/pause-windows                                           latest                   6392f69ae6e7   10 months ago   255MB
```

캐시된 이미지는 기본 OS의 업데이트에 따라 업데이트됩니다. Microsoft가 Windows 컨테이너 기본 이미지에 직접 영향을 미치는 새 Windows 업데이트를 릴리스하면 기본 OS에서 일반 Windows 업데이트로 업데이트가 시작됩니다. 환경을 up-to-date 상태로 유지하면 노드 및 컨테이너 수준에서 보다 안전한 환경이 제공됩니다.

Windows 컨테이너 이미지의 크기는 푸시/풀 작업에 영향을 미치므로 컨테이너 시작 시간이 느려질 수 있습니다. Windows 컨테이너 이미지를 캐싱하면 컨테이너 시작 대신 AMI 빌드 생성 시 비용이 많이 드는 I/O 작업(파일 추출)이 발생할 수 있습니다. 따라서 필요한 모든 이미지 계층이 AMI에서 추출되고 사용할 준비가 되어 Windows 컨테이너가 시작되고 트래픽 수락을 시작할 수 있는 시간이 단축됩니다. 푸시 작업 중에는 이미지를 구성하는 계층만 리포지토리에 업로드됩니다.

다음 예제는 Amazon ECR에서 fluentd-windows-sac2004 이미지의 크기가 390.18MB에 불과함을 보여줍니다. 푸시 작업 중에 발생한 업로드의 양입니다.

다음 예제는 Amazon ECR 리포지토리로 푸시된 유창한 Windows ltsc 이미지를 보여줍니다. ECR에 저장된 계층의 크기는 533.05MB입니다.



아래 `docker image ls`의 출력인 fluentd v1.14-windows-ltsc2019-1의 크기는 디스크에서 6.96GB이지만 해당 양의 데이터를 다운로드하고 추출했다는 의미는 아닙니다.

실제로 풀 작업 중에는 압축된 533.05MB만 다운로드되고 추출됩니다.

```
REPOSITORY                                                                  TAG
            IMAGE ID       CREATED         SIZE
111122223333.dkr.ecr.us-east-1.amazonaws.com/fluentd-windows-coreltsc   latest
            721afca2c725   7 weeks ago     6.96GB
fluent/fluentd                                                              v1.14-windows-
ltsc2019-1   721afca2c725   7 weeks ago     6.96GB
amazonaws.com/eks/pause-windows                                             latest
            6392f69ae6e7   10 months ago   255MB
```

크기 열에는 이미지의 전체 크기인 6.96GB가 표시됩니다. 분류:

- Windows Server Core 2019 LTSC 기본 이미지 = 5.74GB
- Fluentd 비압축 기본 이미지 = 6.96GB
- 디스크 차이 = 1.2GB
- Fluentd 압축된 최종 이미지 ECR = 533.05MB

기본 이미지가 이미 로컬 디스크에 있으므로 디스크의 총량은 1.2GB가 추가됩니다. 다음에 크기 열에 GBs의 양이 표시되면 너무 많이 걱정하지 마세요. 70% 이상이 이미 캐시된 컨테이너 이미지로 디스크에 있을 가능성이 높습니다.

레퍼런스

EC2 이미지 빌더 및 이미지 캐시 전략을 사용하여 Windows 컨테이너 시작 시간 단축

이기종 워크로드 실행

Kubernetes는 동일한 클러스터에 Linux 노드와 Windows 노드를 혼합할 수 있는 이기종 클러스터를 지원합니다. 해당 클러스터 내에서 Linux에서 실행되는 포드와 Windows에서 실행되는 포드가 혼합되어 있을 수 있습니다. 동일한 클러스터에서 여러 버전의 Windows를 실행할 수도 있습니다. 그러나이 결정을 내릴 때 고려해야 할 몇 가지 요인(아래 참조)이 있습니다.

노드PODs 할당 모범 사례

Linux 및 Windows 워크로드를 각 OS별 노드에 유지하려면 노드 선택기와 테인트/내결함의 조합을 사용해야 합니다. 이기종 환경에서 워크로드를 예약하는 주요 목표는 기존 Linux 워크로드와의 호환성 저하를 방지하는 것입니다.

OS별 워크로드가 적절한 컨테이너 호스트에 도달하도록 보장

사용자는 nodeSelectors. 오늘날 모든 Kubernetes 노드에는 다음과 같은 기본 레이블이 있습니다.

```
kubernetes.io/os = [windows|linux]
kubernetes.io/arch = [amd64|arm64|...]
```

포드 사양에와 같은 nodeSelector가 포함되지 않은 경우 모든 호스트"kubernetes.io/os": windows, Windows 또는 Linux에서 포드를 예약할 수 있습니다. Windows 컨테이너는 Windows에서만 실행할 수 있고 Linux 컨테이너는 Linux에서만 실행할 수 있으므로 이는 문제가 될 수 있습니다.

엔터프라이즈 환경에서는 Linux 컨테이너에 대한 많은 수의 기존 배포와 Helm 차트와 같은 off-the-shelf 구성의 에코시스템을 사용하는 것이 드문 일이 아닙니다. 이러한 상황에서는 배포의 nodeSelectors. 대안은 Taints를 사용하는 것입니다.

예: `--register-with-taints='os=windows:NoSchedule'`

EKS를 사용하는 경우 eksctl은 clusterConfig를 통해 테인트를 적용하는 방법을 제공합니다.

```
NodeGroups:
  - name: windows-ng
    amiFamily: WindowsServer2022FullContainer
    ...
    labels:
      nodeclass: windows2022
    taints:
      os: "windows:NoSchedule"
```

모든 Windows 노드에 테인트를 추가하면 스케줄러는 테인트를 허용하지 않는 한 해당 노드에서 포드를 예약하지 않습니다. 포드 매니페스트 예제:

```
nodeSelector:
    kubernetes.io/os: windows
tolerations:
    - key: "os"
      operator: "Equal"
      value: "windows"
      effect: "NoSchedule"
```

동일한 클러스터에서 여러 Windows 빌드 처리

각 포드에서 사용하는 Windows 컨테이너 기본 이미지는 노드와 동일한 커널 빌드 버전과 일치해야 합니다. 동일한 클러스터에서 여러 Windows Server 빌드를 사용하려면 추가 노드 레이블, nodeSelectors를 설정하거나 windows-build라는 레이블을 활용해야 합니다.

Kubernetes 1.17은 자동으로 새 레이블 node.kubernetes.io/windows-build 추가하여 동일한 클러스터에서 여러 Windows 빌드의 관리를 간소화합니다. 이전 버전을 실행하는 경우 Windows 노드에이 레이블을 수동으로 추가하는 것이 좋습니다.

이 레이블은 호환성을 위해 일치해야 하는 Windows 메이저, 마이너 및 빌드 번호를 반영합니다. 다음은 현재 각 Windows Server 버전에 사용되는 값입니다.

Windows Server가 기본 릴리스 채널로 장기 서비스 채널(LTSC)로 이동하고 있다는 점에 유의해야 합니다. Windows Server 반기 채널(SAC)은 2022년 8월 9일에 사용 중지되었습니다. 향후 Windows Server의 SAC 릴리스는 없습니다.

제품 이름	빌드 번호(들)
서버 전체 2022 LTSC	20348년 10월 0일
서버 코어 2019 LTSC	10.0.17763

다음 명령을 통해 OS 빌드 버전을 확인할 수 있습니다.

```
kubectl get nodes -o wide
```

KERNEL-VERSION 출력은 Windows OS 빌드 버전과 일치합니다.

```
NAME                            STATUS   ROLES    AGE   VERSION               INTERNAL-
IP   EXTERNAL-IP     OS-IMAGE                         KERNEL-VERSION
 CONTAINER-RUNTIME
ip-10-10-2-235.ec2.internal   Ready    <none>   23m   v1.24.7-eks-fb459a0
 10.10.2.235   3.236.30.157    Windows Server 2022 Datacenter   10.0.20348.1607
         containerd://1.6.6
ip-10-10-31-27.ec2.internal   Ready    <none>   23m   v1.24.7-eks-fb459a0
 10.10.31.27   44.204.218.24   Windows Server 2019 Datacenter   10.0.17763.4131
         containerd://1.6.6
```

```
ip-10-10-7-54.ec2.internal    Ready    <none>   31m   v1.24.11-eks-a59e1f0   10.10.7.54
    3.227.8.172     Amazon Linux 2                  5.10.173-154.642.amzn2.x86_64
 containerd://1.6.19
```

아래 예제에서는 다른 Windows 노드 그룹 OS 버전을 실행할 때 올바른 Windows 빌드 버전과 일치하도록 포드 매니페스트에 추가 nodeSelector를 적용합니다.

```
nodeSelector:
    kubernetes.io/os: windows
    node.kubernetes.io/windows-build: '10.0.20348'
tolerations:
    - key: "os"
    operator: "Equal"
    value: "windows"
    effect: "NoSchedule"
```

RuntimeClass를 사용하여 포드 매니페스트에서 NodeSelector 및 허용치 간소화

RuntimeClass를 사용하여 테인트 및 허용치 사용 프로세스를 간소화할 수도 있습니다. 이러한 테인트와 허용치를 캡슐화하는 데 사용되는 RuntimeClass 객체를 생성하여이 작업을 수행할 수 있습니다.

다음 매니페스트를 실행하여 RuntimeClass를 생성합니다.

```
apiVersion: node.k8s.io/v1beta1
kind: RuntimeClass
metadata:
  name: windows-2022
handler: 'docker'
scheduling:
  nodeSelector:
    kubernetes.io/os: 'windows'
    kubernetes.io/arch: 'amd64'
    node.kubernetes.io/windows-build: '10.0.20348'
  tolerations:
  - effect: NoSchedule
    key: os
    operator: Equal
    value: "windows"
```

Runtimeclass가 생성되면를 포드 매니페스트의 사양으로 사용하여 할당합니다.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: iis-2022
  labels:
    app: iis-2022
spec:
  replicas: 1
  template:
    metadata:
      name: iis-2022
      labels:
        app: iis-2022
    spec:
      runtimeClassName: windows-2022
      containers:
      - name: iis
```

관리형 노드 그룹 지원

고객이 보다 간소화된 방식으로 Windows 애플리케이션을 실행할 수 있도록 AWS는 2022년 12월 15일에 Windows 컨테이너에 대한 Amazon EKS Managed Node Group(MNG) 지원 지원을 시작했습니다. 운영 팀을 조정하기 위해 Windows MNGs는 Linux MNGs와 동일한 워크플로 및 도구를 사용하여 활성화됩니다. Windows Server 2019 및 2022의 전체 및 코어 AMI(Amazon Machine Image) 패밀리 버전이 지원됩니다.

관리형 노드 그룹(MNG)에는 다음 AMI 패밀리가 지원됩니다.

AMI 패밀리
WINDOWS_CORE_2019_x86_64
WINDOWS_FULL_2019_x86_64
WINDOWS_CORE_2022_x86_64
WINDOWS_FULL_2022_x86_64

추가 설명서

AWS 공식 설명서: https://docs.aws.amazon.com/eks/latest/userguide/windows-support.html

포드 네트워킹(CNI)의 작동 방식을 더 잘 이해하려면 https://docs.aws.amazon.com/eks/latest/userguide/pod-networking.html 링크를 확인하세요.

EKS에서 Windows용 관리형 노드 그룹 배포에 대한 AWS 블로그: https://aws.amazon.com/blogs/containers/deploying-amazon-eks-windows-managed-node-groups/

포드 보안 컨텍스트

포드 보안 정책(PSP)과 포드 보안 표준(PSS)은 Kubernetes에서 보안을 적용하는 두 가지 주요 방법입니다. PodSecurityPolicy는 Kubernetes v1.21부터 더 이상 사용되지 않으며 v1.25에서 제거되며 포드 보안 표준(PSS)은 향후 보안을 적용하기 위한 Kubernetes 권장 접근 방식입니다.

포드 보안 정책(PSP)은 보안 정책을 구현하기 위한 Kubernetes의 기본 솔루션입니다. PSP는 포드 사양의 보안에 민감한 측면을 제어하는 클러스터 수준 리소스입니다. 포드 보안 정책을 사용하여 클러스터에서 포드를 수락하기 위해 충족해야 하는 조건 세트를 정의할 수 있습니다. PSP 기능은 Kubernetes 초기부터 사용할 수 있으며 잘못 구성된 포드가 지정된 클러스터에서 생성되는 것을 차단하도록 설계되었습니다.

포드 보안 정책에 대한 자세한 내용은 Kubernetes 설명서를 참조하세요. Kubernetes 사용 중단 정책에 따라 이전 버전은 기능 사용 중단 후 9개월이 지나면 지원을 받지 못합니다.

반면 권장되는 보안 접근 방식이며 일반적으로 보안 컨텍스트를 사용하여 구현되는 포드 보안 표준(PSS)은 포드 매니페스트의 포드 및 컨테이너 사양의 일부로 정의됩니다. PSS는 Kubernetes 프로젝트 팀이 포드에 대한 보안 관련 모범 사례를 해결하기 위해 정의한 공식 표준입니다. 기준(최소 제한, 기본값), 권한(비제한) 및 제한(가장 제한적)과 같은 정책을 정의합니다.

기준 프로파일부터 시작하는 것이 좋습니다. PSS 기준 프로파일은 최소한의 예외 목록이 필요한 보안과 잠재적 마찰 간에 견고한 균형을 제공하며 워크로드 보안을 위한 좋은 출발점 역할을 합니다. 현재 PSPs 사용하는 경우 PSS로 전환하는 것이 좋습니다. PSS 정책에 대한 자세한 내용은 Kubernetes 설명서에서 확인할 수 있습니다. 이러한 정책은 OPA 및 Kyverno의 도구를 포함한 여러 도구를 사용하여 적용할 수 있습니다. 예를 들어 Kyverno는 여기에 PSS 정책의 전체 컬렉션을 제공합니다.

보안 컨텍스트 설정을 사용하면 프로세스를 선택하고, 프로그램 프로파일을 사용하여 기능을 개별 프로그램으로 제한하고, 권한 에스컬레이션을 허용하고, 시스템 호출을 필터링할 수 있는 권한을 부여할 수 있습니다.

Kubernetes의 Windows 포드에는 보안 컨텍스트와 관련하여 표준 Linux 기반 워크로드와 몇 가지 제한 사항 및 차이점이 있습니다.

Windows는 시스템 네임스페이스 필터와 함께 컨테이너당 작업 객체를 사용하여 컨테이너의 모든 프로세스를 포함하고 호스트와의 논리적 격리를 제공합니다. 네임스페이스 필터링 없이 Windows 컨테이너를 실행할 수 있는 방법은 없습니다. 즉, 호스트의 컨텍스트에서 시스템 권한을 어설션할 수 없으므로 Windows에서 권한 있는 컨테이너를 사용할 수 없습니다.

다음은 유일하게 문서화된 Windows 보안 컨텍스트 옵션이고 나머지`windowsOptions`는 일반 보안 컨텍스트 옵션입니다.

Windows와 Linux에서 지원되는 보안 컨텍스트 속성 목록은 여기에서 공식 설명서를 참조하세요.

포드별 설정은 모든 컨테이너에 적용됩니다. 지정하지 않으면 PodSecurityContext의 옵션이 사용됩니다. SecurityContext와 PodSecurityContext 모두에 설정된 경우 SecurityContext에 지정된 값이 우선합니다.

예를 들어 Windows 옵션인 포드 및 컨테이너에 대한 runAsUserName 설정은 Linux별 runAsUser 설정과 대략적으로 동일하며 다음 매니페스트에서 포드별 보안 컨텍스트가 모든 컨테이너에 적용됩니다.

```
apiVersion: v1
kind: Pod
metadata:
  name: run-as-username-pod-demo
spec:
  securityContext:
    windowsOptions:
      runAsUserName: "ContainerUser"
  containers:
  - name: run-as-username-demo

  nodeSelector:
    kubernetes.io/os: windows
```

다음에서는 컨테이너 수준 보안 컨텍스트가 포드 수준 보안 컨텍스트를 재정의합니다.

```
apiVersion: v1
kind: Pod
metadata:
  name: run-as-username-container-demo
spec:
  securityContext:
```

```
    windowsOptions:
      runAsUserName: "ContainerUser"
  containers:
  - name: run-as-username-demo
    ..
    securityContext:
        windowsOptions:
            runAsUserName: "ContainerAdministrator"
  nodeSelector:
    kubernetes.io/os: windows
```

runAsUserName 필드에 허용되는 값의 예: ContainerAdministrator, ContainerUser, NT AUTHORITY\NETWORK SERVICE, NT AUTHORITY\LOCAL SERVICE

일반적으로 ContainerUser for Windows 포드를 사용하여 컨테이너를 실행하는 것이 좋습니다. 사용자는 컨테이너와 호스트 간에 공유되지 않지만 ContainerAdministrator는 컨테이너의에 대한 추가 권한을 가집니다. 참고로 알아야 할 사용자 이름 제한 사항이 있습니다.

ContainerAdministrator를 사용해야 하는 경우의 좋은 예는 PATH를 설정하는 것입니다. 다음과 같이 사용자 지시문을 사용하여 이를 수행할 수 있습니다.

```
USER ContainerAdministrator
RUN setx /M PATH "%PATH%;C:/your/path"
USER ContainerUser
```

또한 보안 암호는 노드 볼륨에 일반 텍스트로 작성됩니다(linux의 tmpfs/in-memory와 비교). 즉, 두 가지 작업을 수행해야 합니다.

- 파일 ACLs 사용하여 보안 암호 파일 위치 보호
- BitLocker를 사용한 볼륨 수준 암호화 사용

영구 스토리지 옵션

트리 내 볼륨 플러그인과 out-of-tree 볼륨 플러그인이란 무엇입니까?

컨테이너 스토리지 인터페이스(CSI)가 도입되기 전에는 모든 볼륨 플러그인이 트리 내에 있었습니다. 즉, 코어 Kubernetes 바이너리와 함께 빌드, 연결, 컴파일 및 제공되고 코어 Kubernetes API를 확장했습니다. 즉, Kubernetes(볼륨 플러그인)에 새 스토리지 시스템을 추가하려면 코어 Kubernetes 코드 리포지토리에 코드를 확인해야 했습니다.

Out-of-tree 볼륨 플러그인은 Kubernetes 코드 베이스와 독립적으로 개발되며 Kubernetes 클러스터에 확장 프로그램으로 배포(설치)됩니다. 이를 통해 공급업체는 Kubernetes 릴리스 주기와 별도로 대out-of-band 드라이버를 업데이트할 수 있습니다. 이는 Kubernetes가 공급업체에 k8s와 인터페이스하는 표준 방법을 제공하는 스토리지 인터페이스 또는 CSI를 생성했기 때문에 대부분 가능합니다.

Amazon Elastic Kubernetes Services(EKS) 스토리지 클래스 및 CSI 드라이버에 대한 자세한 내용은 https://docs.aws.amazon.com/eks/latest/userguide/storage.html 확인할 수 있습니다.

Windows용 인트리 볼륨 플러그인

Kubernetes 볼륨을 사용하면 데이터 지속성 요구 사항이 있는 애플리케이션을 Kubernetes에 배포할 수 있습니다. 영구 볼륨 관리는 볼륨 프로비저닝/프로비저닝 해제/크기 조정, Kubernetes 노드에 볼륨 연결/분리, 포드의 개별 컨테이너에 볼륨 탑재/분리로 구성됩니다. 특정 스토리지 백엔드 또는 프로토콜에 대해 이러한 볼륨 관리 작업을 구현하기 위한 코드는 Kubernetes 볼륨 플러그인(트리 내 볼륨 플러그인)의 형태로 제공됩니다. Amazon Elastic Kubernetes Services(EKS)에서는 Windows에서 다음 Kubernetes 볼륨 플러그인 클래스가 지원됩니다.

트리 내 볼륨 플러그인: awsElasticBlockStore

Windows 노드에서 트리 내 볼륨 플러그인을 사용하려면 NTFS를 fsType으로 사용하려면 추가 StorageClass를 생성해야 합니다. EKS에서 기본 StorageClass는 ext4를 기본 fsType으로 사용합니다.

StorageClass는 관리자가 제공하는 스토리지의 "클래스"를 설명할 수 있는 방법을 제공합니다. 클래스마다 quality-of-service 수준, 백업 정책 또는 클러스터 관리자가 결정한 임의 정책에 매핑될 수 있습니다. Kubernetes는 클래스가 무엇을 나타내는지에 대한 의견이 없습니다. 이 개념을 다른 스토리지 시스템에서 "프로파일"이라고도 합니다.

다음 명령을 실행하여 확인할 수 있습니다.

```
kubectl describe storageclass gp2
```

출력:

```
Name:            gp2
IsDefaultClass:  Yes
Annotations:     kubectl.kubernetes.io/last-applied-
configuration={"apiVersion":"storage.k8s.io/v1","kind":"StorageClas
","metadata":{"annotations":{"storageclass.kubernetes.io/is-default-
class":"true"},"name":"gp2"},"parameters":{"fsType"
```

```
"ext4","type":"gp2"},"provisioner":"kubernetes.io/aws-
ebs","volumeBindingMode":"WaitForFirstConsumer"}
,storageclass.kubernetes.io/is-default-class=true
Provisioner:           kubernetes.io/aws-ebs
Parameters:            fsType=ext4,type=gp2
AllowVolumeExpansion:  <unset>
MountOptions:          <none>
ReclaimPolicy:         Delete
VolumeBindingMode:     WaitForFirstConsumer
Events:                <none>
```

NTFS를 지원하는 새 StorageClass를 생성하려면 다음 매니페스트를 사용합니다.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: gp2-windows
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  fsType: ntfs
volumeBindingMode: WaitForFirstConsumer
```

다음 명령을 실행하여 StorageClass를 생성합니다.

```
kubectl apply -f NTFSStorageClass.yaml
```

다음 단계는 영구 볼륨 클레임(PVC)을 생성하는 것입니다.

PV(PersistentVolume)는 관리자가 프로비저닝하거나 PVC를 사용하여 동적으로 프로비저닝한 클러스터의 스토리지 조각입니다. 노드가 클러스터 리소스인 것처럼 클러스터의 리소스입니다. 이 API 객체는 NFS, iSCSI 또는 cloud-provider-specific 스토리지 시스템 등 스토리지 구현에 대한 세부 정보를 캡처합니다.

PersistentVolumeClaim(PVC)은 사용자의 스토리지 요청입니다. 클레임은 특정 크기 및 액세스 모드를 요청할 수 있습니다(예: ReadWriteOnce, ReadOnlyMany 또는 ReadWriteMany를 탑재할 수 있음).

사용자는 다양한 사용 사례에 대해 성능과 같은 다양한 속성을 가진 PersistentVolumes가 필요합니다. 클러스터 관리자는 사용자가 볼륨 구현 방법에 대한 세부 정보에 노출되지 않고 크기 및 액세스 모드보다 더 다양한 방식으로 다양한 PersistentVolumes을 제공할 수 있어야 합니다. 이러한 요구 사항에는 StorageClass 리소스가 있습니다.

아래 예제에서 PVC는 네임스페이스 기간 내에 생성되었습니다.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ebs-windows-pv-claim
  namespace: windows
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: gp2-windows
  resources:
    requests:
      storage: 1Gi
```

다음 명령을 실행하여 PVC를 생성합니다.

```
kubectl apply -f persistent-volume-claim.yaml
```

다음 매니페스트는 Windows 포드를 생성하고 VolumeMount를 로 설정하고 PVC를에 연결된 스토리지로 `C:\Data` 사용합니다`C:\Data`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: windows-server-ltsc2019
  namespace: windows
spec:
  selector:
    matchLabels:
      app: windows-server-ltsc2019
      tier: backend
      track: stable
  replicas: 1
  template:
    metadata:
      labels:
        app: windows-server-ltsc2019
        tier: backend
        track: stable
    spec:
      containers:
```

```
    - name: windows-server-ltsc2019
      image: mcr.microsoft.com/windows/servercore:ltsc2019
      ports:
      - name: http
        containerPort: 80
      imagePullPolicy: IfNotPresent
      volumeMounts:
      - mountPath: "C:\\data"
        name: test-volume
    volumes:
      - name: test-volume
        persistentVolumeClaim:
          claimName: ebs-windows-pv-claim
    nodeSelector:
      kubernetes.io/os: windows
      node.kubernetes.io/windows-build: '10.0.17763'
```

PowerShell을 통해 Windows 포드에 액세스하여 결과를 테스트합니다.

```
kubectl exec -it podname powershell -n windows
```

Windows 포드 내에서 다음을 실행합니다. `ls`

출력:

```
PS C:\> ls


    Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          3/8/2021   1:54 PM                data
d-----          3/8/2021   3:37 PM                inetpub
d-r---          1/9/2021   7:26 AM                Program Files
d-----          1/9/2021   7:18 AM                Program Files (x86)
d-r---          1/9/2021   7:28 AM                Users
d-----          3/8/2021   3:36 PM                var
d-----          3/8/2021   3:36 PM                Windows
-a----         12/7/2019   4:20 AM           5510 License.txt
```

데이터 디렉터리는 EBS 볼륨에서 제공됩니다.

Windows용 Out-of-tree

CSI 플러그인과 연결된 코드는 일반적으로 컨테이너 이미지로 배포되고 DaemonSets 및 StatefulSets와 같은 표준 Kubernetes 구문을 사용하여 배포되는 out-of-tree 스크립트 및 바이너리로 제공됩니다. CSI 플러그인은 Kubernetes에서 다양한 볼륨 관리 작업을 처리합니다. CSI 플러그인은 일반적으로 노드 플러그인(각 노드에서 DaemonSet로 실행됨)과 컨트롤러 플러그인으로 구성됩니다.

CSI 노드 플러그인(특히 블록 디바이스 또는 공유 파일 시스템을 통해 노출된 영구 볼륨과 관련된 플러그인)은 디스크 디바이스 스캔, 파일 시스템 탑재 등과 같은 다양한 권한 있는 작업을 수행해야 합니다. 이러한 작업은 호스트 운영 체제마다 다릅니다. Linux 작업자 노드의 경우 컨테이너화된 CSI 노드 플러그인은 일반적으로 권한 있는 컨테이너로 배포됩니다. Windows 작업자 노드의 경우 컨테이너화된 CSI 노드 플러그인에 대한 권한 있는 작업은 각 Windows 노드에 사전 설치해야 하는 커뮤니티 관리형 독립형 바이너리인 csi-proxy를 사용하여 지원됩니다.

Amazon EKS 최적화 Windows AMI에는 2022년 4월부터 CSI 프록시가 포함되어 있습니다. 고객은 Windows 노드의 SMB CSI 드라이버를 사용하여 Amazon FSx for Windows File Server, Amazon FSx for NetApp ONTAP SMB 공유 및/또는 AWS Storage Gateway - File Gateway에 액세스할 수 있습니다.

다음 블로그에는 Amazon FSx for Windows File Server를 Windows 포드의 영구 스토리지로 사용하도록 SMB CSI 드라이버를 설정하는 방법에 대한 구현 세부 정보가 나와 있습니다.

Amazon FSx for Windows File Server

옵션은 SMB Global Mapping이라는 SMB 기능을 통해 Amazon FSx for Windows File Server를 사용하는 것입니다. 이를 통해 호스트에 SMB 공유를 탑재한 다음 해당 공유의 디렉터리를 컨테이너에 전달할 수 있습니다. 컨테이너는 특정 서버, 공유, 사용자 이름 또는 암호로 구성할 필요가 없습니다.이 모든 것은 호스트에서 대신 처리됩니다. 컨테이너는 로컬 스토리지가 있는 것처럼 작동합니다.

> SMB 글로벌 매핑은 오케스트레이터에 투명하며, 보안 문제를 암시할 수 있는 HostPath를 통해 탑재됩니다.

아래 예제에서 경로`G:\Directory\app-state`는 Windows 노드의 SMB 공유입니다.

```
apiVersion: v1
kind: Pod
metadata:
  name: test-fsx
spec:
```

```
  containers:
  - name: test-fsx
    image: mcr.microsoft.com/windows/servercore:ltsc2019
    command:
      - powershell.exe
      - -command
      - "Add-WindowsFeature Web-Server; Invoke-WebRequest -UseBasicParsing
 -Uri 'https://dotnetbinaries.blob.core.windows.net/servicemonitor/2.0.1.6/
ServiceMonitor.exe' -OutFile 'C:\\ServiceMonitor.exe'; echo '<html><body><br/
><br/><marquee><H1>Hello EKS!!!<H1><marquee></body><html>' > C:\\inetpub\\wwwroot\
\default.html; C:\\ServiceMonitor.exe 'w3svc'; "
    volumeMounts:
      - mountPath: C:\dotnetapp\app-state
        name: test-mount
  volumes:
    - name: test-mount
      hostPath:
        path: G:\Directory\app-state
        type: Directory
  nodeSelector:
      beta.kubernetes.io/os: windows
      beta.kubernetes.io/arch: amd64
```

다음 블로그에는 Amazon FSx for Windows File Server를 Windows 포드의 영구 스토리지로 설정하는 방법에 대한 구현 세부 정보가 나와 있습니다.

Windows 컨테이너 이미지 강화

Windows 컨테이너 이미지를 강화하시나요? 수년 동안 전 세계 고객과 협력하여 레거시 워크로드를 컨테이너, 특히 Windows 워크로드로 마이그레이션하는 데 도움을 주고 있습니다. 20년 이상의 경험을 바탕으로 조직이 Windows Server를 강화하는 데 많은 노력과 리소스를 할애하여 CIS 벤치마크에서 런타임 바이러스 백신 보호에 이르기까지 민감한 데이터를 보호하는 모든 것을 구현하는 것을 보았습니다.

그러나 우려되는 추세가 나타났습니다. 이러한 매우 안전한 가상 머신이 컨테이너로 현대화됨에 따라 많은 중요한 강화 관행이 간과되고 있습니다. 기본 이미지(OS)부터 IIS와 같은 웹 서비스에 이르기까지 Windows 보안 모범 사례는 무시되는 경우가 많으며 대부분 컨테이너 호스트 보안에만 중점을 둡니다. 컨테이너는 격리된 네임스페이스에서 작동하지만 여전히 커널 프리미티브를 호스트와 공유한다는 점을 인식하는 것이 중요합니다. 공격자는 일반적으로 컨테이너 호스트를 직접 타겟팅하는 것보다 측면 이동에 더 관심이 있으므로 취약한 컨테이너 보안 설정을 악용하고 민감한 데이터에 액세스할 수 있습니다.

설명서의 목표는 IIS에서 ASP.NET 웹 사이트를 호스팅하는 Windows 컨테이너에 특별히 구현해야 하는 몇 가지 필수 보안 설정을 강조하는 것입니다. 다음 네 가지 주요 영역에 초점을 맞출 것입니다.

- 계정 보안 정책
- 감사 정책
- IIS 보안 모범 사례
- 최소 권한의 원칙

먼저 이러한 각 보안 구성이 Windows 컨테이너를 보호하는 데 중요한 이유를 살펴보고, Windows 컨테이너가 완화하는 특정 위험과 컨테이너가 제공하는 보안 이점을 살펴보겠습니다. 다음으로 Dockerfile에서 이러한 구성을 올바르게 구현하여 잠재적 위협에 대비하여 컨테이너를 강화하는 방법을 보여주는 코드 조각을 살펴보겠습니다. 마지막으로 각 설정을 자세히 분석하여 함수, 컨테이너 보안에 미치는 영향, 애플리케이션 보호에 기여하는 방식에 대한 포괄적인 설명을 제공합니다. 이 접근 방식은 이러한 모범 사례를 적용하는 방법을 보여줄 뿐만 아니라 컨테이너화된 환경에서 강력한 보안 태세를 유지하는 데 필수적인 이유를 이해할 수 있는 통찰력을 제공합니다.

1. 로컬 보안 정책 및 레지스트리를 사용하여 계정 정책(암호 또는 잠금) 구성

Windows Server Core는 [EKS Optimized Windows AMI](https://docs.aws.amazon.com/eks/latest/userguide/eks-optimized-windows-ami.html)의 일부로 사용할 수 있는 최소 설치 옵션입니다. 로컬 보안 정책 및 레지스트리를 사용하여 계정 정책(암호 또는 잠금)을 구성하면 강력한 암호 및 잠금 규칙을 적용하여 시스템 보안이 강화됩니다. 이러한 정책은 사용자가 정의된 최소 길이와 복잡성으로 강력한 암호를 생성하여 일반적인 암호 관련 공격으로부터 보호해야 합니다.

최대 암호 수명을 설정하면 사용자에게 암호를 정기적으로 업데이트하라는 메시지가 표시되므로 자격 증명이 손상될 가능성이 줄어듭니다. 잠금 정책은 지정된 수의 로그인 시도가 실패한 후 계정을 일시적으로 잠가 보호 계층을 추가하여 무차별 대입 공격을 방지하는 데 도움이 됩니다. Windows 레지스트리를 통해 이러한 설정을 구성하면 관리자가 시스템 수준에서 이러한 보안 조치를 적용하여 조직 전체에서 균일성과 규정 준수를 보장할 수 있습니다. 이러한 계정 정책을 Windows 컨테이너에 적용하는 것은 컨테이너가 종종 임시적이고 격리된 워크로드를 위한 것이더라도 보안 일관성을 유지하는 데 필수적입니다.

보안 일관성

- 규정 준수: 컨테이너에 일관된 암호 정책 및 잠금 규칙을 적용하면 특히 엄격한 액세스 제어(예: HIPAA, PCI-DSS와 같은 규정 준수)가 필요한 환경에서 보안 규정 준수를 유지하는 데 도움이 됩니다.

- 강화된 컨테이너: 이러한 설정을 적용하면 무단 액세스 또는 암호 기반 공격에 대해 Windows 컨테이너가 강화되어 컨테이너의 보안 태세가 더 광범위한 시스템 보안 정책에 맞게 조정됩니다.

무차별 대입 공격으로부터 보호

- 계정 잠금: 이러한 설정은 특정 수의 로그인 시도 실패 후 계정을 잠가 무차별 대입 로그인 시도를 방지하는 데 도움이 됩니다. 이렇게 하면 공격자가 무제한의 암호를 시도할 수 없습니다.
- 암호 복잡성: 길이가 충분한 복잡한 암호를 요구하면 컨테이너화된 격리된 환경에서도 취약한 암호가 악용될 가능성이 줄어듭니다.

다중 사용자 시나리오

- 컨테이너화된 애플리케이션이 여러 사용자를 처리하도록 설계되었거나 사용자 인증이 필요한 경우 암호 정책을 적용하면 컨테이너 내의 사용자 계정이 엄격한 보안 규칙을 준수하도록 하여 권한이 부여된 사용자로만 액세스를 제한할 수 있습니다.

영구 Windows 컨테이너

- 컨테이너는 일반적으로 임시로 간주되지만 특정 Windows 컨테이너는 장기 서비스를 실행하거나 사용자 관리를 처리할 수 있으므로 일반 Windows 서버와 유사한 적절한 보안 정책을 적용하는 것이 중요합니다.

하이브리드 환경의 일관성

- 인프라에서 가상 머신과 컨테이너를 모두 실행하는 경우 모든 환경에 동일한 보안 정책(예: 암호/잠금 정책)을 적용하면 균일한 보안 표준을 보장하여 거버넌스 및 관리를 간소화할 수 있습니다.

요약하면 Windows 컨테이너 내에 이러한 계정 정책을 적용하면 컨테이너가 보안 전략의 약점이 되지 않아 암호 공격으로부터 보호하고 전체 환경에서 일관성을 유지할 수 있습니다.

Dockerfile:

```
# Configure account policies for password complexity and lockout
RUN powershell -Command \
      "Write-Output 'Configuring Account Policies (Password/Lockout)...'; \
      NET ACCOUNTS /MINPWLEN:14 /MAXPWAGE:60 /MINPWAGE:14 /LOCKOUTTHRESHOLD:5
```

설명:

이 섹션에서는 Windows 레지스트리를 통해 암호 및 잠금 설정에 대한 계정 정책을 구성합니다. 이러한 정책은 암호 요구 사항 및 계정 잠금 임계값을 제어하여 보안을 강화하는 데 도움이 됩니다.

1. MinimumPasswordLength(MINPWLEN) = 14이 설정은 암호의 최소 문자 수를 정의합니다. 범위는 0~14자이고 기본값은 6자입니다.
2. MaximumPasswordAge(MAXPWAGE) = 60이 설정은 암호가 유효한 최대 일수를 정의합니다. UNLIMITED를 사용하면 제한이 지정되지 않습니다. /MAXPWAGE는 /MINPWAGE보다 작을 수 없습니다. 범위는 1~999이고 기본값은 90일입니다.
3. 잠금 임계값(LOCKOUTTHRESHOLD) = 5이 설정은 실패한 로그인 시도에 대한 임계값을 정의합니다. 5번 잘못 시도하면 계정이 잠깁니다.

이러한 설정은 특정 횟수의 로그인 시도 실패 후 강력한 암호 정책을 적용하고 계정을 잠가 암호 보안을 개선하고 무차별 대입 공격을 방지하는 데 도움이 됩니다.

2. 감사 정책

감사 정책은 로그인 시도 및 권한 사용과 같은 보안 이벤트에 대한 중요한 가시성을 제공하여 무단 액세스를 감지하고, 사용자 활동을 모니터링하고, 규제 표준을 준수하는 데 도움이 되므로 Windows 컨테이너에 중요합니다. 컨테이너의 일시적인 특성에서도 감사 로그는 인시던트 조사, 선제적 위협 탐지 및 컨테이너화된 환경 전반에서 일관된 보안 태세 유지에 필수적입니다.

보안 모니터링 및 규정 준수:

- 사용자 활동 추적: 감사 정책을 통해 관리자는 컨테이너 내에서 로그인 시도 및 권한 사용과 같은 사용자 활동을 모니터링할 수 있습니다. 이는 무단 액세스 또는 의심스러운 동작을 감지하는 데 매우 중요합니다.
- 규정 준수: 많은 조직이 HIPAA, PCI-DSS 및 GDPR과 같은 규정을 준수하기 위해 보안 이벤트를 로깅해야 합니다. 컨테이너에서 감사 정책을 활성화하면 컨테이너화된 환경에서도 이러한 요구 사항을 충족할 수 있습니다.

인시던트 조사:

- 포렌식 및 분석: 컨테이너화된 애플리케이션 또는 서비스가 손상된 경우 감사 로그는 인시던트 후 분석에 유용한 인사이트를 제공할 수 있습니다. 보안 팀이 공격자가 취한 조치를 추적하거나 침해가 발생한 방식을 식별하는 데 도움이 됩니다.

- 실시간 감지: 감사 로그를 통해 관리자는 중요한 이벤트(예: 로그인 시도 실패, 권한 에스컬레이션)에 대한 실시간 알림을 설정할 수 있습니다. 이 사전 예방적 모니터링은 공격을 조기에 탐지하고 대응 시간을 단축하는 데 도움이 됩니다.

환경 간 일관성:

- 균일한 보안 태세: 레지스트리를 통해 컨테이너에 감사 정책을 적용하면 컨테이너화된 환경과 컨테이너화되지 않은 환경 모두에서 일관된 보안 관행을 보장할 수 있습니다. 이렇게 하면 컨테이너가 보안 모니터링의 사각지대가 되지 않습니다.
- 하이브리드 환경의 가시성: 기존 Windows 서버와 컨테이너를 모두 실행하는 조직의 경우 감사 정책은 모든 플랫폼에서 유사한 가시성과 제어를 제공하여 관리를 더 쉽고 효과적으로 만듭니다.

권한 있는 작업 추적:

- 권한 감사 사용: 애플리케이션이 승격된 권한으로 실행되거나 관리 작업이 수행되는 컨테이너 환경에서 권한 있는 작업을 감사하면 책임이 보장됩니다. 컨테이너 내에서 누가 민감한 리소스에 액세스했거나 중요한 작업을 수행했는지 로깅할 수 있습니다.
- 권한 남용 방지: 권한 사용을 모니터링하여 권한이 없는 사용자가 권한을 높이거나 컨테이너 내의 제한된 영역에 액세스하려고 할 때 이를 감지하여 내부 또는 외부 공격을 방지할 수 있습니다.

무단 액세스 시도 감지:

- 실패한 로그인 시도: 실패한 로그인 시도에 대한 감사 정책을 활성화하면 무차별 대입 공격 또는 컨테이너화된 애플리케이션에 대한 무단 액세스를 식별하는 데 도움이 됩니다. 이를 통해 시스템에 액세스하려는 사용자와 빈도를 파악할 수 있습니다.
- 계정 잠금 모니터링: 계정 잠금 이벤트를 감사하면 관리자가 의심스럽거나 악의적인 활동으로 인한 잠재적 잠금을 감지하고 조사할 수 있습니다.

임시 환경에서도 지속적인 보안:

- 임시 보안: 컨테이너는 임시적이므로 자주 삭제하고 다시 생성할 수 있지만 감사는 컨테이너가 실행되는 동안 보안 이벤트가 캡처되도록 하는 데 여전히 중요한 역할을 합니다. 이렇게 하면 컨테이너의 수명 주기 동안 중요한 보안 이벤트가 로깅됩니다.

중앙 집중식 로깅:

- 중앙 집중식 시스템으로 로그 전달: 컨테이너를 중앙 집중식 로깅 시스템(예: ELK 스택, AWS CloudWatch)과 통합하여 여러 컨테이너 인스턴스의 감사 로그를 캡처할 수 있습니다. 이를 통해 인프라 전반의 보안 이벤트를 더 잘 분석하고 상관관계를 파악할 수 있습니다.

Dockerfile:

```
# Configure audit policies for logging security events
RUN powershell -Command \
    "Write-Host 'Configuring Audit Policy..'; \
    Set-ItemProperty -Path 'HKLM:\\SYSTEM\\CurrentControlSet\\Control\\Lsa' -Name
 'SCENoApplyLegacyAuditPolicy' -Value 0; \
    auditpol /set /category:"Logon/Logoff" /subcategory:"Logon" /failure:enable

# Creates STDOUT on Windows Containers (check GitHub LogMonitor:: https://github.com/
microsoft/windows-container-tools/blob/main/LogMonitor/README.md)
COPY LogMonitor.exe LogMonitorConfig.json 'C:\\LogMonitor\\'
WORKDIR /LogMonitor
```

설명:

이 섹션에서는 레지스트리 수정을 사용하여 감사 정책을 구성합니다. 감사 정책은 Windows에서 로깅하는 보안 이벤트를 제어하므로 무단 액세스 시도를 모니터링하고 감지하는 데 도움이 됩니다.

1. SCENoApplyLegacyAuditPolicy = 0 레거시 감사 정책 형식을 비활성화하여 이후 버전의 Windows에 더 세분화된 감사 정책을 도입할 수 있습니다. 이는 최신 감사 구성에 중요합니다.
2. Auditpol 하위 범주: "Logon" 이 설정을 사용하면 성공 및 실패 로그온 이벤트 모두에 대한 감사를 수행할 수 있습니다. 값 3은 Windows가 성공한 로그인 시도와 실패한 로그인 시도를 모두 로깅함을 의미합니다. 이렇게 하면 시스템에 액세스하는 사용자를 모니터링하고 실패한 로그인 시도를 포착하는 데 도움이 됩니다.

이러한 감사 정책은 로그인 시도 및 권한 있는 작업 사용과 같은 중요한 보안 이벤트에 대한 자세한 로그를 제공하므로 보안 모니터링 및 규정 준수에 매우 중요합니다.

3. Windows 컨테이너에 대한 IIS 보안 모범 사례

Windows Containers에서 IIS 모범 사례를 구현하는 것은 애플리케이션이 안전하고 고성능이며 확장 가능하도록 하는 여러 가지 이유로 중요합니다. 컨테이너는 격리와 경량 환경을 제공하지만 취약성과 운영 문제를 방지하기 위해 적절한 구성이 필요합니다. Windows 컨테이너의 IIS에 대한 다음 모범 사례가 중요한 이유는 다음과 같습니다.

보안

- 일반적인 취약성 방지: IIS는 교차 사이트 스크립팅(XSS), 클릭재킹 및 정보 공개와 같은 공격의 대상인 경우가 많습니다. 보안 헤더(예: X-Content-Type-Options, X-Frame-Options, Strict-Transport-Security)를 구현하면 이러한 위협으로부터 애플리케이션을 보호하는 데 도움이 됩니다.
- 격리만으로는 충분하지 않음: 컨테이너는 격리되어 있지만 잘못 구성된 IIS 인스턴스는 서버 버전 세부 정보, 디렉터리 목록 또는 암호화되지 않은 통신과 같은 민감한 정보를 노출할 수 있습니다. 디렉터리 브라우징 및 IIS 버전 헤더 제거와 같은 기능을 비활성화하면 공격 표면을 최소화할 수 있습니다.
- 암호화 및 HTTPS: HTTPS 전용 연결 적용과 같은 모범 사례는 전송 중인 데이터가 암호화되어 민감한 정보가 가로채지 않도록 보호합니다.

성능

- 효율적인 리소스 사용: 동적 및 정적 압축 활성화와 같은 IIS 모범 사례는 대역폭 사용량을 줄이고 로드 시간을 개선합니다. 이러한 최적화는 리소스가 컨테이너와 호스트 시스템 간에 공유되는 컨테이너화된 환경에서 특히 중요합니다.
- 최적화된 로깅: 로깅(예: X-Forwarded-For 헤더 포함)을 올바르게 구성하면 불필요한 로깅 오버헤드를 최소화하면서 클라이언트 활동을 추적할 수 있습니다. 이렇게 하면 성능 저하 없이 문제 해결을 위한 관련 데이터를 수집할 수 있습니다.

확장성 및 유지 관리성

- 환경 간 일관성: 모범 사례를 따르면 IIS 구성이 여러 컨테이너 인스턴스에서 일관되게 유지됩니다. 이렇게 하면 크기 조정이 간소화되고 새 컨테이너가 배포될 때 동일한 보안 및 성능 지침을 준수할 수 있습니다.
- 자동 구성: 폴더 권한 설정 및 불필요한 기능 비활성화와 같은 Dockerfiles의 모범 사례는 각 새 컨테이너가 자동으로 올바르게 구성되도록 합니다. 이렇게 하면 수동 개입이 줄어들고 인적 오류의 위험이 줄어듭니다.

규정 준수

- 규제 요구 사항 충족: 많은 산업에는 암호화된 통신(HTTPS) 및 클라이언트 요청 로깅과 같은 특정 보안 조치를 요구하는 엄격한 규제 요구 사항(예: PCI-DSS, HIPAA)이 있습니다. 컨테이너의 IIS 모범 사례를 따르면 이러한 표준을 준수하는 데 도움이 됩니다.

- 감사 가능성: 감사 정책 및 보안 로깅을 구현하면 감사에 중요한 이벤트를 추적할 수 있습니다. 예를 들어 X-Forwarded-For 헤더를 로깅하면 프록시 기반 아키텍처에 클라이언트 IP 주소가 올바르게 기록됩니다.

공유 환경의 위험 최소화

- 잘못된 구성 방지: 컨테이너는 호스트의 커널을 공유하며, 서로 격리되어 있는 동안 잘못 구성된 IIS 인스턴스는 취약성을 노출하거나 성능 병목 현상을 일으킬 수 있습니다. 모범 사례는 각 IIS 인스턴스가 최적으로 실행되도록 하여 컨테이너 간 문제의 위험을 줄입니다.
- 최소 권한 액세스: 컨테이너 내의 폴더 및 파일에 대한 적절한 권한을 설정하면(예: PowerShell의 Set-Acl 사용) 컨테이너 내의 사용자 및 프로세스가 필요한 액세스 권한만 갖게 되므로 권한 에스컬레이션 또는 데이터 변조의 위험이 줄어듭니다.

임시 환경의 복원력

- 컨테이너의 임시 특성: 컨테이너는 수명이 짧고 자주 재구축되는 경우가 많습니다. IIS 모범 사례를 적용하면 재배포 횟수에 관계없이 각 컨테이너가 안전하고 일관되게 구성됩니다. 이렇게 하면 시간이 지남에 따라 잘못된 구성이 도입되는 것을 방지할 수 있습니다.
- 잠재적 구성 오류 완화: 모범 사례(예: 취약한 프로토콜 또는 헤더 비활성화)를 자동으로 적용하면 컨테이너 재시작 또는 업데이트 중에 구성이 잘못될 위험이 최소화됩니다.

Dockerfile:

```
# Enforce HTTPS (disable HTTP) -- Only if container is target for SSL termination
RUN powershell -Command \
    "$httpBinding = Get-WebBinding -Name 'Default Web Site' -Protocol http | Where-
Object { $_.bindingInformation -eq '*:80:' }; \
    if ($httpBinding) { Remove-WebBinding -Name 'Default Web Site' -Protocol http -Port
 80; } \
    $httpsBinding = Get-WebBinding -Name 'Default Web Site' -Protocol https | Where-
Object { $_.bindingInformation -eq '*:443:' }; \
    if (-not $httpsBinding) { New-WebBinding -Name 'Default Web Site' -Protocol https -
Port 443 -IPAddress '*'; }"

# Use secure headers
RUN powershell -Command \
    "Write-Host 'Adding security headers...'; \
```

```
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 'system.applicationHost/sites/siteDefaults/logFile/customFields' -name
 "." -value @{logFieldName='X-Forwarded-For';sourceName='X-Forwarded-
For';sourceType='RequestHeader'}; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpProtocol/customHeaders" -name "." -value @{name='Strict-
Transport-Security';value='max-age=31536000; includeSubDomains'}; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpProtocol/customHeaders" -name "." -value @{name='X-Content-Type-
Options';value='nosniff'}; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpProtocol/customHeaders" -name "." -value @{name='X-XSS-
Protection';value='1; mode=block'}; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpProtocol/customHeaders" -name "." -value @{name='X-Frame-
Options';value='DENY'};"

# Disable IIS version disclosure
RUN powershell -Command \
    "Write-Host 'Disabling IIS version disclosure...'; \
    Import-Module WebAdministration; \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/security/requestFiltering" -name "removeServerHeader" -value
 "true";"

# Set IIS Logging Best Practices
RUN powershell -Command \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/directoryBrowse" -name "enabled" -value "false"; \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpErrors" -name "existingResponse" -value "PassThrough"; \

# Enable IIS dynamic and static compression to optimize performance
RUN powershell -Command \
    "Write-Host 'Enabling IIS compression...'; \
    Enable-WindowsOptionalFeature -Online -FeatureName IIS-HttpCompressionDynamic; \
    Import-Module WebAdministration; \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/urlCompression" -name "doDynamicCompression" -value "true"; \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/urlCompression" -name "doStaticCompression" -value "true"

# Ensure proper folder permissions using PowerShell's Set-Acl
```

```
RUN powershell -Command \
    "Write-Host 'Setting folder permissions for IIS...'; \
    $path = 'C:\\inetpub\\wwwroot'; \
    $acl = Get-Acl $path; \
    $iusr = New-Object System.Security.Principal.NTAccount('IIS_IUSRS'); \
    $rule = New-Object System.Security.AccessControl.FileSystemAccessRule($iusr,
 'ReadAndExecute', 'ContainerInherit, ObjectInherit', 'None', 'Allow'); \
    $acl.SetAccessRule($rule); \
    $users = New-Object System.Security.Principal.NTAccount('Users'); \
    $rule2 = New-Object System.Security.AccessControl.FileSystemAccessRule($users,
 'ReadAndExecute', 'ContainerInherit, ObjectInherit', 'None', 'Allow'); \
    $acl.SetAccessRule($rule2); \
    Set-Acl -Path $path -AclObject $acl"
```

설명:

이 명령은 요청이 프록시 또는 로드 밸런서를 통과할 때 원래 클라이언트 IP 주소를 캡처하는 데 일반적으로 사용되는 X-Forwarded-For 헤더를 로깅하도록 IIS를 구성합니다. 기본적으로 IIS는 로드 밸런서 또는 역방향 프록시의 IP 주소만 로깅하므로이 사용자 지정 로그 필드를 추가하면 보안 감사, 분석 및 문제 해결을 위한 실제 클라이언트 IP를 추적하는 데 도움이 됩니다.

1. 요청이 프록시 또는 로드 밸런서를 통과할 때 원래 클라이언트 IP 주소를 캡처하는 데 일반적으로 사용되는 X-Forwarded-For 헤더입니다. 기본적으로 IIS는 로드 밸런서 또는 역방향 프록시의 IP 주소만 로깅하므로이 사용자 지정 로그 필드를 추가하면 보안 감사, 분석 및 문제 해결을 위한 실제 클라이언트 IP를 추적하는 데 도움이 됩니다.
2. Strict-Transport-Security(HSTS) 브라우저가 HTTPS를 통해서만 통신하도록 합니다. max-age=31536000은이 정책이 1년 동안 적용되도록 지정하고 includeSubDomains는 모든 하위 도메인에 정책을 적용합니다.
3. X-Content-Type-Options 브라우저가 선언된 콘텐츠 유형에서 다른 응답을 "MIME-sniffing"하지 못하도록 합니다. 이렇게 하면 일부 유형의 공격을 방지하는 데 도움이 됩니다.
4. X-XSS-Protection 브라우저에서 교차 사이트 스크립팅(XSS) 보호를 활성화합니다.
5. X-Frame-Options 페이지가 iframe에 임베딩되는 것을 방지하여 클릭재킹 공격으로부터 보호합니다.
6. IIS 버전 공개 비활성화이 명령은 HTTP 응답에서 서버 헤더를 비활성화하며, 기본적으로 사용 중인 IIS 버전을 표시합니다. 이 정보를 숨기면 공격자가 IIS 버전과 관련된 취약성을 식별하고 대상으로 지정할 위험을 줄이는 데 도움이 됩니다.
7. HTTPS 전용 연결 활성화이 (코멘트 출력) 섹션에서는 HTTPS 연결을 적용하고 HTTP를 비활성화합니다. 주석 처리되지 않은 경우 Dockerfile은 포트 443(HTTPS)에서만 수신하도록 IIS를 구성하고

포트 80에서 기본 HTTP 바인딩을 제거합니다. 이는 컨테이너 내에서 SSL을 종료할 때 유용하며 모든 트래픽이 암호화되도록 합니다.

8. 디렉터리 찾아보기 비활성화 기본 문서가 없을 때 IIS가 디렉터리 목록을 표시하지 못하도록 합니다. 이렇게 하면 내부 파일 구조가 사용자에게 노출되지 않습니다.
9. 사용자 지정 오류 페이지 전달 애플리케이션에 자체 오류 처리가 있는 경우 IIS는 기본 IIS 오류 페이지를 표시하는 대신 애플리케이션의 오류 페이지를 통과하도록 합니다.
10. 세부 오류 모드 로컬 요청에 대해서만 자세한 오류 메시지를 표시하도록 IIS를 구성하므로 개발자가 외부 사용자에게 민감한 정보를 노출하지 않고 문제를 진단할 수 있습니다.
11. 적절한 폴더 권한 보장이 블록은 IIS 웹 루트(C:\inetpub\wwwroot)에 대한 폴더 권한을 구성합니다. IIS_IUSRS 및 사용자 그룹에 대한 읽기 및 실행 권한을 설정하여 이러한 사용자가 폴더에 액세스하지만 파일을 수정할 수 없도록 합니다. 올바른 권한을 설정하면 웹 서버에서 호스팅하는 파일에 대한 무단 액세스 또는 변조 위험이 최소화됩니다.

Windows Containers의 IIS 모범 사례를 따르면 컨테이너화된 애플리케이션의 보안, 고성능 및 확장성이 보장됩니다. 이러한 관행은 취약성을 방지하고, 리소스 사용을 최적화하고, 규정 준수를 보장하고, 컨테이너 인스턴스 간 일관성을 유지하는 데 도움이 됩니다. 컨테이너는 격리되도록 설계되었지만 위험을 최소화하고 프로덕션 환경에서 애플리케이션의 신뢰성을 보장하려면 적절한 구성이 필요합니다.

4. 최소 권한의 원칙

최소 권한 원칙(PoLP)은 특히 컨테이너화된 환경 내에서 보안을 강화하고 위험을 최소화하는 등 몇 가지 중요한 이유로 Windows 컨테이너에 매우 중요합니다. 이 원칙은 시스템 또는 애플리케이션이 제대로 작동하는 데 필요한 최소 수준의 권한으로 작동해야 한다고 규정합니다. Windows 컨테이너에서 중요한 이유는 다음과 같습니다.

공격 표면 최소화

- 컨테이너는 다양한 시스템 구성 요소와 상호 작용하는 애플리케이션을 실행하는 경우가 많으며, 애플리케이션이 더 많은 권한을 가질수록 해당 구성 요소에 대한 액세스 범위가 넓어집니다. 컨테이너의 권한을 필요한 것으로만 제한하면 PoLP는 공격 표면을 크게 줄여 공격자가 컨테이너가 손상된 경우 컨테이너를 악용하는 것을 더 어렵게 만듭니다.

손상된 컨테이너의 영향 제한

- Windows 컨테이너가 손상된 경우 과도한 권한(예: 관리자 또는 루트 수준 액세스)이 있는 애플리케이션을 실행하면 공격자가 중요한 시스템 파일을 제어하거나 컨테이너 호스트 전체에 권한을 에스컬레이션할 수 있습니다. PoLP를 적용하면 컨테이너가 침해되더라도 공격자가 수행할 수 있는 작업

이 제한되어 민감한 리소스 또는 기타 컨테이너에 대한 추가 에스컬레이션 및 액세스를 방지할 수 있습니다.

멀티테넌트 환경의 보호

- 클라우드 또는 엔터프라이즈 환경에서는 동일한 물리적 또는 가상 인프라에서 여러 컨테이너를 실행할 수 있습니다. PoLP는 손상된 컨테이너가 다른 테넌트에 속한 리소스 또는 데이터에 액세스할 수 없도록 합니다. 이 격리는 공유 멀티테넌트 환경에서 보안을 유지하여 컨테이너 간의 측면 이동을 방지하는 데 매우 중요합니다.

권한 에스컬레이션 완화

- 공격자는 높은 권한으로 실행되는 컨테이너를 사용하여 시스템 내에서 권한을 에스컬레이션할 수 있습니다. PoLP는 시스템 리소스에 대한 컨테이너의 액세스를 제한하여 이러한 위험을 완화함으로써 컨테이너 환경을 넘어 무단 작업 또는 권한 에스컬레이션을 방지합니다.

규정 준수 및 감사

- 많은 규제 표준 및 보안 프레임워크(예: PCI DSS, HIPAA, GDPR)에서는 시스템이 PoLP를 준수하여 민감한 데이터에 대한 액세스를 제한해야 합니다. 제한된 권한이 있는 Windows 컨테이너를 실행하면 조직이 이러한 규정을 준수하고 애플리케이션에 특별히 필요한 리소스에 대한 액세스 권한만 부여할 수 있습니다.

잘못된 구성의 위험 줄이기

- 컨테이너가 불필요한 권한으로 실행되는 경우 사소한 구성 오류로도 심각한 보안 취약성이 발생할 수 있습니다. 예를 들어 관리자로 실행되는 컨테이너가 실수로 인터넷에 노출된 경우 공격자가 시스템을 제어할 수 있습니다. PoLP는 기본적으로 제한된 권한으로 인해 이러한 위험을 방지하여 잘못된 구성을 덜 위험하게 만듭니다.

컨테이너 보안 태세 개선

- PoLP를 따르면 컨테이너가 기본 호스트 시스템과 서로 더 잘 격리됩니다. 이렇게 하면 컨테이너화된 애플리케이션이 정의된 범위를 벗어나 시스템 파일 또는 프로세스에 액세스하거나 수정할 가능성을 줄여 호스트 운영 체제 및 기타 워크로드의 무결성을 유지할 수 있습니다.

Dockerfile:

```
# Strongly recommended that when deploying a Windows server container to any multi-
tenant environment that your application runs via the ContainerUser account
USER ContainerUser
```

설명:

이 단원에서 USER ContainerUser 명령은 Windows 컨테이너 내의 애플리케이션이 기본 관리자 계정 대신 ContainerUser 계정에서 실행되도록 지정합니다.

특히 멀티테넌트 환경에서 이것이 중요한 이유는 다음과 같습니다.

1. 최소 권한 원칙: ContainerUser 계정은 권한이 제한된 관리자가 아닌 사용자입니다. 이 계정으로 애플리케이션을 실행하면 최소 권한 원칙을 준수하므로 악용 위험을 최소화하는 데 도움이 됩니다. 공격자가 애플리케이션을 손상시키려면 시스템에 대한 액세스가 제한되어 잠재적 손상이 줄어듭니다.
2. 향상된 보안: 멀티테넌트 환경에서 컨테이너는 동일한 기본 인프라를 공유할 수 있습니다. ContainerUser로 실행하면 한 컨테이너가 손상되더라도 중요한 시스템 파일 또는 기타 컨테이너에 액세스하거나 수정할 수 있는 관리 권한이 없습니다. 이렇게 하면 공격 표면이 크게 줄어듭니다.
3. 루트 액세스 방지: 기본적으로 컨테이너는 승격된 권한(Linux 컨테이너의 루트 액세스와 유사)으로 실행될 수 있으며, 이는 악용될 경우 위험할 수 있습니다. ContainerUser를 사용하면 애플리케이션이 불필요한 관리 권한으로 실행되지 않으므로 공격자가 권한을 에스컬레이션하는 것이 더 어려워집니다.
4. 멀티테넌트 환경 모범 사례: 여러 사용자 또는 조직이 동일한 인프라(예: 클라우드)를 공유하는 환경에서는 보안이 중요합니다. 제한된 권한으로 애플리케이션을 실행하면 한 테넌트의 애플리케이션이 다른 테넌트의 애플리케이션에 영향을 미치지 않으므로 플랫폼 전체에서 민감한 데이터와 리소스를 보호할 수 있습니다.

USER ContainerUser 명령은 애플리케이션이 최소한의 권한으로 실행되도록 하여 컨테이너가 손상된 경우 발생할 수 있는 손상을 제한하여 멀티테넌트 환경에서 보안을 강화합니다. 이는 컨테이너화된 환경에서 무단 액세스 또는 권한 에스컬레이션을 방지하는 모범 사례입니다.

최소 권한 원칙은 보안 침해의 잠재적 영향을 제한하고, 공격 표면을 줄이며, 중요한 시스템 구성 요소에 대한 무단 액세스를 방지하기 때문에 Windows 컨테이너에 필수적입니다. 필요한 권한으로만 컨테이너화된 애플리케이션을 실행하면 조직은 특히 멀티테넌트 및 공유 인프라에서 컨테이너 환경의 보안과 안정성을 크게 향상시킬 수 있습니다.

최종 생각: 오늘날의 위협 환경에서 Windows 컨테이너를 보호해야 하는 이유

위협이 점점 더 정교해지고 풍부해지고 있는 오늘날의 빠르게 진화하는 디지털 세계에서는 Windows 컨테이너를 보호하는 것이 단순한 권장 사항이 아니라 절대적으로 필요합니다. 컨테이너는 애플리케이션을 패키징하고 배포하는 가볍고 유연한 방법을 제공하지만 보안 취약성에 영향을 받지 않습니다. 더 많은 기업이 인프라를 간소화하기 위해 컨테이너를 채택함에 따라 제대로 보호되지 않으면 사이버 공격의 잠재적 대상이 됩니다.

인터넷에는 패치되지 않은 취약성을 대상으로 하는 악의적인 공격자부터 구성 오류에 대한 자동 봇 스캔에 이르기까지 다양한 위협이 넘쳐납니다. 올바른 보안 조치가 마련되어 있지 않으면 컨테이너를 악용하여 민감한 데이터를 노출하거나, 권한을 에스컬레이션하거나, 광범위한 인프라를 손상시킬 수 있는 공격의 진입점 역할을 할 수 있습니다. 따라서 컨테이너 보안이 환경의 다른 부분을 보호하는 것만큼 중요합니다.

Windows 컨테이너를 사용할 때는 여전히 많은 기존 보안 모범 사례가 적용됩니다. 강력한 계정 정책을 구현하고, IIS 구성을 보호하고, HTTPS를 적용하고, 엄격한 방화벽 규칙을 사용하고, 중요 파일에 최소 권한 액세스를 적용하는 것은 모두 컨테이너가 공격에 대한 복원력을 유지하도록 하는 주요 조치입니다. 또한 정기적인 감사 및 로깅을 통해 컨테이너 내부에서 일어나는 일을 파악할 수 있으므로 의심스러운 활동이 완전히 중단된 인시던트로 전환되기 전에 포착할 수 있습니다.

또한 Windows 컨테이너 보안은 민감한 데이터를 보호하고 애플리케이션 무결성을 보장해야 하는 규제 요구 사항에 부합합니다. 클라우드 네이티브 아키텍처와 컨테이너화된 아키텍처가 점점 더 보편화됨에 따라 기본 이미지부터 실행 중인 컨테이너에 이르기까지 모든 계층에서 보안을 보장하면 운영을 보호하고 고객 신뢰를 유지하는 데 도움이 됩니다.

요약하면, 컨테이너화된 애플리케이션의 증가는 점점 더 많은 사이버 위협과 함께 컨테이너 보안을 최신 인프라 관리의 타협할 수 없는 측면으로 만듭니다. 모범 사례를 준수하고 취약성을 지속적으로 모니터링하면 기업은 보안을 손상시키지 않고 Windows 컨테이너의 민첩성과 효율성을 누릴 수 있습니다. 위협이 많은이 환경에서는 Windows 컨테이너를 보호하는 것이 단순한 옵션이 아니라 필수 항목입니다.

하이브리드 배포 모범 사례

이 가이드에서는 EKS Hybrid Nodes 또는 EKS Anywhere를 사용하여 온프레미스 또는 엣지 환경에서 배포를 실행하는 방법에 대한 지침을 제공합니다.

현재 다음 주제에 대한 가이드를 게시했습니다.

- EKS 하이브리드 노드 및 네트워크 연결 해제 모범 사례

EKS 하이브리드 노드 및 네트워크 연결 해제

EKS Hybrid Nodes 아키텍처는 자체 데이터 센터 또는 엣지 로케이션에서 로컬 Kubernetes 클러스터를 완전히 실행하는 데 익숙한 고객에게 새로운 아키텍처가 될 수 있습니다. EKS Hybrid Nodes를 사용하면 Kubernetes 컨트롤 플레인이 AWS 리전에서 실행되고 노드만 온프레미스에서 실행되어 Kubernetes 클러스터 아키텍처가 “확장” 또는 “확장”됩니다.

이로 인해 “노드가 Kubernetes 컨트롤 플레인에서 연결 해제되면 어떻게 되나요?”라는 일반적인 질문이 발생합니다.

이 가이드에서는 다음 주제를 검토하여 해당 질문에 답합니다. 각 애플리케이션은 종속성, 구성 및 환경에 따라 다르게 작동할 수 있으므로 네트워크 연결 해제를 통해 애플리케이션의 안정성과 신뢰성을 검증하는 것이 좋습니다. EKS Hybrid Nodes 및 자체 애플리케이션을 사용하여 네트워크 연결 해제를 테스트하기 위해 참조할 수 있는 테스트 설정, 절차 및 결과는 aws-samples/eks-hybrid-examples GitHub 리포지토리를 참조하세요. GitHub 리포지토리에는이 가이드에 설명된 동작을 검증하는 데 사용되는 테스트에 대한 추가 세부 정보도 포함되어 있습니다.

- 네트워크 연결 해제를 통한 안정성 모범 사례
- 네트워크 연결 해제를 통한 Kubernetes 포드 장애 조치 동작
- 네트워크 연결 해제를 통한 애플리케이션 네트워크 트래픽
- 네트워크 연결 해제를 통해 자격 증명 호스팅

네트워크 연결 해제를 통한 안정성 모범 사례

고가용성 네트워킹

하이브리드 노드와 Kubernetes 컨트롤 플레인 간의 네트워크 연결 해제를 방지하는 가장 좋은 방법은 온프레미스 환경에서 AWS로 또는 AWS와 중복되고 복원력이 뛰어난 연결을 사용하는 것입니다. 이

러한 솔루션으로 고가용성 하이브리드 네트워크를 설계하는 방법에 대한 자세한 내용은 AWS Direct Connect Resiliency Toolkit 및 AWS Site-to-Site VPN 설명서를 참조하세요.

고가용성 애플리케이션

애플리케이션을 설계할 때는 장애 도메인과 다양한 유형의 중단의 영향을 고려하세요. Kubernetes는 노드, 영역 및 리전 도메인에 걸쳐 애플리케이션 복제본을 배포하고 유지 관리하는 기본 제공 메커니즘을 제공합니다. 이러한 메커니즘의 사용은 애플리케이션 아키텍처, 환경 및 가용성 요구 사항에 따라 달라집니다. 예를 들어, 상태 비저장 애플리케이션을 여러 복제본과 함께 배포할 수 있고 임의 호스트 및 인프라 용량으로 이동할 수 있으며 노드 선택기 및 토폴로지 분산 제약 조건을 사용하여 여러 도메인에서 애플리케이션의 인스턴스를 실행할 수 있습니다. Kubernetes에서 복원력이 뛰어난 애플리케이션을 빌드하기 위한 애플리케이션 수준 기법에 대한 자세한 내용은 EKS 모범 사례 가이드를 참조하세요.

Kubernetes는 포드를 다른 노드로 이동할지 여부를 결정할 때 Kubernetes 컨트롤 플레인과 연결이 해제된 노드에 대한 영역 정보를 평가합니다. 영역의 모든 노드에 연결할 수 없는 경우 Kubernetes는 해당 영역의 노드에 대한 포드 제거를 취소합니다. 여러 데이터 센터 또는 물리적 위치에서 노드가 실행되는 배포가 있는 경우 데이터 센터 또는 물리적 위치에 따라 각 노드에 영역을 할당하는 것이 가장 좋습니다. 클라우드의 노드로 EKS를 실행하면이 영역 레이블이 AWS cloud-controller-manager에 의해 자동으로 적용됩니다. 그러나 cloud-controller-manager는 하이브리드 노드에서 사용되지 않으므로 kubelet 구성을 통해이 정보를 전달할 수 있습니다. 하이브리드 노드에 대한 노드 구성에서 영역을 구성하는 방법의 예는 다음과 같습니다. 하이브리드 노드 CLI()를 사용하여 하이브리드 노드를 클러스터에 연결하면 구성이 전달됩니다`nodeadm`. `topology.kubernetes.io/zone` 레이블에 대한 자세한 내용은 Kubernetes 설명서를 참조하세요. 하이브리드 노드 CLI에 대한 자세한 내용은 Hybrid Nodes nodeadm 참조를 참조하세요.

```
apiVersion: node.eks.aws/v1alpha1
kind: NodeConfig
spec:
  cluster:
    name: my-cluster
    region: my-region
  kubelet:
    flags:
       - --node-labels=topology.kubernetes.io/zone=dc1
  hybrid:
    ...
```

네트워크 모니터링

하이브리드 연결에 AWS Direct Connect 또는 AWS Site-to-Site VPN을 사용하는 경우 CloudWatch 경보, 로그 및 지표를 활용하여 하이브리드 연결 상태를 관찰하고 문제를 진단할 수 있습니다. 자세한 내용은 AWS Direct Connect 리소스 모니터링 및 AWS Site-to-Site VPN 연결 모니터링을 참조하세요.

하이브리드 node-lifecycle-controller가 보고하는 `NodeNotReady` 이벤트에 대한 경보를 생성하는 것이 좋습니다. Controller Manager에 대해 EKS 컨트롤 플레인 로깅을 활성화하고 상태가 “NodeNotReady”인 “노드에 대한 상태 변경 이벤트 메시지 기록” 메시지에 대해 CloudWatch에서 지표 필터를 생성하여이 경보를 생성할 수 있습니다. 지표 필터를 생성한 후 원하는 임계값을 기반으로이 필터에 대한 경보를 생성할 수 있습니다. 자세한 내용은 CloudWatch 설명서의 로그에 대한 경보를 참조하세요.

Transit Gateway(TGW) 및 Virtual Private Gateway(VGW) 기본 제공 지표를 사용하여 TGW 또는 VGW로 들어오고 나가는 네트워크 트래픽을 관찰할 수 있습니다. 이러한 지표에 대한 경보를 생성하여 네트워크 트래픽이 정상 수준 아래로 떨어지는 시나리오를 감지하여 하이브리드 노드와 EKS 컨트롤 플레인 간의 잠재적인 네트워크 문제를 나타낼 수 있습니다. TGW 및 VGW 지표는 다음 표에 설명되어 있습니다.

Gateway	지표	설명
전송 게이트웨이	BytesIn	연결에서 TGW가 수신한 바이트(EKS 컨트롤 플레인에서 하이브리드 노드로)
전송 게이트웨이	BytesOut	TGW에서 연결로 전송된 바이트(하이브리드 노드에서 EKS 컨트롤 플레인으로)
가상 프라이빗 게이트웨이	TunnelDataIn	VPN 터널을 통해 연결의 AWS 측에서 고객 게이트웨이로 전송된 바이트(EKS 컨트롤 플레인에서 하이브리드 노드로)
가상 프라이빗 게이트웨이	TunnelDataOut	고객 게이트웨이에서 VPN 터널을 통해 연결의 AWS 측에서 수신된 바이트(하이브리드 노드에서 EKS 컨트롤 플레인으로)

또한 CloudWatch Network Monitor를 사용하여 하이브리드 연결에 대한 심층적인 인사이트를 얻어 평균 복구 시간을 줄이고 네트워크 문제가 AWS 또는 환경에서 발생하는지 확인할 수 있습니다. CloudWatch Network Monitor를 사용하여 하이브리드 네트워크 연결의 패킷 손실 및 지연 시간을 시각

화하고 알림 및 임계값을 설정한 다음 네트워크 성능을 개선하기 위한 조치를 취할 수 있습니다. 자세한 내용은 Amazon CloudWatch Network Monitor 사용을 참조하세요.

EKS는 클러스터 및 애플리케이션의 상태를 모니터링하기 위한 몇 가지 옵션을 제공합니다. 클러스터 상태의 경우 EKS 콘솔의 관찰성 대시보드를 사용하여 문제를 신속하게 감지, 해결 및 해결할 수 있습니다. 클러스터, 애플리케이션 및 인프라 모니터링에 Amazon Managed Service for Prometheus, AWS Distro for Open Telemetry(ADOT) 및 CloudWatch를 사용할 수도 있습니다. EKS 관찰성 옵션에 대한 자세한 내용은 클러스터 성능 모니터링 및 로그 보기를 참조하세요.

로컬 문제 해결

하이브리드 노드와 EKS 컨트롤 플레인 간의 네트워크 연결 해제에 대비하기 위해 보조 모니터링 및 로깅 백엔드를 설정하여 리전 AWS 서비스에 연결할 수 없을 때 애플리케이션의 관찰성을 유지할 수 있습니다. 예를 들어 지표와 로그를 여러 백엔드로 전송하도록 AWS Distro for Open Telemetry(ADOT) 수집기를 구성할 수 있습니다. CLI와 같은 로컬 도구를 사용하여 포드 및 컨테이너와 로컬로 상호 작용`crictl`하여 `kubectl` 또는 일반적으로 Kubernetes API 서버 엔드포인트를 쿼리하는 다른 Kubernetes API 호환 클라이언트를 대체할 수도 있습니다. 에 대한 자세한 내용은 Cri-tools GitHub의 `crictl` 설명서를 `crictl`참조하세요. 다음은 몇 가지 유용한 `crictl` 명령입니다.

호스트에서 실행 중인 포드 나열:

```
crictl pods
```

호스트에서 실행 중인 컨테이너 나열:

```
crictl ps
```

호스트에서 실행 중인 이미지를 나열합니다.

```
crictl images
```

호스트에서 실행 중인 컨테이너의 로그를 가져옵니다.

```
crictl logs CONTAINER_NAME
```

호스트에서 실행 중인 포드의 통계를 가져옵니다.

```
crictl statsp
```

애플리케이션 네트워크 트래픽

하이브리드 노드를 사용할 때는 애플리케이션 트래픽의 네트워크 흐름과 애플리케이션을 클러스터 외부에 노출하는 데 사용하는 기술을 고려하고 이해하는 것이 중요합니다. 애플리케이션 로드 밸런싱 및 수신을 위한 다양한 기술은 네트워크 연결 해제 중에 다르게 작동합니다. 예를 들어 애플리케이션 로드 밸런싱에 Cilium의 BGP 제어 플레인 기능을 사용하는 경우 네트워크 연결 해제 중에 포드 및 서비스에 대한 BGP 세션이 중단될 수 있습니다. 이는 BGP 스피커 기능이 Cilium 에이전트와 통합되고 Kubernetes 컨트롤 플레인에서 연결이 끊어지면 Cilium 에이전트가 계속 다시 시작되기 때문입니다. 재시작 이유는 Cilium의 상태 확인이 Kubernetes 컨트롤 플레인에 대한 액세스와 결합되어 있기 때문에 Cilium의 상태 확인이 실패하기 때문입니다(Cilium v1.17의 옵트인 개선이 적용된 CFP: #31702 참조). 마찬가지로 AWS 리전에서 생성된 애플리케이션 트래픽에 Application Load Balancer(ALB) 또는 Network Load Balancer(NLB)를 사용하는 경우 온프레미스 환경이 AWS 리전에 대한 연결을 끊으면 해당 트래픽이 일시적으로 중단될 수 있습니다. 프로덕션에 배포하기 전에 네트워크 연결 해제 중에 로드 밸런싱 및 수신에 사용하는 기술이 안정적으로 유지되는지 확인하는 것이 좋습니다. aws-samples/eks-hybrid-examples GitHub 리포지토리의 예제는 L2 모드에서 로드 밸런싱을 위해 MetalLB를 사용하며, 이는 하이브리드 노드와 EKS 컨트롤 플레인 간의 네트워크 연결 해제 중에 안정적으로 유지됩니다.

원격 AWS 서비스에 대한 종속성 검토

하이브리드 노드를 사용할 때는 온프레미스 또는 엣지 환경 외부의 리전 AWS 서비스에 대해 수행하는 종속성을 알고 있어야 합니다. 예를 들어 Amazon S3 또는 Amazon RDS for 애플리케이션 데이터 액세스, 지표 및 로그에 Amazon Managed Service for Prometheus 또는 CloudWatch 사용, 리전에서 생성된 트래픽에 Application and Network Load Balancer 사용, Amazon Elastic Container Registry에서 컨테이너 가져오기 등이 있습니다. 온프레미스 환경과 AWS 간의 네트워크 연결 해제 중에는 이러한 서비스에 액세스할 수 없습니다. 온프레미스 환경에서 AWS와의 네트워크 연결이 끊기 쉬운 경우 AWS 서비스 사용을 검토하고 해당 서비스에 대한 연결이 끊어도 애플리케이션의 정적 안정성이 손상되지 않는지 확인합니다.

Kubernetes 포드 장애 조치 동작 조정

호스트 간에 이식할 수 없는 애플리케이션 또는 포드 장애 조치를 위한 예비 용량이 없는 리소스 제약 환경에서 네트워크 연결 해제 중에 포드 장애 조치 동작을 조정할 수 있는 옵션이 있습니다. 일반적으로 애플리케이션의 리소스 요구 사항을 고려하고 노드에 장애가 발생할 경우 하나 이상의 애플리케이션 인스턴스가 다른 호스트로 장애 조치할 수 있는 충분한 용량을 확보하는 것이 중요합니다.

- 옵션 1 - DaemonSets 사용:이 옵션은 클러스터의 모든 노드에서 실행될 수 있고 실행되어야 하는 애플리케이션에 적용됩니다. DaemonSets는 연결할 수 없는 테인트를 허용하도록 자동으로 구성되므로 DaemonSet 포드가 네트워크 연결 해제를 통해 노드에 바인딩됩니다.

- 옵션 2 - 연결할 수 없는 테인트에 tolerationSeconds 맞게 조정: 네트워크 연결 해제 중에 포드가 노드에 바인딩된 상태로 유지되는 시간을 조정할 수 있습니다. 이렇게 하려면 지정한 기간(tolerationSeconds애플리케이션 사양의) 동안 NoExecute 효과로 연결할 수 없는 테인트를 허용하도록 애플리케이션 포드를 구성합니다. 이 옵션을 사용하면 네트워크 연결이 끊어지면 애플리케이션 포드가 tolerationSeconds 만료될 때까지 노드에 바인딩된 상태로 유지됩니다. tolerationSeconds를 사용하여 연결할 수 없는 테인트를 늘리NoExecute면 연결할 수 없는 호스트에서 실행되는 포드가 다른 연결 가능한 정상 호스트로 이동하는 데 시간이 더 오래 걸릴 수 있으므로 신중하게 고려하세요.
- 옵션 3: 사용자 지정 컨트롤러: NoExecute 효과가 있는 연결할 수 없는 테인트에 대해 Kubernetes를 모니터링하는 사용자 지정 컨트롤러(또는 기타 소프트웨어)를 생성하고 실행할 수 있습니다. 이 테인트가 감지되면 사용자 지정 컨트롤러는 애플리케이션별 지표를 확인하여 애플리케이션 상태를 평가할 수 있습니다. 애플리케이션이 정상인 경우 사용자 지정 컨트롤러는 연결할 수 없는 테인트를 제거하여 네트워크 연결 해제 중에 노드에서 포드가 제거되지 않도록 할 수 있습니다.

다음은 연결할 수 없는 테인트에 tolerationSeconds 대해를 사용하여 배포를 구성하는 방법의 예입니다. 이 예에서 tolerationSeconds는 1800 (30분)로 설정되어 있습니다. 즉, 네트워크 연결 해제가 30분 이상 지속되는 경우에만 연결할 수 없는 노드에서 실행되는 포드가 제거됩니다.

```
apiVersion: apps/v1
kind: Deployment
metadata:
...
spec:
...
      tolerations:
      - key: "node.kubernetes.io/unreachable"
        operator: "Exists"
        effect: "NoExecute"
        tolerationSeconds: 1800
```

네트워크 연결 해제를 통한 Kubernetes 포드 장애 조치

먼저 노드와 Kubernetes 컨트롤 플레인 간의 네트워크 연결 해제 중에 Kubernetes가 작동하는 방식에 영향을 미치는 주요 개념, 구성 요소 및 설정을 검토합니다. EKS는 업스트림 Kubernetes 규정을 준수하므로 여기에 설명된 모든 Kubernetes 개념, 구성 요소 및 설정이 EKS 및 EKS Hybrid Nodes 배포에 적용됩니다.

네트워크 연결 해제 중 포드 장애 조치 동작을 개선하기 위해 특히 EKS가 개선되었습니다. 자세한 내용은 업스트림 Kubernetes 리포지토리의 GitHub 문제 #131294 및 #131481을 참조하세요.

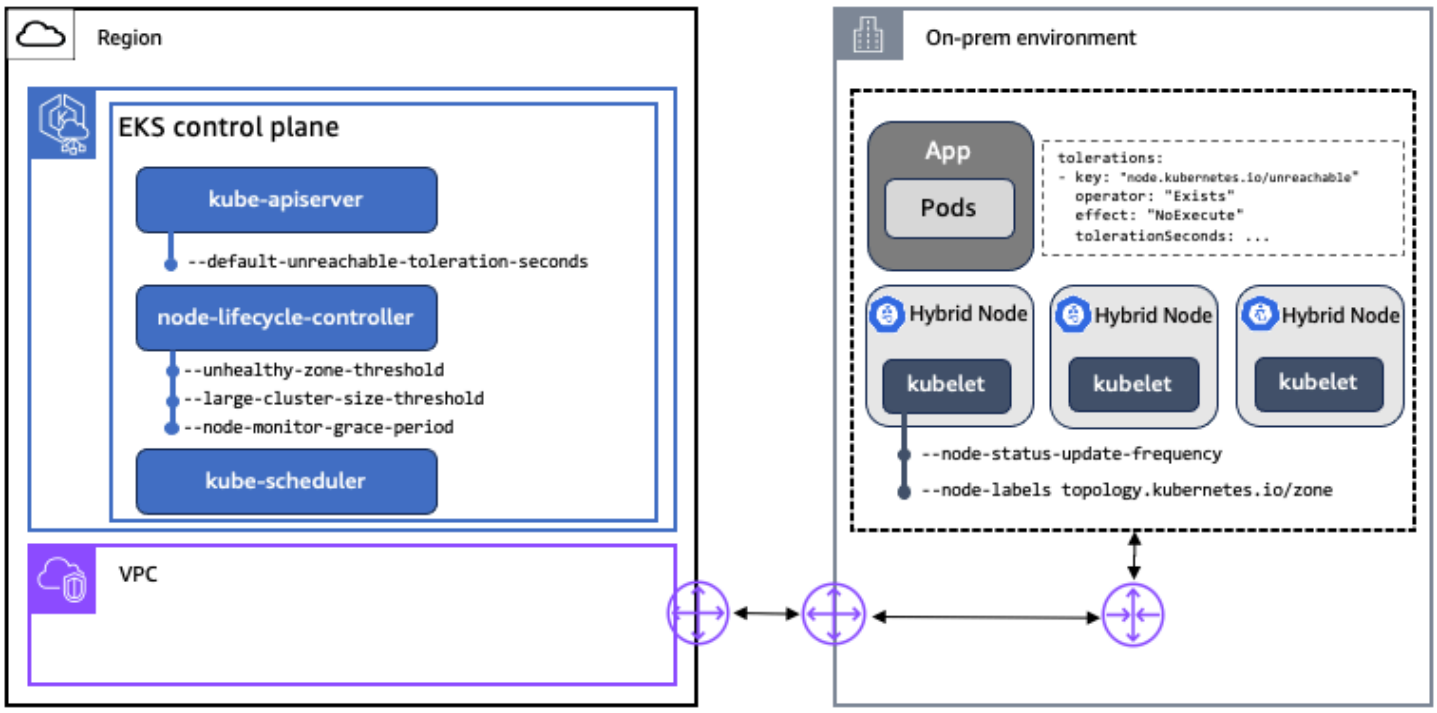
개념

테인트 및 허용치: 테인트 및 허용치는 Kubernetes에서 노드에 대한 포드 예약을 제어하는 데 사용됩니다. 테인트는 node-lifecycle-controller에 의해 설정되어 노드가 예약에 적합하지 않거나 해당 노드의 포드를 제거해야 함을 나타냅니다. 네트워크 연결 해제로 인해 노드에 연결할 수 없는 경우 node-lifecycle-controller는 NoSchedule 효과와 특정 조건이 충족되면 NoExecute 효과와 함께 node.kubernetes.io/unreachable 테인트를 적용합니다. node.kubernetes.io/unreachable 테인은 NodeCondition Ready를 알 수 없음에 해당합니다. 사용자는 PodSpec의 애플리케이션 수준에서 테인트에 대한 허용치를 지정할 수 있습니다.

- NoSchedule: 일치하는 허용 오차가 없는 한 테인트된 노드에 새 포드가 예약되지 않습니다. 노드에서 이미 실행 중인 포드는 제거되지 않습니다.
- NoExecute: 테인트를 허용하지 않는 포드는 즉시 제거됩니다. 테인트를 허용하는 포드(tolerationSeconds를 지정하지 않음)는 영구적으로 바인딩됩니다. 지정된 tolerationSeconds로 테인트를 허용하는 포드는 지정된 시간 동안 바인딩된 상태로 유지됩니다. 이 시간이 경과하면 노드 수명 주기 컨트롤러는 노드에서 포드를 제거합니다.

노드 임대: Kubernetes는 임대 API를 사용하여 Kubernetes API 서버에 kubelet 노드 하트비트를 전달합니다. 모든 노드에는 일치하는 이름을 가진 임대 객체가 있습니다. 내부적으로 각 kubelet 하트비트는 리스 객체의 spec.renewTime 필드를 업데이트합니다. Kubernetes 컨트롤 플레인은이 필드의 타임스탬프를 사용하여 노드 가용성을 결정합니다. 노드가 Kubernetes 컨트롤 플레인에서 연결 해제된 경우 리스에 대한 spec.renewTime을 업데이트할 수 없으며 컨트롤 플레인은 이를 NodeCondition Ready를 알 수 없음으로 해석합니다.

구성 요소



구성 요소	하위 구성 요소	설명
Kubernetes 컨트롤 플레인	kube-api-server	API 서버는 Kubernetes API를 노출하는 Kubernetes 컨트롤 플레인의 핵심 구성 요소입니다.
Kubernetes 컨트롤 플레인	node-lifecycle-controller	kube-controller-manager가 실행하는 컨트롤러 중 하나입니다. 노드 문제를 감지하고 대응할 책임이 있습니다.
Kubernetes 컨트롤 플레인	kube-scheduler	노드가 할당되지 않은 새로 생성된 포드를 감시하고 실행할 노드를 선택하는 컨트롤 플레인 구성 요소입니다.
Kubernetes 노드	kubelet	클러스터의 각 노드에서 실행되는 에이전트입니다. kubelet은 PodSpecs를 감시하고 해당 PodSpecs에 설명된 컨테이너가 실행 중이고 정상인지 확인합니다.

구성 설정

구성 요소	설정	설명	K8s 기본 값	EKS 기본 값	EKS에서 구성 가능
kube-api-server	default-unreachable-toleration-seconds	이러한 허용치`unreachable:NoExecute` 가 아직 없는 모든 포드에 기본적으로 추가되는 `tolerationSeconds` 에 대한 허용치의를 나타냅니다.	300	300	아니요
node-lifecycle-controller	node-monitor-grace-period	비정상으로 표시되기 전에 노드가 응답하지 않을 수 있는 시간입니다. kubelet의 보다 N 배여야 합니다. `nodeStatusUpdateFrequency` 여기서 N은 kubelet이 노드 상태를 게시하는 데 허용되는 재시도 횟수입니다.	40	40	아니요
node-lifecycle-controller	large-cluster-size-threshold	노드 node-lifecycle-controller 처리하는 노드 수입니다. `--secondary-node-eviction-rate` 는이 크기 이하의 클러스터의 경우 0으로 재정의됩니다.	50	100,000건	아니요
node-lifecycle-controller	unhealthy-zone-threshold	해당 영역을 비정상으로 처리하려면 준비되지 않음이어야 하는 영역의 노드 비율입니다.	55%	55%	아니요
kubelet	node-status-update-frequency	kubelet이 컨트롤 플레인에 노드 상태를 게시하는 빈도입니다. node-lifecycle-controller`nodeMonitorGracePeriod` 에서와 호환되어야 합니다.	10	10	예
kubelet	노드 레이블	클러스터에 노드를 등록할 때 추가할 레이블입니다. 하이브리드 노드로 레이블	없음	없음	예

구성 요소	설정	설명	K8s 기본 값	EKS 기본 값	EKS에서 구성 가능
		을 지정하여 노드를 영역으로 그룹화할 `topology.kubernetes.io/zone` 수 있습니다.			

네트워크 연결 해제를 통한 Kubernetes 포드 장애 조치

여기에 설명된 동작은 포드가 기본 설정으로 Kubernetes 배포로 실행되고 EKS가 Kubernetes 공급자로 사용된다고 가정합니다. 실제 동작은 환경, 네트워크 연결 해제 유형, 애플리케이션, 종속성 및 클러스터 구성에 따라 다를 수 있습니다. 이 가이드의 내용은 특정 애플리케이션, 클러스터 구성 및 플러그인의 하위 집합을 사용하여 검증되었습니다. 프로덕션으로 전환하기 전에 자체 환경과 자체 애플리케이션에서 동작을 테스트하는 것이 좋습니다.

노드와 Kubernetes 컨트롤 플레인 간에 네트워크 연결이 끊어지면 연결이 끊긴 각 노드의 kubelet이 Kubernetes 컨트롤 플레인과 통신할 수 없습니다. 따라서 연결이 복원될 때까지 kubelet이 해당 노드에서 포드를 제거할 수 없습니다. 즉, 네트워크 연결 해제 전에 해당 노드에서 실행 중인 포드는 연결 해제 중에 계속 실행되며, 다른 장애로 인해 종료되지 않는다고 가정합니다. 요약하면 노드와 Kubernetes 컨트롤 플레인 간의 네트워크 연결 해제 중에 정적 안정성을 달성할 수 있지만 연결이 복원될 때까지 노드 또는 워크로드에서 변경 작업을 수행할 수 없습니다.

네트워크 연결 해제의 특성에 따라 다양한 포드 장애 조치 동작을 생성하는 다섯 가지 주요 시나리오가 있습니다. 모든 시나리오에서 노드가 Kubernetes 컨트롤 플레인에 다시 연결되면 운영자 개입 없이 클러스터가 다시 정상 상태가 됩니다. 아래 시나리오에서는 관찰 결과에 따라 예상되는 결과를 간략하게 설명하지만 이러한 결과가 가능한 모든 애플리케이션 및 클러스터 구성에 적용되지 않을 수 있습니다.

시나리오 1: 전체 클러스터 중단

예상 결과: 연결할 수 없는 노드의 포드는 제거되지 않으며 해당 노드에서 계속 실행됩니다.

전체 클러스터 중단은 클러스터의 모든 노드가 Kubernetes 컨트롤 플레인에서 연결 해제되었음을 의미합니다. 이 시나리오에서 컨트롤 플레인의 node-lifecycle-controller는 클러스터의 모든 노드에 연결할 수 없음을 감지하고 포드 제거를 취소합니다.

클러스터 관리자는 연결 해제 `Not Ready` 중에 상태의 모든 노드를 볼 수 있습니다. 포드 상태는 변경되지 않으며 연결 해제 및 후속 재연결 중에 노드에 새 포드가 예약되지 않습니다.

시나리오 2: 전체 영역 중단

예상 결과: 연결할 수 없는 노드의 포드는 제거되지 않으며 해당 노드에서 계속 실행됩니다.

전체 영역 중단은 영역의 모든 노드가 Kubernetes 컨트롤 플레인에서 연결 해제되었음을 의미합니다. 이 시나리오에서 컨트롤 플레인의 node-lifecycle-controller는 영역의 모든 노드에 연결할 수 없음을 감지하고 포드 제거를 취소합니다.

클러스터 관리자는 연결 해제 `Not Ready` 중에 상태의 모든 노드를 볼 수 있습니다. 포드 상태는 변경되지 않으며 연결 해제 및 후속 재연결 중에 노드에 새 포드가 예약되지 않습니다.

시나리오 3: 대부분 영역 중단

예상 결과: 연결할 수 없는 노드의 포드는 제거되지 않으며 해당 노드에서 계속 실행됩니다.

대다수 영역 중단은 지정된 영역의 대부분의 노드가 Kubernetes 컨트롤 플레인에서 연결 해제됨을 의미합니다. Kubernetes의 영역은 `topology.kubernetes.io/zone` 레이블이 동일한 노드로 정의됩니다. 클러스터에 영역이 정의되지 않은 경우 대부분의 중단은 전체 클러스터의 노드 대부분이 연결 해제됨을 의미합니다. 기본적으로 대다수는 Kubernetes와 EKS 모두에서 55%로 `unhealthy-zone-threshold`설정된 node-lifecycle-controller의에 의해 정의됩니다. `large-cluster-size-threshold`는 EKS에서 100,000으로 설정되므로 영역 내 노드의 55% 이상에 연결할 수 없는 경우 포드 제거가 취소됩니다(대부분의 클러스터가 100,000개 노드보다 훨씬 작은 경우).

클러스터 관리자는 연결 해제 `Not Ready` 중에 상태의 대부분의 노드를 영역에서 볼 수 있지만 포드의 상태는 변경되지 않으며 다른 노드에서 다시 예약되지 않습니다.

위의 동작은 3개 이상의 노드가 있는 클러스터에만 적용됩니다. 노드가 3개 이하인 클러스터에서는 연결할 수 없는 노드의 포드가 제거되도록 예약되고 정상 노드에 새 포드가 예약됩니다.

테스트 중에 대부분의 영역 노드에 연결할 수 없는 경우에도 네트워크 연결 해제 중에 정확히 하나의 연결할 수 없는 노드에서 포드가 제거되는 경우가 가끔 있었습니다. Kubernetes node-lifecycle-controller에서 이러한 동작의 원인으로 발생할 수 있는 경합 상태를 계속 조사하고 있습니다.

시나리오 4: 소수 영역 중단

예상 결과: 연결할 수 없는 노드에서 포드가 제거되고 사용 가능한 적격 노드에 새 포드가 예약됩니다.

소수의 중단은 영역의 노드 중 더 적은 비율이 Kubernetes 컨트롤 플레인에서 연결 해제됨을 의미합니다. 클러스터에 영역이 정의되지 않은 경우 소수의 중단은 전체 클러스터에 있는 소수의 노드가 연결 해제됨을 의미합니다. 명시된 대로 소수는 기본적으로 55%인 node-lifecycle-controller의 `unhealthy-zone-threshold` 설정에 의해 정의됩니다. 이 시나리오에서 네트워크 연결 해제가

`default-unreachable-toleration-seconds` (5분) 및 `node-monitor-grace-period` (40초)보다 오래 지속되고 영역 내 노드의 55% 미만이 연결할 수 없는 경우 새 포드는 정상 노드에 예약되고 연결할 수 없는 노드의 포드는 제거 대상으로 표시됩니다.

클러스터 관리자는 정상 노드에서 생성된 새 포드를 보고 연결이 해제된 노드의 포드는 로 표시됩니다`Terminating`. 연결이 해제된 노드의 포드는 `Terminating` 상태가 되더라도 노드가 Kubernetes 컨트롤 플레인에 다시 연결될 때까지 완전히 제거되지 않습니다.

시나리오 5: 네트워크 중단 중 노드 재시작

예상 결과: 연결할 수 없는 노드의 포드는 노드가 Kubernetes 컨트롤 플레인에 다시 연결될 때까지 시작되지 않습니다. 포드 장애 조치는 연결할 수 없는 노드 수에 따라 시나리오 1~3에 설명된 로직을 따릅니다.

네트워크 중단 중 노드 재시작은 네트워크 연결 해제와 동시에 노드에서 다른 장애(예: 전원 주기, out-of-memory 이벤트 또는 기타 문제)가 발생했음을 의미합니다. 네트워크 연결 해제가 시작될 때 해당 노드에서 실행 중이었던 포드는 kubelet도 다시 시작되었다면 연결 해제 중에 자동으로 다시 시작되지 않습니다. kubelet은 시작 중에 Kubernetes API 서버를 쿼리하여 실행해야 하는 포드를 알아봅니다. 네트워크 연결 해제로 인해 kubelet이 API 서버에 연결할 수 없는 경우 포드를 시작하는 데 필요한 정보를 검색할 수 없습니다.

이 시나리오에서는 `crictl` CLI와 같은 로컬 문제 해결 도구를 사용하여 포드를 'break-glass' 측정값으로 수동으로 시작할 수 없습니다. Kubernetes는 일반적으로 기존 포드를 다시 시작하는 대신 장애가 발생한 포드를 제거하고 새 포드를 생성합니다(자세한 내용은 컨테이너화된 GitHub 리포지토리의 #10213 참조). 정적 포드는 kubelet에 의해 제어되고 이러한 시나리오 중에 다시 시작할 수 있는 유일한 Kubernetes 워크로드 객체입니다. 그러나 일반적으로 애플리케이션 배포에 정적 포드를 사용하지 않는 것이 좋습니다. 대신 여러 호스트에 여러 복제본을 배포하여 노드 장애 및 노드와 Kubernetes 컨트롤 플레인 간의 네트워크 연결 해제와 같은 여러 동시 장애 발생 시 애플리케이션 가용성을 보장합니다.

네트워크 연결 해제를 통한 애플리케이션 네트워크 트래픽

이 페이지의 주제는 노드와 Kubernetes 컨트롤 플레인 간의 네트워크 연결 해제 중 Kubernetes 클러스터 네트워킹 및 애플리케이션 트래픽과 관련이 있습니다.

Cilium

Cilium에는 IP 주소 관리(IPAM), 캡슐화, 로드 밸런싱 및 클러스터 라우팅을 위한 여러 모드가 있습니다. 이 가이드에서 검증된 모드는 클러스터 범위 IPAM, VXLAN 오버레이, BGP 로드 밸런싱 및 kube-

proxy를 사용했습니다. Cilium은 BGP 로드 밸런싱 없이도 사용되어 MetalLB L2 로드 밸런싱으로 대체되었습니다.

Cilium 설치의 기본은 Cilium 연산자와 Cilium 에이전트로 구성됩니다. Cilium 연산자는 배포로 실행되고 Cilium 사용자 지정 리소스 정의(CRDs를 등록하고, IPAM을 관리하고, 클러스터 객체를 다른 기능 중에서 Kubernetes API 서버와 동기화합니다. Cilium 에이전트는 각 노드에서 DaemonSet로 실행되고 eBPF 프로그램을 관리하여 클러스터에서 실행되는 워크로드에 대한 네트워크 규칙을 제어합니다.

일반적으로 Cilium에서 구성한 클러스터 내 라우팅은 네트워크 연결 해제 중에도 계속 사용할 수 있으며 포드 네트워크에 대한 클러스터 내 트래픽 흐름 및 IP 테이블(iptables) 규칙을 준수하여 확인할 수 있습니다.

```
ip route show table all | grep cilium
```

```
10.86.2.0/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16 mtu 1450
10.86.2.64/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16 mtu 1450
10.86.2.128/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16 mtu 1450
10.86.2.192/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16 mtu 1450
10.86.3.0/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16
10.86.3.16 dev cilium_host proto kernel scope link
...
```

그러나 네트워크 연결 해제 중에 상태 확인이 Kubernetes API 서버와의 연결 상태와 결합되어 Cilium 운영자와 Cilium 에이전트가 다시 시작됩니다. 네트워크 연결 해제 중에 Cilium 운영자 및 Cilium 에이전트의 로그에 다음이 표시될 것으로 예상됩니다. 네트워크 연결 해제 중에 `crictl` CLI와 같은 도구를 사용하여 로그를 포함하여 이러한 구성 요소의 재시작을 관찰할 수 있습니다.

```
msg="Started gops server" address="127.0.0.1:9890" subsys=gops
msg="Establishing connection to apiserver" host="https://<k8s-cluster-ip>:443"
 subsys=k8s-client
msg="Establishing connection to apiserver" host="https://<k8s-cluster-ip>:443"
 subsys=k8s-client
msg="Unable to contact k8s api-server" error="Get \"https://<k8s-cluster-ip>:443/
api/v1/namespaces/kube-system\": dial tcp <k8s-cluster-ip>:443: i/o timeout"
 ipAddr="https://<k8s-cluster-ip>:443" subsys=k8s-client
msg="Start hook failed" function="client.(*compositeClientset).onStart
 (agent.infra.k8s-client)" error="Get \"https://<k8s-cluster-ip>:443/api/v1/namespaces/
kube-system\": dial tcp <k8s-cluster-ip>:443: i/o timeout"
msg="Start failed" error="Get \"https://<k8s-cluster-ip>:443/api/v1/namespaces/kube-
system\": dial tcp <k8s-cluster-ip>:443: i/o timeout" duration=1m5.003834026s
msg=Stopping
```

```
msg="Stopped gops server" address="127.0.0.1:9890" subsys=gops
msg="failed to start: Get \"https://<k8s-cluster-ip>:443/api/v1/namespaces/kube-system
\": dial tcp <k8s-cluster-ip>:443: i/o timeout" subsys=daemon
```

애플리케이션 로드 밸런싱에 Cilium의 BGP 컨트롤 플레인 기능을 사용하는 경우 BGP 스피커 기능이 Cilium 에이전트와 통합되고 Kubernetes 컨트롤 플레인에서 연결이 해제되면 Cilium 에이전트가 계속 다시 시작되므로 네트워크 연결 해제 중에 포드 및 서비스에 대한 BGP 세션이 중단될 수 있습니다. 자세한 내용은 Cilium 설명서의 Cilium BGP 컨트롤 플레인 작업 가이드를 참조하세요. 또한 전원 주기 또는 시스템 재부팅과 같은 네트워크 연결 해제 중에 동시 장애가 발생하는 경우, 노드가 Kubernetes 컨트롤 플레인에 다시 연결되고 Cilium이 다시 시작될 때 경로가 다시 생성되더라도 이러한 작업을 통해 Cilium 경로가 보존되지 않습니다.

Calico

곧 출시 예정

MetalLB

MetalLB에는 L2 모드와 BGP 모드라는 두 가지 로드 밸런싱 모드가 있습니다. 이러한 로드 밸런싱 모드의 작동 방식과 제한 사항에 대한 자세한 내용은 MetalLB 설명서를 참조하세요. 이 가이드의 검증은 L2 모드에서 MetalLB를 사용했습니다.이 모드에서는 클러스터의 한 시스템이 Kubernetes Service의 소유권을 가지며 IPv4용 ARP를 사용하여 로컬 네트워크에서 로드 밸런서 IP 주소에 연결할 수 있습니다. MetalLB를 실행할 때 IP 할당을 담당하는 컨트롤러와 IP 주소가 할당된 광고 서비스를 담당하는 각 노드에서 실행되는 스피커가 있습니다. MetalLB 컨트롤러는 배포로 실행되고 MetalLB 스피커는 DaemonSet로 실행됩니다. 네트워크 연결이 끊어지는 동안 MetalLB 컨트롤러와 스피커는 Kubernetes API 서버에서 클러스터 리소스를 관찰하지 못하지만 계속 실행됩니다. 무엇보다도 외부 연결에 MetalLB를 사용하는 서비스는 네트워크 연결 해제 중에도 계속 사용할 수 있고 액세스할 수 있습니다.

kube-proxy

EKS 클러스터에서 kube-proxy는 각 노드에서 DaemonSet로 실행되며 서비스 IP 주소를 기본 포드의 IP 주소로 변환하여 서비스와 포드 간의 통신을 가능하게 하는 네트워크 규칙을 관리하는 역할을 합니다. kube-proxy로 구성된 IP 테이블(iptables) 규칙은 네트워크 연결 해제 중에 유지되며 클러스터 내 라우팅은 계속 작동하고 kube-proxy 포드는 계속 실행됩니다.

다음 iptables 명령을 사용하여 kube-proxy 규칙을 관찰할 수 있습니다. 첫 번째 명령은 `PREROUTING` 체인을 통과하는 패킷이 `KUBE-SERVICES` 체인으로 전달되는 것을 보여줍니다.

```
iptables -t nat -L PREROUTING
```

```
Chain PREROUTING (policy ACCEPT)
target         prot opt source      destination
KUBE-SERVICES  all  --  anywhere    anywhere      /* kubernetes service portals */
```

KUBE-SERVICES 체인을 검사하면 다양한 클러스터 서비스에 대한 규칙을 볼 수 있습니다.

```
Chain KUBE-SERVICES (2 references)
target                     prot opt source      destination
KUBE-SVL-NZTS37XDTDNXGCKJ  tcp  --  anywhere    172.16.189.136  /* kube-system/hubble-
peer:peer-service cluster IP /
KUBE-SVC-2BINP2AXJOTI3HJ5  tcp  --  anywhere    172.16.62.72    / default/metallb-
webhook-service cluster IP /
KUBE-SVC-LRNEBRA3Z5YGJ4QC  tcp  --  anywhere    172.16.145.111  / default/redis-leader
 cluster IP /
KUBE-SVC-I7SKRZYQ7PWYV5X7  tcp  --  anywhere    172.16.142.147  / kube-system/eks-
extension-metrics-api:metrics-api cluster IP /
KUBE-SVC-JD5MR3NA4I4DYORP  tcp  --  anywhere    172.16.0.10     / kube-system/kube-
dns:metrics cluster IP /
KUBE-SVC-TCOU7JCQXEZGVUNU  udp  --  anywhere    172.16.0.10     / kube-system/kube-
dns:dns cluster IP /
KUBE-SVC-ERIFXISQEP7F7OF4  tcp  --  anywhere    172.16.0.10     / kube-system/kube-
dns:dns-tcp cluster IP /
KUBE-SVC-ENODL3HWJ5BZY56Q  tcp  --  anywhere    172.16.7.26     / default/frontend
 cluster IP /
KUBE-EXT-ENODL3HWJ5BZY56Q  tcp  --  anywhere    <LB-IP>    / default/frontend
 loadbalancer IP /
KUBE-SVC-NPX46M4PTMTKRN6Y  tcp  --  anywhere    172.16.0.1      / default/
kubernetes:https cluster IP /
KUBE-SVC-YU5RV2YQWHLZ5XPR  tcp  --  anywhere    172.16.228.76   / default/redis-
follower cluster IP /
KUBE-NODEPORTS             all  --  anywhere    anywhere        / kubernetes service
 nodeports; NOTE: this must be the last rule in this chain */
```

프런트엔드 서비스의 체인에서 애플리케이션을 검사하면 서비스를 지원하는 포드 IP 주소를 확인할 수 있습니다.

```
iptables -t nat -L KUBE-SVC-ENODL3HWJ5BZY56Q
```

```
Chain KUBE-SVC-ENODL3HWJ5BZY56Q (2 references)
target                     prot opt source    destination
KUBE-SEP-EKXE7ASH7Y74BGBO  all  --  anywhere  anywhere    /* default/frontend ->
 10.86.2.103:80 / statistic mode random probability 0.33333333349
```

```
KUBE-SEP-GCY3OUXWSVMSEAR6  all  --  anywhere  anywhere    / default/frontend ->
 10.86.2.179:80 / statistic mode random probability 0.50000000000
KUBE-SEP-6GJJR3EF5AUP2WBU  all  --  anywhere  anywhere    / default/frontend ->
 10.86.3.47:80 */
```

다음 kube-proxy 로그 메시지는 네트워크 연결 해제 중에 Kubernetes API 서버에서 노드 및 엔드포인트 리소스에 대한 업데이트를 감시하려고 할 때 예상됩니다.

```
"Unhandled Error" err="k8s.io/client-go/informers/factory.go:160: Failed to watch
 *v1.Node: failed to list *v1.Node: Get \"https://<k8s-endpoint>/api/v1/nodes?
fieldSelector=metadata.name%3D<node-name>&resourceVersion=2241908\": dial tcp <k8s-
ip>:443: i/o timeout" logger="UnhandledError"
"Unhandled Error" err="k8s.io/client-go/informers/factory.go:160: Failed to watch
 *v1.EndpointSlice: failed to list *v1.EndpointSlice: Get \"https://<k8s-endpoint>/
apis/discovery.k8s.io/v1/endpointslices?labelSelector=%21service.kubernetes.io
%2Fheadless%2C%21service.kubernetes.io%2Fservice-proxy-name&resourceVersion=2242090\":
 dial tcp <k8s-ip>:443: i/o timeout" logger="UnhandledError"
```

CoreDNS

기본적으로 EKS 클러스터의 포드는 CoreDNS 클러스터 IP 주소를 클러스터 내 DNS 쿼리의 이름 서버로 사용합니다. EKS 클러스터에서 CoreDNS는 노드에서 배포로 실행됩니다. 하이브리드 노드를 사용하면 하이브리드 노드에서 로컬로 실행되는 CoreDNS 복제본이 있는 경우 네트워크 연결 해제 중에 포드가 CoreDNS와 계속 통신할 수 있습니다. 클라우드에 노드가 있고 온프레미스 환경에 하이브리드 노드가 있는 EKS 클러스터가 있는 경우 각 환경에 하나 이상의 CoreDNS 복제본을 두는 것이 좋습니다. CoreDNS는 네트워크 연결 해제 전에 생성된 레코드에 대한 DNS 쿼리를 계속 제공하고 정적 안정성을 위해 네트워크 재연결을 통해 계속 실행됩니다.

다음 CoreDNS 로그 메시지는 Kubernetes API 서버의 객체를 나열하려고 할 때 네트워크 연결 해제 중에 예상됩니다.

```
Failed to watch *v1.Namespace: failed to list *v1.Namespace: Get "https://<k8s-cluster-
ip>:443/api/v1/namespaces?resourceVersion=2263964": dial tcp <k8s-cluster-ip>:443: i/o
 timeout
Failed to watch *v1.Service: failed to list *v1.Service: Get "https://<k8s-cluster-
ip>:443/api/v1/services?resourceVersion=2263966": dial tcp <k8s-cluster-ip>:443: i/o
 timeout
Failed to watch *v1.EndpointSlice: failed to list *v1.EndpointSlice: Get "https://<k8s-
cluster-ip>:443/apis/discovery.k8s.io/v1/endpointslices?resourceVersion=2263896": dial
 tcp <k8s-cluster-ip>: i/o timeout
```

네트워크 연결 해제를 통해 자격 증명 호스팅

EKS Hybrid Nodes는 EKS 컨트롤 플레인으로 노드를 인증하는 데 사용되는 임시 IAM 자격 증명을 위해 AWS Systems Manager(SSM) 하이브리드 활성화 및 AWS IAM Roles Anywhere와 통합됩니다. SSM과 IAM Roles Anywhere 모두 온프레미스 호스트에서 관리하는 임시 자격 증명을 자동으로 새로 고칩니다. 클러스터의 하이브리드 노드에서 SSM 하이브리드 활성화 또는 IAM Roles Anywhere와 같은 단일 자격 증명 공급자를 사용하는 것이 좋지만 둘 다 사용하는 것은 아닙니다.

SSM 하이브리드 활성화

SSM에서 프로비저닝한 임시 자격 증명은 1시간 동안 유효합니다. SSM을 자격 증명 공급자로 사용할 때는 자격 증명 유효 기간을 변경할 수 없습니다. 임시 자격 증명은 만료되기 전에 SSM에 의해 자동으로 교체되며, 교체는 노드 또는 애플리케이션의 상태에 영향을 주지 않습니다. 그러나 SSM 에이전트와 SSM 리전 엔드포인트 간에 네트워크 연결이 끊어지면 SSM이 자격 증명을 새로 고칠 수 없으며 자격 증명이 만료될 수 있습니다.

SSM은 SSM 리전 엔드포인트에 연결할 수 없는 경우 자격 증명 새로 고침 재시도에 지수 백오프를 사용합니다. SSM 에이전트 버전 `3.3.808.0` 이상(2024년 8월 릴리스)에서는 지수 백오프가 30분으로 제한됩니다. 네트워크 연결 해제 기간에 따라 SSM이 자격 증명을 새로 고치는 데 최대 30분이 걸릴 수 있으며, 자격 증명이 새로 고쳐질 때까지 하이브리드 노드가 EKS 컨트롤 플레인에 다시 연결되지 않습니다. 이 시나리오에서는 SSM 에이전트를 다시 시작하여 자격 증명을 강제로 새로 고칠 수 있습니다. 현재 SSM 자격 증명 새로 고침 동작의 부작용으로 각 노드의 SSM 에이전트가 자격 증명을 새로 고치는 시점에 따라 노드가 서로 다른 시간에 다시 연결될 수 있습니다. 따라서 이미 다시 연결된 노드에 아직 다시 연결되지 않은 노드에서 포드 장애 조치가 발생할 수 있습니다.

SSM 에이전트 버전을 가져옵니다. SSM 콘솔의 Fleet Manager 섹션을 확인할 수도 있습니다.

```
# AL2023, RHEL
yum info amazon-ssm-agent
# Ubuntu
snap list amazon-ssm-agent
```

SSM 에이전트를 다시 시작합니다.

```
# AL2023, RHEL
systemctl restart amazon-ssm-agent
# Ubuntu
systemctl restart snap.amazon-ssm-agent.amazon-ssm-agent
```

SSM 에이전트 로그 보기:

```
tail -f /var/log/amazon/ssm/amazon-ssm-agent.log
```

네트워크 연결 해제 중 예상되는 로그 메시지:

```
INFO [CredentialRefresher] Credentials ready
INFO [CredentialRefresher] Next credential rotation will be in 29.995040663666668
 minutes
ERROR [CredentialRefresher] Retrieve credentials produced error: RequestError: send
 request failed
INFO [CredentialRefresher] Sleeping for 35s before retrying retrieve credentials
ERROR [CredentialRefresher] Retrieve credentials produced error: RequestError: send
 request failed
INFO [CredentialRefresher] Sleeping for 56s before retrying retrieve credentials
ERROR [CredentialRefresher] Retrieve credentials produced error: RequestError: send
 request failed
INFO [CredentialRefresher] Sleeping for 1m24s before retrying retrieve credentials
```

IAM Roles Anywhere

IAM Roles Anywhere에서 프로비저닝한 임시 자격 증명은 기본적으로 1시간 동안 유효합니다. IAM Roles Anywhere 프로파일의 `durationSeconds` 필드를 통해 IAM Roles Anywhere로 자격 증명 유효 기간을 구성할 수 있습니다. 최대 자격 증명 유효 기간은 12시간입니다. 하이브리드 노드 IAM 역할의 `MaxSessionDuration` 설정은 IAM Roles Anywhere 프로파일의 `durationSeconds` 설정보다 커야 합니다.

IAM Roles Anywhere를 하이브리드 노드의 자격 증명 공급자로 사용하는 경우 kubelet이 온디맨드 자격 증명을 얻기 위해 `aws_signing_helper credential-process`를 호출하기 때문에 네트워크 연결 해제 후 일반적으로 네트워크 복원 후 몇 초 내에 EKS 컨트롤 플레인에 다시 연결합니다. 하이브리드 노드 또는 네트워크 연결 해제와 직접 관련이 없지만 IAM Roles Anywhere를 사용할 때 인증서 만료에 대한 알림 및 알림을 구성할 수 있습니다. 자세한 내용은 IAM Roles Anywhere의 알림 설정 사용자 지정을 참조하세요.

AI/ML 워크로드 실행 모범 사례

> **Tip**
>
> Get Hands on with EKS를 방문하여 예정된 Amazon EKS AI/ML 이벤트 및 워크숍에 대해 알아보세요.

EKS에서 AI/ML 워크로드를 실행할 때 모범 사례를 구현하면 해당 워크로드가 성능, 비용 효율성, 복원력 및 적절한 리소스 제공을 보장할 수 있습니다. 모범 사례는 컴퓨팅, 네트워킹, 스토리지, 관찰성 및 성능의 일반 섹션으로 나뉩니다.

Feedback

이 가이드는 광범위한 EKS/Kubernetes 커뮤니티에서 직접적인 피드백과 제안을 수집하기 위해 GitHub에서 릴리스됩니다. 가이드에 포함해야 한다고 생각되는 모범 사례가 있는 경우 GitHub 리포지토리에 문제를 제기하거나 PR을 제출하세요. 새로운 기능이 서비스에 추가되거나 새로운 모범 사례가 발전할 때 가이드를 정기적으로 업데이트하는 것이 목적입니다.

컴퓨팅 및 오토 스케일링

GPU 리소스 최적화 및 비용 관리

잘 알려진 레이블을 사용하여 GPU 요구 사항으로 워크로드 예약

다양한 GPU 특성(예: GPU, GPU 메모리)에 민감한 AI/ML 워크로드의 경우 Karpenter 및 관리형 노드 그룹에 사용되는 노드 유형에서 지원하는 알려진 예약 레이블을 사용하여 GPU 요구 사항을 지정하는 것이 좋습니다. 이를 정의하지 않으면 GPU 리소스가 부적절한 인스턴스에서 포드가 예약되어 실패하거나 성능이 저하될 수 있습니다. nodeSelector 또는 노드 선호도를 사용하여 포드가 실행할 노드를 지정하고 포드의 리소스 섹션에서 컴퓨팅 리소스(CPU, 메모리, GPUs 등)를 설정하는 것이 좋습니다.

예제

예를 들어 Karpenter를 사용할 때 GPU 이름 노드 선택기를 사용하는 경우:

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: gpu-pod-example
spec:
  containers:
  - name: ml-workload
    image: <image>
    resources:
      limits:
        nvidia.com/gpu: 1  # Request one NVIDIA GPU
  nodeSelector:
    karpenter.k8s.aws/instance-gpu-name: "l40s"  # Run on nodes with NVIDIA L40S GPUs
```

Kubernetes 디바이스 플러그인을 사용하여 GPUs 노출

노드에 GPUs를 노출하려면 Kubernetes 스케줄러가 사용 가능한 GPU가 있는 노드에 포드를 할당할 수 있도록 구성된 NVIDIA GPUs 드라이버를 노드의 운영 체제 및 컨테이너 런타임에 설치해야 합니다. NVIDIA Kubernetes 디바이스 플러그인의 설정 프로세스는 사용 중인 EKS 가속 AMI에 따라 달라집니다.

- Bottlerocket Accelerated AMI: 이 AMI에는 NVIDIA GPU 드라이버가 포함되어 있으며 NVIDIA Kubernetes 디바이스 플러그인은 사전 설치되어 즉시 사용할 수 있으므로 GPU 지원을 즉시 사용할 수 있습니다. Kubernetes 스케줄러에 GPUs를 노출하는 데 추가 구성이 필요하지 않습니다.
- AL2023 가속 AMI :이 AMI에는 NVIDIA GPU 드라이버가 포함되어 있지만 NVIDIA Kubernetes 디바이스 플러그인은 사전 설치되어 있지 않습니다. 일반적으로 DaemonSet를 통해 디바이스 플러그인을 별도로 설치하고 구성해야 합니다. eksctl을 사용하여 클러스터를 생성하고 ClusterConfig에서 GPU 인스턴스 유형(예: `g5.xlarge`)을 지정하는 경우 `eksctl`는 자동으로 적절한 AMI를 선택하고 NVIDIA Kubernetes 디바이스 플러그인을 설치합니다. 자세한 내용은 eksctl 설명서의 GPU 지원을 참조하세요.

대신 EKS 가속 AMIs 및 NVIDIA GPU 운영자를 사용하여 NVIDIA Kubernetes 디바이스 플러그인과 같은 구성 요소를 관리하려면 사전 설치된 NVIDIA GPU 드라이버 및 NVIDIA 컨테이너 툴킷 NVIDIA 설명서에 따라 NVIDIA GPU 드라이버 및 NVIDIA 컨테이너 툴킷의 관리를 비활성화해야 합니다.

NVIDIA 디바이스 플러그인이 활성 상태이고 GPUs가 올바르게 노출되었는지 확인하려면 다음을 실행합니다.

```
kubectl describe node | grep nvidia.com/gpu
```

이 명령은 nvidia.com/gpu 리소스가 노드의 용량과 할당 가능한 리소스에 있는지 확인합니다. 예를 들어 GPU가 하나 있는 노드에는가 표시되어야 합니다nvidia.com/gpu: 1. 자세한 내용은 Kubernetes GPU 예약 안내서를 참조하세요.

다양한 EC2 인스턴스 유형 사용

the section called “데이터 영역” 섹션에 설명된 대로 가능한 한 다양한 EC2 인스턴스 유형을 사용하는 것이 Amazon EKS의 확장성을 위한 중요한 모범 사례입니다. 이 권장 사항은 가속화된 하드웨어(예: GPUs 있는 인스턴스에도 적용됩니다. 하나의 인스턴스 유형만 사용하는 클러스터를 생성하고 리전 용량을 초과하여 노드 수를 조정하려고 하면 사용 가능한 인스턴스가 없음을 나타내는 용량 부족 오류(ICE)가 발생할 수 있습니다. 임의로 다각화하기 전에 AI/ML 워크로드의 고유한 특성을 이해하는 것이 중요합니다. EC2 인스턴스 유형 탐색기 도구를 사용하여 사용 가능한 인스턴스 유형을 검토하여 특정 컴퓨팅 요구 사항과 일치하는 인스턴스 유형 목록을 생성하고 클러스터에서 사용할 수 있는 인스턴스 유형을 임의로 제한하지 마세요.

가속 컴퓨팅 인스턴스는 단기, 중기 및 안정 상태 워크로드에 맞게 다양한 구매 모델로 제공됩니다. 예약을 피하려는 단기적이고 유연한 내결함성 워크로드의 경우 스팟 인스턴스를 살펴보세요. 용량 블록, 온디맨드 인스턴스 및 절감형 플랜을 사용하면 중장기 워크로드 기간에 가속화된 컴퓨팅 인스턴스를 프로비저닝할 수 있습니다. 원하는 구매 옵션에서 필요한 용량에 성공적으로 액세스할 가능성을 높이려면 다양한 인스턴스 유형 및 가용 영역 목록을 사용하는 것이 좋습니다. 또는 특정 구매 모델에 대해 ICEs가 발생하는 경우 다른 모델을 사용하여 다시 시도하세요.

예제 다음 예제에서는 Karpenter NodePool이 3세대(예: p3)보다 큰 G 및 P 인스턴스를 프로비저닝하도록 활성화하는 방법을 보여줍니다. 자세한 내용은 확장성 섹션을 참조하세요.

```
- key: karpenter.k8s.aws/instance-category
  operator: In
  values: ["g", "p"] # Diversifies across G-series and P-series
- key: karpenter.k8s.aws/instance-generation
  operator: Gt
  values: ["3"] # Selects instance generations greater than 3
```

GPUs용 스팟 인스턴스 사용에 대한 자세한 내용은 아래의 “Karpenter를 사용하여 GPUs용 Amazon EC2 스팟 인스턴스 사용 고려”를 참조하세요.

Karpenter에서 GPUs용 Amazon EC2 스팟 인스턴스 사용 고려

Amazon EC2 스팟 인스턴스를 사용하면 AWS 클라우드에서 미사용 EC2 용량을 활용할 수 있으며 온디맨드 요금에 비해 최대 90% 할인된 가격으로 사용할 수 있습니다. Amazon EC2 EC2 스팟 인스턴

스를 중단할 수 있습니다. 자세한 내용은 Amazon EC2 사용 설명서의 스팟 인스턴스를 참조하세요. Amazon EC2 스팟은 내결함성, 상태 비저장 및 유연한(시간 및 인스턴스 유형) 워크로드에 적합한 선택일 수 있습니다. 스팟 인스턴스 사용 시기에 대한 자세한 내용은 EC2 스팟 인스턴스 모범 사례를 참조하세요. 스팟 친화적인 경우 AI/ML 워크로드에 스팟 인스턴스를 사용할 수도 있습니다.

사용 사례

스팟 친화적 워크로드는 빅 데이터, 컨테이너화된 워크로드, CI/CD, 상태 비저장 웹 서버, 고성능 컴퓨팅(HPC) 및 렌더링 워크로드일 수 있습니다. 스팟 인스턴스는 인스턴스 노드 간에 유연하지 않거나, 상태 저장, 내결함성 또는 긴밀하게 결합된 워크로드(예: 컴퓨팅을 위해 서로 크게 의존하는 병렬 프로세스가 있는 워크로드로, 컴퓨팅 유체 역학 또는 복잡한 상호 종속성이 있는 분산 데이터베이스와 같은 MPI 기반 고성능 컴퓨팅 애플리케이션과 같은 일정한 노드 간 통신이 필요함)에는 적합하지 않습니다. 다음은 권장되는 특정 사용 사례입니다(특정 순서 없음).

- 실시간 온라인 추론: 워크로드가 스팟 친화적인 한 실시간 추론 워크로드의 비용 최적화 조정에 스팟 인스턴스를 사용합니다. 즉, 추론 시간은 2분 미만이고, 애플리케이션은 중단에 대한 내결함성이 있으며, 다양한 인스턴스 유형에서 실행될 수 있습니다. 잠재적인 스팟 중단을 처리하기 위해 애플리케이션 수준의 내결함성을 구현하면서 인스턴스 다양성(예: 여러 인스턴스 유형 및 가용 영역) 또는 예약을 통해 고가용성을 보장합니다.
- 하이퍼 파라미터 튜닝: 특히 단기 실험의 경우 중단이 큰 손실 없이 허용될 수 있으므로 스팟 인스턴스를 사용하여 탐색 튜닝 작업을 기회적으로 실행합니다.
- 데이터 증강: 스팟 인스턴스를 사용하여 중단될 경우 체크포인트에서 다시 시작할 수 있는 데이터 사전 처리 및 증강 작업을 수행하여 스팟의 변수 가용성에 이상적입니다.
- 모델 미세 조정: 강력한 체크포인트 메커니즘을 사용하여 스팟 인스턴스를 미세 조정하여 마지막으로 저장된 상태에서 재개하여 인스턴스 중단의 영향을 최소화합니다.
- 배치 추론: 스팟 인스턴스를 사용하여 작업을 일시 중지하고 재개할 수 있는 non-real-time 방식으로 대량의 오프라인 추론 요청을 처리하여 스팟의 비용 절감과 가장 잘 일치하고 재시도 또는 다각화를 통해 잠재적 중단을 처리합니다.
- 기회 훈련 하위 집합: 중단이 허용되고 인스턴스 유형 또는 리전 간 다각화와 같은 효율성 최적화를 적용할 수 있는 한계 또는 실험 훈련 워크로드(예: 파라미터가 1천만 개 미만인 소형 모델)에 스팟 인스턴스를 사용합니다. 단, 잠재적 중단으로 인한 프로덕션 규모 훈련에는 권장되지 않습니다.

고려 사항

Amazon EKS에서 가속화된 워크로드에 스팟 인스턴스를 사용하려면 다음과 같은 여러 가지 주요 고려사항이 있습니다(특정 순서 없음).

- Karpenter를 사용하여 고급 통합이 활성화된 스팟 인스턴스를 관리합니다. Karpenter NodePool에서 karpenter.sh/capacity-type "스팟"으로 지정하면 Karpenter는 추가 구성 없이 기본적으로 스팟 인스턴스를 프로비저닝합니다. 그러나 활용도가 낮은 스팟 노드를 저렴한 스팟 대안으로 대체하는 고급 Spot-to-Spot 통합을 활성화하려면 Karpenter 컨트롤러 인수에서 --feature-gates SpotToSpotConsolidation=true를 설정하거나 FEATURE_GATES 환경 변수를 통해 SpotToSpotConsolidation 기능 게이트를 활성화해야 합니다. Karpenter는 price-capacity-optimized 할당 전략을 사용하여 EC2 인스턴스를 프로비저닝합니다. NodePool 요구 사항 및 포드 제약 조건에 따라 Karpenter 빈은 예약할 수 없는 포드를 압축하고 다양한 인스턴스 유형 세트를 Amazon EC2 플릿 API로 전송합니다. EC2 인스턴스 유형 탐색기 도구를 사용하여 특정 컴퓨팅 요구 사항에 맞는 인스턴스 유형 목록을 생성할 수 있습니다.
- 워크로드가 상태 비저장, 내결함성 및 유연성을 갖추었는지 확인합니다. 워크로드는 상태 비저장, 내결함성, 인스턴스/GPU 크기 측면에서 유연해야 합니다. 이를 통해 스팟 중단 후 원활하게 재개할 수 있으며 인스턴스 유연성을 통해 잠재적으로 스팟을 더 오래 유지할 수 있습니다. 스팟 중단 이벤트를 포착하기 위해 AWS SQS 대기열의 이름으로 settings.interruptionQueue Helm 값을 구성하여 Karpenter에서 스팟 중단 처리를 활성화합니다.interruptionQueue 예를 들어 Helm을 통해 설치할 때는 --set "settings.interruptionQueue=${CLUSTER_NAME}"을 사용합니다. 예를 보려면 Karpenter 시작하기 안내서를 참조하세요. Karpenter는 스팟 중단 이벤트를 발견하면 중단 이벤트 전에 노드(들)를 자동으로 차단, 테인트, 드레이닝 및 종료하여 포드의 종료 유예 기간을 극대화합니다. 동시에 Karpenter는 가능한 한 빨리 준비할 수 있도록 새 노드를 즉시 시작합니다.
- 인스턴스 유형 선택을 지나치게 제한하지 마세요. 인스턴스 유형을 최대한 제한해서는 안 됩니다. 인스턴스 유형을 제한하지 않으면 더 낮은 비용으로 스팟 인스턴스 중단 빈도를 낮추면서 대규모로 스팟 용량을 확보할 가능성이 더 높습니다. 예를 들어 특정 유형(예: g5.xlarge)으로 제한하지 마세요. karpenter.k8s.aws/instance-category 및 karpenter.k8s.aws/instance-generation과 같은 키를 사용하여 다양한 인스턴스 범주 및 세대 집합을 지정하는 것이 좋습니다. Karpenter를 사용하면 여러 인스턴스 유형 및 가용 영역(AZs. 또한 AI/ML 워크로드에 특정 또는 제한된 수의 액셀러레이터가 필요하지만 리전 간에 유연한 경우 스팟 배치 점수를 사용하여 시작 전에 워크로드를 배포할 최적의 리전을 동적으로 식별할 수 있습니다.
- 더 많은 수의 유사한 EC2 인스턴스 패밀리를 포함하도록 NodePool 요구 사항을 확장합니다. 모든 스팟 인스턴스 풀은 특정 가용 영역(AZ)의 특정 인스턴스 유형에 대한 미사용 EC2 인스턴스 용량으로 구성됩니다. Karpenter는 새 노드를 프로비저닝하려고 할 때 NodePool의 요구 사항과 일치하는 인스턴스 유형을 선택합니다. 호환되는 인스턴스 유형이 AZ에서 스팟 용량을 가지고 있지 않으면 프로비저닝이 실패합니다. 이 문제를 방지하려면 GPU 메모리 또는 Ray Tracing과 같은 하드웨어 요구 사항을 고려하면서 크기 및 가용 영역(AZs)를 허용하세요. 인스턴스는 유형이 다를 수 있으므로 워크로드가 각 유형에서 실행될 수 있고 얻는 성능이 요구 사항을 충족하는지 확인해야 합니다.
- 리전의 모든 가용 영역을 활용합니다. 가용 용량은 가용 영역(AZ)에 따라 다르며, 특정 인스턴스 유형은 한 AZ에서는 사용할 수 없지만 다른 AZ에서는 풍부할 수 있습니다. 인스턴스 유형과 가용 영

역의 각 고유한 조합은 별도의 스팟 용량 풀을 구성합니다. Karpenter NodePool 요구 사항 내 리전의 모든 AZs에서 용량을 요청하면 한 번에 더 많은 풀을 효과적으로 검색할 수 있습니다. 이렇게 하면 스팟 용량 풀 수가 최대화되어 스팟 용량을 획득할 확률이 높아집니다. 이를 위해 NodePool 구성에서 topology.kubernetes.io/zone 키를 완전히 생략하여 Karpenter가 리전에서 사용 가능한 모든 AZs 중에서 선택할 수 있도록 하거나 연산자를 사용하여에 AZs를 명시적으로 나열하고 값(예: us-west-2a)을 제공합니다.

- 스팟 배치 점수(SPS)를 사용하여 스팟 인스턴스를 사용하여 필요한 용량에 성공적으로 액세스할 가능성을 파악할 수 있습니다. 스팟 배치 점수(SPS)는 스팟 요청이 성공할 가능성을 평가하는 데 도움이 되는 점수를 제공하는 도구입니다. SPS를 사용하는 경우 먼저 스팟 인스턴스에 대한 컴퓨팅 요구 사항을 지정한 다음 Amazon EC2는 스팟 요청이 성공할 가능성이 있는 상위 10개 리전 또는 가용 영역(AZs)을 반환합니다. 리전 및 가용 영역은 1~10의 척도로 점수가 매겨집니다. 10점은 스팟 요청이 성공할 가능성이 높지만 성공이 보장되지 않음을 나타냅니다. 1점은 스팟 요청이 성공할 가능성이 전혀 없음을 나타냅니다. 다른 리전 또는 가용 영역에 대해 동일한 점수가 반환될 수 있습니다. 자세한 내용은 AWS에서 스팟 배치 점수 추적기 대시보드 구축 지침을 참조하세요. 스팟 용량이 항상 변동함에 따라 SPS는 워크로드 제약 조건(예: 유연성, 성능, 크기 등)에 가장 적합한 인스턴스 유형, AZs 및 리전 조합을 식별하는 데 도움이 됩니다. AI/ML 워크로드에 특정 또는 제한된 수의 액셀러레이터가 필요하지만 리전 간에 유연한 경우 스팟 배치 점수를 사용하여 시작 전에 워크로드를 배포할 최적의 리전을 동적으로 식별할 수 있습니다. 스팟 용량을 확보할 가능성을 자동으로 파악할 수 있도록 SPS 트래커 대시보드 구축에 대한 지침을 제공합니다. 이 솔루션은 다양한 설정(예: GPUs)에 대한 YAML 구성을 사용하여 시간 경과에 따른 SPS 점수를 모니터링하고, CloudWatch에 지표를 저장하고, 구성을 비교하기 위한 대시보드를 제공합니다. 워크로드당 대시보드를 정의하여 vCPU, 메모리 및 GPU 요구 사항을 평가하여 다른 AWS 리전 사용을 고려하는 등 EKS 클러스터에 대한 최적의 설정을 보장합니다. 자세한 내용은 스팟 배치 점수 작동 방식을 참조하세요.
- 스팟 중단을 원활하게 처리하고 테스트합니다. 종료 기간이 2분보다 긴 포드의 경우 해당 포드가 다시 예약되기 전에 이전 노드가 중단되어 워크로드 가용성에 영향을 미칠 수 있습니다. 애플리케이션을 설계할 때 2분 스팟 중단 공지를 고려하세요. 장기 실행 애플리케이션에서 체크포인트 구현(예: 중단 후 재개하기 위해 Amazon S3)와 같은 영구 스토리지에 진행 상황 저장 Pod 사양에서 terminationGracePeriodSeconds(기본값은 30초)를 확장하여 정상적인 종료를 위해 더 많은 시간을 허용합니다. 정리와 같은 정상적인 종료 활동을 위해 애플리케이션 내에서 preStop 수명 주기 후크 및/또는 SIGTERM 신호를 사용하여 중단을 처리합니다. 상태 저장, 및 연결 종료. 규모 조정 시간이 중요하고 애플리케이션이 트래픽을 처리할 준비가 되는 데 2분 이상 걸리는 실시간 워크로드의 경우 the section called “스토리지” 및 the section called “성능” 모범 사례를 검토하여 컨테이너 시작 및 ML 모델 로드 시간을 최적화하는 것이 좋습니다. 대체 노드를 테스트하려면 AWS Fault Injection Service(FIS)를 사용하여 스팟 중단을 시뮬레이션합니다.

이러한 핵심 스팟 모범 사례 외에도 Amazon EKS에서 GPU 워크로드를 관리할 때 이러한 요소를 고려하세요. CPU 기반 워크로드와 달리 GPU 워크로드는 GPU 기능 및 사용 가능한 GPU 메모리와 같은 하드웨어 세부 정보에 특히 민감합니다. GPU 워크로드는 CPU에 비해 사용 가능한 옵션이 적어 사용할 수 있는 인스턴스 유형에 따라 제한될 수 CPUs. 첫 번째 단계로 워크로드가 인스턴스에 유연한지 평가합니다. 워크로드에서 사용할 수 있는 인스턴스 유형 수를 모르는 경우 개별적으로 테스트하여 호환성과 기능을 확인합니다. 다각화가 워크로드를 계속 작동하도록 하고 성능 영향(예: 처리량 또는 완료 시간)을 이해하면서 최대한 유연하게 다각화할 수 있는 방법을 식별합니다. 워크로드 다각화의 일환으로 다음 사항을 고려하세요.

- CUDA 및 프레임워크 호환성을 검토합니다. GPU 워크로드는 특정 하드웨어, GPU 유형(예: p3의 V100과 p4의 A100)에 최적화되거나 TensorFlow와 같은 라이브러리의 특정 CUDA 버전에 맞게 작성될 수 있으므로 워크로드의 호환성을 검토해야 합니다. 이러한 호환성은 런타임 오류, 충돌, GPU 가속화 장애(예: PyTorch 또는 TensorFlow와 같은 프레임워크와 일치하지 않는 CUDA 버전이 실행을 방해할 수 있음) 또는 FP16/INT8 정밀도와 같은 하드웨어 기능을 활용하는 기능을 방지하는 데 매우 중요합니다.
- GPU 메모리. DCGM Exporter와 같은 도구를 사용하여 모델의 메모리 요구 사항을 평가하고 런타임 중에 모델의 메모리 사용량을 프로파일링하고 karpenter.k8s.aws/instance-gpu-memory와 같은 잘 알려진 레이블에서 인스턴스 유형에 필요한 최소 GPU 메모리를 설정해야 합니다. GPU VRAM은 인스턴스 유형마다 다르며(예: NVIDIA T4는 16GB, A10G는 24GB, V100은 16-32GB), ML 모델(예: 대규모 언어 모델)은 사용 가능한 메모리를 초과하여 out-of-memory(OOM) 오류 또는 충돌을 일으킬 수 있습니다. EKS의 스팟 인스턴스의 경우 다각화를 제한할 수 있습니다. 예를 들어 모델이 맞지 않으면 하위 VRAM 유형을 포함할 수 없으므로 용량 풀에 대한 액세스가 제한되고 중단 위험이 증가할 수 있습니다. 단일 GPU, 단일 노드 추론(예: GPU 리소스를 활용하기 위해 동일한 노드에 예약된 여러 포드)의 경우 스팟 구성에 충분한 VRAM이 있는 인스턴스 유형만 포함할 수 있으므로 다각화가 제한될 수 있습니다.
- 부동 소수점 정밀도 및 성능. 모든 Nvidia GPU 아키텍처의 부동 소수점 정밀도가 동일한 것은 아닙니다(예: FP16/INT8). 워크로드에 필요한 코어 유형(CUDA/Tensor/RT) 성능과 부동 소수점 정밀도를 평가합니다. 가격이 저렴하고 성능이 낮은 GPU에서 실행하는 것이 더 낫다는 의미는 아니므로, 다각화의 영향을 이해하려면 특정 기간 내에 완료된 작업 측면에서 성능을 평가하는 것이 좋습니다.

시나리오: 실시간 추론 워크로드의 다각화

스팟 인스턴스의 실시간 온라인 추론 워크로드의 경우 호환되는 GPU 인스턴스 패밀리 및 세대를 다각화하도록 Karpenter NodePool을 구성할 수 있습니다. 이 접근 방식은 GPU 기능, 메모리 및 아키텍처에 대한 제약을 통해 성능을 유지하면서 여러 스팟 풀에서 끌어와 고가용성을 보장합니다. 인스턴스 용량이 제한될 때 대안을 사용하여 중단을 최소화하고 추론 지연 시간을 최적화할 수 있습니다. 이 예제 NodePool 상태는 GPU 메모리가 20GB 이상인 3보다 큰 g 및 p 시리즈 인스턴스를 사용합니다.

예제

```
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: gpu-inference-spot
spec:
  template:
    metadata:
      labels:
        role: gpu-spot-worker
    spec:
      requirements:
        - key: karpenter.sh/capacity-type
          operator: In
          values: ["spot"] # Use Spot Instances
        - key: karpenter.k8s.aws/instance-category
          operator: In
          values: ["g", "p"] # Diversifies across G-series and P-series
        - key: karpenter.k8s.aws/instance-generation
          operator: Gt
          values: ["3"] # Selects instance generations greater than 3
        - key: kubernetes.io/arch
          operator: In
          values: ["amd64"] # Specifies AMD64 architecture, compatible with NVIDIA GPUs
        - key: karpenter.k8s.aws/instance-gpu-memory
          operator: Gt
          values: ["20480"] # Ensures more than 20GB (20480 MiB) total GPU memory
      taints:
        - key: nvidia.com/gpu
          effect: NoSchedule
      nodeClassRef:
        name: gpu-inference-ec2
        group: karpenter.k8s.aws
        kind: EC2NodeClass
      expireAfter: 720h
  limits:
    cpu: 100
    memory: 100Gi
  disruption:
    consolidationPolicy: WhenEmptyOrUnderutilized
    consolidateAfter: 5m # Enables consolidation of underutilized nodes after 5 minutes
```

장기 실행 훈련 작업에 대한 체크포인트 구현

체크포인트는 프로세스 상태를 주기적으로 저장하여 중단 시 마지막으로 저장된 지점에서 재개할 수 있도록 하는 내결함성 기법입니다. 기계 학습에서는 일반적으로 장기 실행 작업이 모델 가중치와 옵티마이저 상태를 저장하여 하드웨어 문제 또는 스팟 인스턴스 중단과 같은 장애 후 훈련을 재개할 수 있는 훈련과 관련이 있습니다.

체크포인트를 사용하여 훈련 중에 기계 학습(ML) 모델의 상태를 저장합니다. 체크포인트는 모델의 스냅샷이며 ML 프레임워크의 콜백 함수로 구성할 수 있습니다. 저장된 체크포인트를 사용하여 마지막으로 저장된 체크포인트에서 훈련 작업을 다시 시작할 수 있습니다. 체크포인트를 사용하면 훈련 작업 또는 인스턴스가 예기치 않게 중단되어 모델 스냅샷을 훈련 중에 저장할 수 있습니다. 이렇게 하면 향후 체크포인트에서 모델 훈련을 재개할 수 있습니다. 노드 복원력 시스템을 구현하는 것 외에도 하드웨어 장애 또는 Amazon EC2 스팟 인스턴스 중단으로 인한 중단을 포함하여 중단의 영향을 완화하기 위해 체크포인트를 구현하는 것이 좋습니다.

체크포인트를 사용하지 않으면 중단으로 인해 컴퓨팅 시간이 낭비되고 진행 상황이 손실되어 장기 실행 훈련 작업에 비용이 많이 들 수 있습니다. 체크포인트를 사용하면 작업이 주기적으로 상태(예: 모델 가중치 및 최적화 프로그램 상태)를 저장하고 중단 후 마지막 체크포인트(마지막으로 처리된 배치)에서 재개할 수 있습니다. 체크포인트를 구현하려면 대규모 배치로 데이터를 처리하고 훈련 작업이 진행되는 동안 Mountpoint for Amazon S3 CSI 드라이버를 통해 Amazon S3 버킷과 같은 영구 스토리지에 중간 결과를 저장하도록 애플리케이션을 설계합니다. Amazon S3

사용 사례

체크포인트는 특정 시나리오에서 내결함성과 성능 오버헤드의 균형을 맞추는 데 특히 유용합니다. 다음과 같은 경우 체크포인트를 사용하는 것이 좋습니다.

- 작업 기간이 몇 시간을 초과함: 장기 실행 훈련 작업(예: 소규모 모델의 경우 >1~2시간 또는 수십억 개의 파라미터가 있는 대규모 파운데이션 모델의 경우 일/주)의 경우 중단으로 인한 진행 상황 손실에 비용이 많이 듭니다. 작업이 짧을수록 I/O 오버헤드가 정당화되지 않을 수 있습니다.
- 스팟 인스턴스 또는 하드웨어 장애의 경우: EC2 스팟(2분 알림) 또는 하드웨어 장애(예: GPU 메모리 오류)와 같이 중단되기 쉬운 환경에서 체크포인트를 사용하면 빠르게 재개할 수 있으므로 내결함성 워크로드에서 비용을 절감할 수 있습니다.
- 대규모 분산 훈련: 수백/수만 개의 액셀러레이터(예: >100 GPUs)가 있는 설정의 경우, 장애 사이의 평균 시간이 규모에 따라 선형적으로 감소합니다. 모델/데이터 병렬 처리를 위해를 사용하여 동시 체크포인트 액세스를 처리하고 완전한 재시작을 방지합니다.

- 리소스 요구가 높은 대규모 모델: 클러스터 크기로 인해 장애가 불가피한 페타바이트 규모의 LLM 훈련에서 계층형 접근 방식(임시 장애의 경우 5~30분마다 빠른 로컬, 주요 장애의 경우 시간당 내구성)은 복구 시간과 효율성을 최적화합니다.

ML 용량 블록을 사용하여 P 및 Trainium 인스턴스의 용량 보장

ML용 용량 블록을 사용하면 자주 찾는 GPU 인스턴스, 특히 P 인스턴스(예: p6-b200, p5, p5e, p5en, p4d, p4de) 및 Trainium 인스턴스(예: trn1, trn2)를 예약하여 거의 즉시 또는 향후 날짜에 시작하여 단기 기계 학습(ML) 워크로드를 지원할 수 있습니다. 이러한 예약은 모델 훈련 및 미세 조정과 같은 컴퓨팅 집약적 작업의 용량을 보장하는 데 적합합니다. EC2 용량 블록 요금은 예약 요금과 운영 체제 요금으로 구성됩니다. 요금에 대한 자세한 내용은 ML 요금에 대한 EC2 용량 블록을 참조하세요.

예측 가능한 용량 보장을 위해 Amazon EKS에서 AI/ML 워크로드용 GPUs를 예약하려면 단기 또는 온디맨드 용량 예약(ODCRs)을 위해 ML 용량 블록을 활용하여 범용 용량 보장을 수행하는 것이 좋습니다.

- ODCRs 사용하면 특정 가용 영역에서 EC2 인스턴스 용량(예: g5 또는 p5와 같은 GPU 인스턴스)을 일정 기간 동안 예약하여 수요가 많은 경우에도 가용성을 보장할 수 있습니다. ODCRs 장기 약정이 없지만 사용 여부에 관계없이 예약 용량에 대한 온디맨드 요금을 지불합니다. EKS에서 ODCRs은 Karpenter 및 관리형 노드 그룹과 같은 노드 유형에서 지원됩니다. Karpenter에서 ODCRs의 우선 순위를 지정하려면 `capacityReservationSelectorTerms` 필드를 사용하도록 NodeClass를 구성합니다. Karpenter NodePools 설명서를 참조하세요.
- 용량 블록은 GPU(예: p5, p4d) 또는 Trainium(trn1, trn2) 인스턴스에 대한 특수한 예약 메커니즘으로, 모델 훈련, 미세 조정 또는 실험과 같은 단기 ML 워크로드를 위해 설계되었습니다. 미래 날짜부터 정의된 기간(일반적으로 24시간~182일) 동안 예약된 시간에 대해서만 요금을 지불하고 용량을 예약합니다. 선불이며 용량 요구 사항에 대한 사전 계획이 필요하고 자동 크기 조정을 지원하지 않지만 지연 시간이 짧은 네트워킹을 위해 EC2 UltraClusters 공동 배치됩니다. 예약 기간에 대해서만 요금이 부과됩니다. 자세한 내용은 용량 블록 찾기 및 구매를 참조하거나 ML용 용량 블록으로 관리형 노드 그룹 생성의 지침에 따라 용량 블록으로 관리형 노드 그룹을 설정하여 시작하세요.

AWS Management Console을 통해 용량을 예약하고 ML 용량 블록을 사용하도록 노드를 구성합니다. 워크로드 일정에 따라 예약을 계획하고 스테이징 클러스터에서 테스트합니다. 자세한 내용은 용량 블록 설명서를 참조하세요.

G Amazon EC2 인스턴스에 대한 온디맨드, Amazon EC2 스팟 또는 온디맨드 용량 예약(ODCRs) 고려

G Amazon EC2 인스턴스의 경우 온디맨드, Amazon EC2 스팟 인스턴스 및 온디맨드 용량 예약과 같은 다양한 구매 옵션을 고려합니다. ODCRs 사용하면 특정 가용 영역에서 특정 기간 동안 EC2 인스턴스 용량을 예약하여 수요가 많은 경우에도 가용성을 보장할 수 있습니다. P 및 Trainium 인스턴스에서만 사용할 수 있는 ML 용량 블록과 달리 ODCRs은 G 인스턴스를 비롯한 다양한 인스턴스 유형에 사용할 수 있으므로 추론 또는 그래픽과 같은 다양한 GPU 기능이 필요한 워크로드에 적합합니다. Amazon EC2 스팟 인스턴스를 사용할 때는 다양한 인스턴스 유형, 크기 및 가용 영역에 걸쳐 다양한 기능을 사용할 수 있어야 스팟을 더 오래 유지할 수 있습니다.

ODCRs 장기 약정이 없지만 사용 여부에 관계없이 예약 용량에 대한 온디맨드 요금을 지불합니다. ODCRs 즉시 사용할 수 있도록 생성하거나 향후 날짜로 예약하여 용량 계획에 유연성을 제공할 수 있습니다. Amazon EKS에서 ODCRs은 Karpenter 및 관리형 노드 그룹과 같은 노드 유형에서 지원됩니다. Karpenter에서 ODCRs의 우선 순위를 지정하려면 `capacityReservationSelectorTerms` 필드를 사용하도록 NodeClass를 구성합니다. Karpenter NodePools 설명서를 참조하세요. CLI 명령을 포함한 ODCRs 생성에 대한 자세한 내용은 온디맨드 용량 예약 시작하기를 참조하세요.

기타 가속화된 인스턴스 유형 및 크기 고려

Amazon EKS의 ML 워크로드에서 성능과 비용을 모두 최적화하려면 적절한 가속화된 인스턴스와 크기를 선택해야 합니다. 예를 들어 GPU 인스턴스 패밀리마다 GPU 메모리와 같은 성능과 기능이 다릅니다. 가장 경제적인 옵션을 선택하는 데 도움이 되도록 가속 컴퓨팅의 EC2 인스턴스 유형 페이지에서 사용 가능한 GPU 인스턴스를 검토하세요. 여러 인스턴스 유형 및 크기를 평가하여 특정 워크로드 요구 사항에 가장 적합한 인스턴스를 찾습니다. GPUs 수, 메모리 및 네트워크 성능과 같은 요소를 고려합니다. 적절한 GPU 인스턴스 유형과 크기를 신중하게 선택하면 EKS 클러스터에서 리소스 사용률과 비용 효율성을 높일 수 있습니다.

EKS 노드에서 GPU 인스턴스를 사용하는 경우 기본적으로 네`kube-system`임스페이스에 `nvidia-device-plugin-daemonset` 포드가 있습니다. 인스턴스에서 GPU를 완전히 활용하는지 여부를 빠르게 파악하려면 다음과 같이 nvidia-smi(nvidia-smi)를 사용할 수 있습니다.

```
kubectl exec nvidia-device-plugin-daemonset-xxxxx \
  -n kube-system -- nvidia-smi \
  --query-
gpu=index,power.draw,power.limit,temperature.gpu,utilization.gpu,utilization.memory,memory.free
 \
  --format=csv -l 5
```

- `utilization.memory`가 100%에 가까우면 코드(들)가 메모리 경계일 가능성이 높습니다. 즉, GPU(메모리)가 완전히 활용되지만 추가 성능 최적화를 조사해야 한다고 제안할 수 있습니다.
- `utilization.gpu`가 100%에 가까우면 반드시 GPU가 완전히 사용된다는 의미는 아닙니다. 더 나은 지표는 대 `power.draw`의 비율입니다`power.limit`. 이 비율이 100% 이상이면 코드(들)가 GPU의 컴퓨팅 용량을 완전히 활용하는 것입니다.
- `-l 5` 플래그에는 5초마다 지표를 출력하라는 메시지가 표시됩니다. 단일 GPU 인스턴스 유형의 경우 인덱스 쿼리 플래그가 필요하지 않습니다.

자세한 내용은 AWS 설명서의 GPU 인스턴스를 참조하세요.

시간 조각화, MIG 및 분수 GPU 할당을 사용하여 GPU 리소스 할당 최적화

Kubernetes의 정적 리소스 제한(예: CPU, 메모리, GPU 수)은 특히 추론과 같은 동적 AI/ML 워크로드의 경우 과다 프로비저닝 또는 활용도 저하를 초래할 수 있습니다. 올바른 GPU를 선택하는 것이 중요합니다. 소량 또는 급증하는 워크로드의 경우 시간 분할을 사용하면 컴퓨팅 리소스를 공유하여 여러 워크로드가 단일 GPU를 공유할 수 있으므로 효율성을 개선하고 낭비를 줄일 수 있습니다. GPU 공유는 다양한 옵션을 통해 달성할 수 있습니다.

- 노드 선택기/노드 선호도를 활용하여 일정에 영향: 프로비저닝된 노드와 포드가 워크로드에 적합한 GPUs에 예약되어 있는지 확인합니다(예: `karpenter.k8s.aws/instance-gpu-name: "a100"`).
- 시간 조각화: 시간 경과에 따라 GPU의 컴퓨팅 리소스를 공유하도록 워크로드를 예약하므로 물리적 파티셔닝 없이 동시에 실행할 수 있습니다. 이는 컴퓨팅 요구가 가변적인 워크로드에 적합하지만 메모리 격리가 부족할 수 있습니다.
- 다중 인스턴스 GPU(MIG): MIG를 사용하면 단일 NVIDIA GPU를 격리된 여러 인스턴스로 분할할 수 있으며 NVIDIA Ampere(예: A100 GPU), NVIDIA Hopper(예: H100 GPU) 및 NVIDIA Blackwell(예: Blackwell GPUs) GPUs에서 지원됩니다. 각 MIG 인스턴스는 전용 컴퓨팅 및 메모리 리소스를 수신하여 다중 테넌트 환경 또는 리소스 보장이 필요한 워크로드에서 리소스를 공유할 수 있으므로 시간 분할을 통해 배치 크기가 다른 여러 모델을 제공하는 등의 시나리오를 포함하여 GPU 리소스 사용률을 최적화할 수 있습니다.
- 분수 GPU 할당: 소프트웨어 기반 예약을 사용하여 GPU 컴퓨팅 또는 메모리의 일부를 워크로드에 할당하여 동적 워크로드에 유연성을 제공합니다. Run:ai 플랫폼의 일부인 NVIDIA KAI 스케줄러를 사용하면 포드가 소수의 GPU 리소스를 요청할 수 있습니다.

EKS에서 이러한 기능을 활성화하려면 GPUs를 예약 가능한 리소스로 노출하고 시간 조각화 및 MIG를 지원하는 NVIDIA 디바이스 플러그인을 배포할 수 있습니다. 자세한 내용은 Kubernetes의 시간 조각화

GPUs 및 NVIDIA 시간 조각화 및 가속화된 EC2 인스턴스를 사용한 Amazon EKS의 GPU 공유를 참조하세요.

예제

예를 들어 NVIDIA 디바이스 플러그인을 사용하여 시간 조각화를 활성화하려면:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nvidia-device-plugin-config
  namespace: kube-system
data:
  config.yaml: |
    version: v1
    sharing:
      timeSlicing:
        resources:
        - name: nvidia.com/gpu
          replicas: 4  # Allow 4 pods to share each GPU
```

예제

예를 들어 분수 GPU 할당에 KAI 스케줄러를 사용하려면 NVIDIA GPU 연산자와 함께 배포하고 포드 사양에서 분수 GPU 리소스를 지정합니다.

```
apiVersion: v1
kind: Pod
metadata:
  name: fractional-gpu-pod-example
  annotations:
    gpu-fraction: "0.5"  # Annotation for 50% GPU
  labels:
    runai/queue: "default"  # Required queue assignment
spec:
  containers:
  - name: ml-workload
    image: nvcr.io/nvidia/pytorch:25.04-py3
    resources:
      limits:
        nvidia.com/gpu: 1
  nodeSelector:
```

```
    nvidia.com/gpu: "true"
  schedulerName: kai-scheduler
```

노드 복원력 및 훈련 작업 관리

자동 복구를 통한 노드 상태 확인 구현

여러 노드에서 다중 GPU 모델 훈련과 같이 노드 간 통신이 자주 필요한 Amazon EKS의 분산 훈련 작업의 경우 GPU 또는 EFA 장애와 같은 하드웨어 문제로 인해 훈련 작업이 중단될 수 있습니다. 이러한 중단으로 인해 특히 안정적인 하드웨어를 사용하는 장기 실행 AI/ML 워크로드의 경우 훈련 진행 상황이 손실되고 비용이 증가할 수 있습니다.

GPU 워크로드를 실행하는 EKS 클러스터의 GPU 장애와 같은 하드웨어 장애에 대한 복원력을 추가하려면 EKS 노드 모니터링 에이전트를 자동 복구 또는 Amazon SageMaker HyperPod와 함께 활용하는 것이 좋습니다. 자동 복구 기능이 있는 EKS 노드 모니터링 에이전트는 노드 상태 모니터링 및 표준 Kubernetes 메커니즘을 사용한 자동 복구와 같은 기능을 제공하지만 SageMaker HyperPod는 대상 복원력과 심층 상태 확인 및 자동 작업 재개와 같은 대규모 ML 훈련을 위해 특별히 설계된 추가 기능을 제공합니다.

- 노드 자동 복구가 포함된 EKS 노드 모니터링 에이전트는 GPU 또는 네트워킹 장애와 같은 문제를 식별하기 위해 가속화된 하드웨어와 관련된 표준 조건Ready과 같은 표준 조건을 포함하여 로그를 읽고 NodeConditions를 적용하여 노드 상태를 지속적으로 모니터링합니다. 노드가 비정상으로 간주되면 노드 자동 복구는 노드를 코딩하고 새 노드로 교체합니다. 포드 일정 변경 및 작업 재시작은 표준 Kubernetes 메커니즘과 작업의 재시작 정책에 따라 달라집니다.
- SageMaker HyperPod 심층 상태 확인 및 상태 모니터링 에이전트는 GPU 및 Trainium 기반 인스턴스의 상태를 지속적으로 모니터링합니다. 레이블(예: node-health-status)을 사용하여 노드 상태를 관리함으로써 AI/ML 워크로드에 맞게 조정됩니다. 노드가 비정상으로 간주되면 HyperPod는 GPUs. 기본적으로 기본 상태 확인을 통해 EFA의 네트워킹 관련 장애를 감지하고 중단된 훈련 작업에 대한 자동 재개를 지원하므로 마지막 체크포인트에서 작업을 계속할 수 있으므로 대규모 ML 작업의 중단이 최소화됩니다.

EFA를 사용하는 자동 복구 및 SageMaker HyperPod 클러스터가 있는 EKS 노드 모니터링 에이전트의 경우 원격 직접 메모리 액세스(RDMA) 오류 및 패킷 삭제와 같은 EFA별 지표를 모니터링하려면 AWS EFA 드라이버가 설치되어 있어야 합니다. 또한 CloudWatch Observability 추가 기능을 배포하거나 Prometheus 및 Grafana와 함께 DCGM Exporter와 같은 도구를 사용하여 EFA, GPU 및 SageMaker HyperPod의 경우 기능과 관련된 특정 지표를 모니터링하는 것이 좋습니다.

중단에 민감한 워크로드에 대해 Karpenter 통합 비활성화

처리, 대규모 AI/ML 예측 작업 또는 훈련과 같은 중단에 민감한 워크로드의 경우 작업 실행 중에 중단이 발생하지 않도록 Karpenter 통합 정책을 조정하는 것이 좋습니다. Karpenter의 통합 기능은 사용률이 낮은 노드를 종료하거나 저렴한 대체 노드로 대체하여 클러스터 비용을 자동으로 최적화합니다. 그러나 워크로드가 GPU를 완전히 활용하는 경우에도 Karpenter는 포드의 요구 사항을 충족하는 적절한 크기의 저렴한 인스턴스 유형을 식별하면 노드를 통합할 수 있으며, 이로 인해 작업이 중단될 수 있습니다.

`WhenEmptyOrUnderutilized` 통합 정책은 노드를 조기에 종료하여 실행 시간이 길어질 수 있습니다. 예를 들어, 중단으로 인해 포드 다시 예약, 데이터 다시 로드로 인해 작업 재개가 지연될 수 있으며, 이는 장기 실행 배치 추론 작업에 비용이 많이 들 수 있습니다. 이를 완화하기 위해를 `consolidationPolicy` 로 설정하고 워크로드 급증 중에 노드를 유지하도록 1시간과 같은 `consolidateAfter` 기간을 `WhenEmpty` 구성할 수 있습니다. 예:

```
disruption:
  consolidationPolicy: WhenEmpty
  consolidateAfter: 60m
```

이 접근 방식은 중단 비용이 컴퓨팅 비용 절감을 능가하는 실시간 온라인 추론 데이터 처리 또는 모델 훈련과 같이 급증하는 배치 추론 워크로드 및 기타 중단에 민감한 작업의 포드 시작 지연 시간을 개선합니다. Karpenter NodePool 중단 예산은 Karpenter 중단을 관리하는 또 다른 기능입니다. 예산을 사용하면 특정 시점에 선택한 NodePool에서 특정 수 이하의 노드 노드가 중단되지 않도록 할 수 있습니다. 또한 중단 예산을 사용하여 특정 시간(예: 피크 시간)에 모든 노드가 중단되는 것을 방지할 수 있습니다. 자세한 내용은 Karpenter 통합 설명서를 참조하세요.

ttlSecondsAfterFinished를 사용하여 Kubernetes 작업 자동 정리

완료된 작업 객체`ttlSecondsAfterFinished`를 자동으로 삭제하려면 Amazon EKS에서 Kubernetes 작업을 설정하는 것이 좋습니다. 링거링 작업 객체는 API 서버 메모리와 같은 클러스터 리소스를 사용하고 대시보드(예: Grafana, Amazon CloudWatch)를 복잡하게 만들어 모니터링을 복잡하게 만듭니다. 예를 들어 TTL을 1시간으로 설정하면 완료 직후 작업이 제거되어 클러스터가 정돈됩니다. 자세한 내용은 완료된 작업에 대한 자동 정리를 참조하세요.

우선 순위가 높은 작업/워크로드에 대해 우선 순위가 낮은 작업 선점 구성

Amazon EKS의 혼합 우선 순위 AI/ML 워크로드의 경우 우선 순위가 낮은 작업 선점으로 구성하여 우선 순위가 높은 작업(예: 실시간 추론)이 리소스를 즉시 수신하도록 할 수 있습니다. 선점 없이 배치 프로세스(예: 배치 추론, 데이터 처리), 비배치 서비스(예: 백그라운드 작업, cron 작업) 또는 CPU/메모

리 집약적 작업(예: 웹 서비스)과 같은 우선 순위가 낮은 워크로드는 노드를 점유하여 중요한 포드를 지연시킬 수 있습니다. 우선 순위가 높은 포드에 리소스가 필요한 경우 Kubernetes는 우선 순위가 낮은 포드를 제거하여 GPUs. CPUs 우선 순위를 할당하고 제거 동작을 제어`PriorityClass`하려면 Kubernetes`PodDisruptionBudget`를 사용하는 것이 좋습니다.

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: low-priority
value: 100
---
spec:
  priorityClassName: low-priority
```

자세한 내용은 Kubernetes Priority and Preemption 설명서를 참조하세요.

애플리케이션 조정 및 성능

Karpenter 또는 정적 노드를 사용하여 ML 워크로드에 맞게 컴퓨팅 용량 조정

Amazon EKS의 기계 학습(ML) 워크플로에 대한 비용 효율적이고 응답성이 뛰어난 컴퓨팅 용량을 보장하려면 워크로드의 특성 및 비용 약정에 맞게 노드 프로비저닝 전략을 조정하는 것이 좋습니다. 다음은 Karpenter를 사용한 just-in-time 조정과 예약 용량에 대한 정적 노드 그룹이라는 두 가지 고려 사항입니다.

- Karpenter와 같은 Just-in-time 데이터 영역 스케일러: 컴퓨팅 요구가 가변적인 동적 ML 워크플로(예: GPU 기반 추론 후 CPU 기반 플로팅)의 경우 Karpenter와 같은 just-in-time 데이터 영역 스케일러를 사용하는 것이 좋습니다.
- 예측 가능한 워크로드에 정적 노드 그룹 사용: 예측 가능한 정상 상태 ML 워크로드의 경우 또는 예약 인스턴스를 사용할 때 EKS 관리형 노드 그룹은 예약 용량이 완전히 프로비저닝되고 활용되도록 하여 절감 효과를 극대화할 수 있습니다. 이 접근 방식은 RIs 또는 ODCRs을 통해 커밋된 특정 인스턴스 유형에 적합합니다.

예제

다음은 인스턴스 생성이 3보다 큰 g Amazon EC2 인스턴스를 시작할 수 있는 다양한 Karpenter NodePool의 예입니다.

```
apiVersion: karpenter.sh/v1
```

```
kind: NodePool
metadata:
  name: gpu-inference
spec:
  template:
    spec:
      nodeClassRef:
        group: karpenter.k8s.aws
        kind: EC2NodeClass
        name: default
      requirements:
        - key: karpenter.sh/capacity-type
          operator: In
          values: ["on-demand"]
        - key: karpenter.k8s.aws/instance-category
          operator: In
          values: ["g"]
        - key: karpenter.k8s.aws/instance-generation
          operator: Gt
          values: ["3"]
        - key: kubernetes.io/arch
          operator: In
          values: ["amd64"]
      taints:
        - key: nvidia.com/gpu
          effect: NoSchedule
  limits:
    cpu: "1000"
    memory: "4000Gi"
    nvidia.com/gpu: "10"  *# Limit the total number of GPUs to 10 for the NodePool*
  disruption:
    consolidationPolicy: WhenEmpty
    consolidateAfter: 60m
    expireAfter: 720h
```

예제

훈련 워크로드에 정적 노드 그룹을 사용하는 예:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: ml-cluster
  region: us-west-2
```

```
managedNodeGroups:
  - name: gpu-node-group
    instanceType: p4d.24xlarge
    minSize: 2
    maxSize: 2
    desiredCapacity: 2
    taints:
      - key: nvidia.com/gpu
        effect: NoSchedule
```

테인트 및 허용치를 사용하여 가속화되지 않은 워크로드가 가속화된 인스턴스에서 예약되지 않도록 방지

GPU 리소스에서 가속화되지 않은 워크로드를 예약하는 것은 컴퓨팅 효율적이지 않으므로 테인트와 허용 오차를 사용하여 가속화되지 않은 워크로드 포드가 부적절한 노드에 예약되지 않도록 하는 것이 좋습니다. 자세한 내용은 Kubernetes 설명서를 참조하세요.

모델 성능에 따라 규모 조정

추론 워크로드의 경우 Kubernetes Event-Driven Autoscaling(KEDA)을 사용하여 적절한 휴지 기간에 추론 요청 또는 토큰 처리량과 같은 모델 성능 지표를 기반으로 확장하는 것이 좋습니다. 정적 조정 정책은 리소스를 과다 프로비저닝하거나 과소 프로비저닝하여 비용과 지연 시간에 영향을 미칠 수 있습니다. KEDA 설명서에서 자세히 알아보세요.

고급 GPU 관리를 위한 동적 리소스 할당

동적 리소스 할당(DRA)은 Kubernetes GPU 리소스 관리의 근본적인 발전을 나타냅니다. DRA는 기존 디바이스 플러그인 제한 사항을 넘어 정교한 GPU 공유, 토폴로지 인식 및 노드 간 리소스 조정을 지원합니다. Amazon EKS 버전 1.33에서 사용할 수 있는 DRA는 다음을 제공하여 AI/ML 워크로드의 중요한 문제를 해결합니다.

- 세분화된 GPU 할당
- 다중 프로세스 서비스(MPS) 및 다중 인스턴스 GPU(MIG)와 같은 고급 공유 메커니즘
- NVIDIA GB200 UltraServers를 포함한 차세대 하드웨어 아키텍처 지원

기존 GPU 할당은 GPUs 불투명한 정수 리소스로 취급하여 사용률이 크게 떨어집니다(프로덕션 클러스터에서 종종 30~40%). 이는 소수 리소스만 필요한 경우에도 워크로드GPUs에 독점적으로 액세스할 수 있기 때문입니다. DRA는 Kubernetes 스케줄러에 하드웨어 특성 및 워크로드 요구 사항에 대한 완

전한 가시성을 제공하는 구조화된 선언적 할당을 도입하여이 모델을 변환합니다. 이를 통해 지능형 배치 결정과 효율적인 리소스 공유가 가능합니다.

NVIDIA 디바이스 플러그인 대신 DRA를 사용할 때의 이점

NVIDIA 디바이스 플러그인(버전 부터 시작`0.12.0`)은 시간 조각화, MPS 및 MIG를 포함한 GPU 공유 메커니즘을 지원합니다. 그러나 DRA에서 다루는 아키텍처 제한 사항이 있습니다.

NVIDIA 디바이스 플러그인 제한 사항

- 정적 구성: GPU 공유 구성(시간 분할 복제본 및 MPS 설정)에는를 통해 클러스터 전체에서 사전 구성이 필요합니다`ConfigMaps`. 따라서 워크로드마다 다른 공유 전략을 제공하기가 어렵습니다.
- 제한적인 세분화 선택: 디바이스 플러그인은 노드 레이블을 통해 GPU 특성을 노출하지만 워크로드는 일정 결정의 일부로 특정 GPU 구성(메모리 크기 및 컴퓨팅 기능)을 동적으로 요청할 수 없습니다.
- 노드 간 리소스 조정 없음: 여러 노드에서 분산 GPU 리소스를 관리하거나 NVIDIA GB200과 같은 시스템의 NVLink 도메인과 같은 복잡한 토폴로지 요구 사항을 표현할 수 없습니다.
- 스케줄러 제약: Kubernetes 스케줄러는 GPU 리소스를 불투명한 정수로 취급하여 토폴로지 인식 결정을 내리거나 복잡한 리소스 종속성을 처리하는 기능을 제한합니다.
- 구성 복잡성: 서로 다른 공유 전략을 설정하려면 여러 개의 신중`ConfigMaps`한 노드 레이블 지정이 필요하므로 운영 복잡성이 발생합니다.

DRA를 사용한 솔루션

- 동적 리소스 선택: DRA를 사용하면 워크로드가를 통해 요청 시간에 세부 요구 사항(GPU 메모리, 드라이버 버전 및 특정 속성)을 지정할 수 있습니다`resourceclaims`. 이를 통해 보다 유연한 리소스 매칭이 가능합니다.
- 토폴로지 인식: DRA는 구조화된 파라미터 및 디바이스 선택기를 통해 노드 간 GPU 통신 및 메모리 일관성 상호 연결과 같은 복잡한 요구 사항을 처리합니다.
- 노드 간 리소스 관리: IMEX 채널을 `computeDomains` 통해 GB200과 같은 시스템에 중요한 여러 노드에서 분산 GPU 리소스를 조정할 수 있습니다.
- 워크로드별 구성: 각는 서로 다른 공유 전략 및 구성을 `ResourceClaim` 지정하므로 클러스터 전체 설정이 아닌 워크로드별로 세분화된 제어가 가능합니다.
- 향상된 스케줄러 통합: DRA는 스케줄러에 자세한 디바이스 정보를 제공하고 하드웨어 토폴로지 및 리소스 특성에 따라 보다 지능적인 배치 결정을 가능하게 합니다.

중요: DRA는 NVIDIA 디바이스 플러그인을 완전히 대체하지 않습니다. NVIDIA DRA 드라이버는 디바이스 플러그인과 함께 작동하여 향상된 기능을 제공합니다. 디바이스 플러그인은 기본 GPU 검색 및 관리를 계속 처리하는 반면, DRA는 고급 할당 및 예약 기능을 추가합니다.

DRA에서 지원하는 인스턴스 및 해당 기능

DRA 지원은 다음 표와 같이 Amazon EC2 인스턴스 패밀리 및 GPU 아키텍처에 따라 다릅니다.

인스턴스 패밀리	GPU 유형	시간 조각화	MIG 지원	MPS 지원	IMEX 지원	사용 사례
G5	NVIDIA A10G	예	아니요	예	아니요	추론 및 그래픽 워크로드
G6	NVIDIA L4	예	아니요	예	아니요	AI 추론 및 비디오 처리
G6e	NVIDIA L40S	예	아니요	예	아니요	훈련, 추론 및 그래픽
P4d/P4de	NVIDIA A100	예	예	예	아니요	대규모 훈련 및 HPC
P5	NVIDIA H100	예	예	예	아니요	파운데이션 모델 훈련
P6	NVIDIA B200	예	예	예	아니요	10억 또는 1조 파라미터 모델, 분산 훈련 및 추론
P6e	NVIDIA GB200	예	예	예	예	10억 또는 1조 파라미터 모델, 분산 훈련 및 추론

다음은 테이블의 각 기능에 대한 설명입니다.

- 시간 조각화: 여러 워크로드가 시간 경과에 따라 GPU 컴퓨팅 리소스를 공유할 수 있습니다.
- 다중 인스턴스 GPU(MIG): 격리된 GPU 인스턴스를 생성하는 하드웨어 수준 파티셔닝입니다.

- 다중 프로세스 서비스(MPS): 단일 GPU에서 여러 CUDA 프로세스의 동시 실행을 활성화합니다.
- Internode Memory Exchange(IMEX): GB200 UltraServers의 노드 간 메모리 일관성 통신.

추가 리소스

Kubernetes DRA 및 NVIDIA DRA 드라이버에 대한 자세한 내용은 GitHub의 다음 리소스를 참조하세요.

- Kubernetes dynamic-resource-allocation
- DRA에 대한 Kubernetes 개선 제안
- GPUs용 NVIDIA DRA 드라이버
- NVIDIA DRA 예제 및 빠른 시작

고급 GPU 관리를 위한 동적 리소스 할당 설정

다음 주제에서는 고급 GPU 관리를 위해 동적 리소스 할당(DRA)을 설정하는 방법을 보여줍니다.

사전 조건

Amazon EKS에서 DRA를 구현하기 전에 환경이 다음 요구 사항을 충족하는지 확인합니다.

클러스터 구성

- 버전 `1.33` 이상을 실행하는 Amazon EKS 클러스터
- Amazon EKS 관리형 노드 그룹(DRA는 현재 Karpenter가 아닌 AL2023 및 Bottlerocket NVIDIA 최적화 AMIs 있는 관리형 노드 그룹에서만 지원됨)
- 적절한 인스턴스 유형이 있는 NVIDIA GPU 지원 작업자 노드

필수 구성 요소

- NVIDIA 디바이스 플러그인 버전 `0.17.1` 이상
- NVIDIA DRA 드라이버 버전 `25.3.0` 이상

1단계: eksctl을 사용하여 DRA 지원 노드 그룹으로 클러스터 생성

1. 라는 클러스터 구성 파일을 생성합니다`dra-eks-cluster.yaml`.

```
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: dra-eks-cluster
  region: us-west-2
  version: '1.33'

managedNodeGroups:
- name: gpu-dra-nodes
  amiFamily: AmazonLinux2023
  instanceType: g6.12xlarge
  desiredCapacity: 2
  minSize: 1
  maxSize: 3

  labels:
    node-type: "gpu-dra"
    nvidia.com/gpu.present: "true"

  taints:
  - key: nvidia.com/gpu
    value: "true"
    effect: NoSchedule
```

2. 클러스터를 생성합니다.

```
eksctl create cluster -f dra-eks-cluster.yaml
```

2단계: NVIDIA 디바이스 플러그인 배포

NVIDIA 디바이스 플러그인을 배포하여 기본 GPU 검색을 활성화합니다.

1. NVIDIA 디바이스 플러그인 Helm 리포지토리를 추가합니다.

```
helm repo add nvidia https://nvidia.github.io/k8s-device-plugin
helm repo update
```

2. 디바이스 플러그인에 대한 사용자 지정 값을 생성합니다.

```
cat <<EOF > nvidia-device-plugin-values.yaml
gfd:
  enabled: true
nfd:
  enabled: true
tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
EOF
```

3. NVIDIA 디바이스 플러그인을 설치합니다.

```
helm install nvidia-device-plugin nvidia/nvidia-device-plugin \
 --namespace nvidia-device-plugin \
 --create-namespace \
 --version v0.17.1 \
 --values nvidia-device-plugin-values.yaml
```

3단계: NVIDIA DRA 드라이버 Helm 차트 배포

1. DRA 드라이버에 대한 `dra-driver-values.yaml` 값 파일을 생성합니다.

```
---
nvidiaDriverRoot: /

gpuResourcesEnabledOverride: true

resources:
  gpus:
    enabled: true
  computeDomains:
    enabled: true  # Enable for GB200 IMEX support

controller:
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule

kubeletPlugin:
```

```
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: "nvidia.com/gpu.present"
            operator: In
            values: ["true"]
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
```

2. NVIDIA NGC Helm 리포지토리를 추가합니다.

```
helm repo add nvidia https://helm.ngc.nvidia.com/nvidia
helm repo update
```

3. NVIDIA DRA 드라이버를 설치합니다.

```
helm install nvidia-dra-driver nvidia/nvidia-dra-driver-gpu \
 --version="25.3.0-rc.2" \
 --namespace nvidia-dra-driver \
 --create-namespace \
 --values dra-driver-values.yaml
```

4단계: DRA 설치 확인

1. DRA API 리소스를 사용할 수 있는지 확인합니다.

```
kubectl api-resources | grep resource.k8s.io/v1beta1
```

예상 출력은 다음과 같습니다.

```
deviceclasses resource.k8s.io/v1beta1 false DeviceClass
resourceclaims resource.k8s.io/v1beta1 true ResourceClaim
resourceclaimtemplates resource.k8s.io/v1beta1 true ResourceClaimTemplate
resourceslices resource.k8s.io/v1beta1 false ResourceSlice
```

2. 사용 가능한 디바이스 클래스를 확인합니다.

```
kubectl get deviceclasses
```

다음은 예상 출력의 예입니다.

```
NAME                                        AGE
compute-domain-daemon.nvidia.com            4h39m
compute-domain-default-channel.nvidia.com   4h39m
gpu.nvidia.com                              4h39m
mig.nvidia.com                              4h39m
```

새로 생성된 G6 GPU 인스턴스가 DRA가 활성화된 Amazon EKS 클러스터에 조인하면 다음 작업이 발생합니다.

- NVIDIA DRA 드라이버는 A10G GPU를 자동으로 검색하고 해당 노드`resourceslices`에 2개를 생성합니다.
- `gpu.nvidia.com` 조각은 사양(메모리, 컴퓨팅 기능 등)에 물리적 A10G GPU 디바이스를 등록합니다.
- A10G는 MIG 파티셔닝을 지원하지 않으므로 `compute-domain.nvidia.com` 조각은 GPU의 전체 컴퓨팅 컨텍스트를 나타내는 단일 컴퓨팅 도메인을 생성합니다.
- 그런 `resourceslices` 다음 Kubernetes API 서버에 게시되어를 통해 GPU 리소스를 예약할 수 있습니다`resourceclaims`.

  이제 DRA 스케줄러는를 통해 GPU 리소스를 요청하는 포드에이 GPU를 지능적으로 할당하여 기존 디바이스 플러그인 접근 방식에 비해 더 유연한 리소스 관리를 `resourceclaimtemplates`제공할 수 있습니다. 이는 수동 개입 없이 자동으로 발생합니다. DRA 드라이버가 리소스 검색 및 등록 프로세스를 완료하면 노드를 GPU 워크로드에 사용할 수 있게 됩니다.

  다음 명령을 실행할 때:

  ```
  kubectl get resourceslices
  ```

  다음은 예상 출력의 예입니다.

  ```
  NAME                                                        NODE
              DRIVER                          POOL                                AGE
  ```

```
ip-100-64-129-47.ec2.internal-compute-domain.nvidia.com-rwsts
 ip-100-64-129-47.ec2.internal    compute-domain.nvidia.com
 ip-100-64-129-47.ec2.internal    35m
ip-100-64-129-47.ec2.internal-gpu.nvidia.com-6kndg
 ip-100-64-129-47.ec2.internal    gpu.nvidia.com
 ip-100-64-129-47.ec2.internal    35m
```

계속해서 the section called “동적 리소스 할당을 사용하여 간단한 GPU 워크로드 예약”로 이동하세요.

동적 리소스 할당을 사용하여 간단한 GPU 워크로드 예약

동적 리소스 할당(DRA)을 사용하여 간단한 GPU 워크로드를 예약하려면 다음 단계를 수행합니다. 계속하기 전에를 따랐는지 확인합니다the section called “고급 GPU 관리를 위한 동적 리소스 할당 설정”.

1. `basic-gpu-claim-template.yaml`이름이 인 파일을 사용하여 GPU 할당에 `ResourceClaimTemplate` 대한 기본을 생성합니다.

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: gpu-test1

---
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  namespace: gpu-test1
  name: single-gpu
spec:
  spec:
    devices:
      requests:
      - name: gpu
        deviceClassName: gpu.nvidia.com
```

2. 템플릿을 적용합니다.

```
kubectl apply -f basic-gpu-claim-template.yaml
```

3. 상태를 확인합니다.

```
kubectl get resourceclaimtemplates -n gpu-test1
```

다음은 예 출력입니다.

```
NAME         AGE
single-gpu   9m16s
```

4. 이름이 인 파일과 ResourceClaimTemplate 함께를 사용하는 포드를 생성합니다basic-gpu-pod.yaml.

```
---
apiVersion: v1
kind: Pod
metadata:
  namespace: gpu-test1
  name: gpu-pod
  labels:
    app: pod
spec:
  containers:
  - name: ctr0
    image: ubuntu:22.04
    command: ["bash", "-c"]
    args: ["nvidia-smi -L; trap 'exit 0' TERM; sleep 9999 & wait"]
    resources:
      claims:
      - name: gpu0
  resourceClaims:
  - name: gpu0
    resourceClaimTemplateName: single-gpu
  nodeSelector:
    NodeGroupType: gpu-dra
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: "nvidia.com/gpu"
    operator: "Exists"
    effect: "NoSchedule"
```

5. 포드를 적용하고 모니터링합니다.

```
kubectl apply -f basic-gpu-pod.yaml
```

6. 포드 상태를 확인합니다.

```
kubectl get pod -n gpu-test1
```

다음은 예상 출력의 예입니다.

```
NAME      READY   STATUS    RESTARTS   AGE
gpu-pod   1/1     Running   0          13m
```

7. `ResourceClaim` 상태를 확인합니다.

```
kubectl get resourceclaims -n gpu-test1
```

다음은 예상 출력의 예입니다.

```
NAME                  STATE                AGE
gpu-pod-gpu0-l76cg    allocated,reserved   9m6s
```

8. 포드 로그를 보고 GPU 정보를 확인합니다.

```
kubectl logs gpu-pod -n gpu-test1
```

다음은 예상 출력의 예입니다.

```
GPU 0: NVIDIA L4 (UUID: GPU-da7c24d7-c7e3-ed3b-418c-bcecc32af7c5)
```

DRA를 사용한 고급 GPU 최적화 기술을 the section called “동적 리소스 할당을 통한 GPU 최적화 기법” 보려면를 계속 진행합니다.

동적 리소스 할당을 통한 GPU 최적화 기법

최신 GPU 워크로드는 최적의 사용률과 비용 효율성을 달성하려면 정교한 리소스 관리가 필요합니다. DRA를 사용하면 다양한 사용 사례와 하드웨어 기능을 해결하는 몇 가지 고급 최적화 기술을 사용할 수 있습니다.

- 시간 분할을 사용하면 여러 워크로드가 시간이 지남에 따라 GPU 컴퓨팅 리소스를 공유할 수 있으므로 산발적인 GPU 사용량이 있는 추론 워크로드에 적합합니다. 예제는 the section called “시간 조각화로 GPU 워크로드 최적화” 섹션을 참조하세요.

- 다중 프로세스 서비스(MPS)를 사용하면 시간 조각화보다 더 나은 격리를 통해 단일 GPU에서 여러 CUDA 프로세스를 동시에 실행할 수 있습니다. 예제는 the section called “MPS를 사용하여 GPU 워크로드 최적화” 섹션을 참조하세요.
- 다중 인스턴스 GPU(MIG)는 하드웨어 수준 파티셔닝을 제공하여 전용 컴퓨팅 및 메모리 리소스를 사용하여 격리된 GPU 인스턴스를 생성합니다. 예제는 the section called “다중 인스턴스 GPU를 사용하여 GPU 워크로드 최적화” 섹션을 참조하세요.
- Internode Memory Exchange(IMEX)를 사용하면 NVIDIA GB200 시스템에서 분산 훈련을 위해 노드 간 메모리 일관성 통신을 사용할 수 있습니다. 예제는 the section called “GB200 P6e 인스턴스를 사용하여 IMEX로 GPU 워크로드 최적화” 섹션을 참조하세요.

이러한 기술은 리소스 사용률을 크게 개선할 수 있습니다. 조직은 GPU 사용률이 기존 할당의 경우 30~40%에서 최적화된 공유 전략의 경우 80~90%로 증가한다고 보고합니다. 기술 선택은 워크로드 특성, 격리 요구 사항 및 하드웨어 기능에 따라 달라집니다.

시간 조각화로 GPU 워크로드 최적화

시간 분할을 사용하면 여러 워크로드가 동일한 물리적 GPU에서 순차적으로 실행되도록 예약하여 GPU 컴퓨팅 리소스를 공유할 수 있습니다. 산발적인 GPU 사용량이 있는 추론 워크로드에 적합합니다.

다음 단계를 수행합니다.

1. 라는 파일을 사용하여 시간 조각화를 `ResourceClaimTemplate` 위한을 정의합니다`timeslicing-claim-template.yaml`.

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: timeslicing-gpu

---
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: timeslicing-gpu-template
  namespace: timeslicing-gpu
spec:
  spec:
    devices:
```

```
      requests:
      - name: shared-gpu
        deviceClassName: gpu.nvidia.com
      config:
      - requests: ["shared-gpu"]
        opaque:
          driver: gpu.nvidia.com
          parameters:
            apiVersion: resource.nvidia.com/v1beta1
            kind: GpuConfig
            sharing:
              strategy: TimeSlicing
```

2. 라는 파일을 사용하여 시간 조각을 사용하여 포드를 정의합니다`timeslicing-pod.yaml`.

```
---
# Pod 1 - Inference workload
apiVersion: v1
kind: Pod
metadata:
  name: inference-pod-1
  namespace: timeslicing-gpu
  labels:
    app: gpu-inference
spec:
  restartPolicy: Never
  containers:
  - name: inference-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "-c"]
    args:
    - |
      import torch
      import time
      import os
      print(f"=== POD 1 STARTING ===")
      print(f"GPU available: {torch.cuda.is_available()}")
      print(f"GPU count: {torch.cuda.device_count()}")
      if torch.cuda.is_available():
          device = torch.cuda.current_device()
          print(f"Current GPU: {torch.cuda.get_device_name(device)}")
          print(f"GPU Memory:
 {torch.cuda.get_device_properties(device).total_memory / 1024**3:.1f} GB")
          # Simulate inference workload
```

```
          for i in range(20):
              x = torch.randn(1000, 1000).cuda()
              y = torch.mm(x, x.t())
              print(f"Pod 1 - Iteration {i+1} completed at {time.strftime('%H:%M:
%S')}")
              time.sleep(60)
      else:
          print("No GPU available!")
          time.sleep(5)
    resources:
      claims:
      - name: shared-gpu-claim
  resourceClaims:
  - name: shared-gpu-claim
    resourceClaimTemplateName: timeslicing-gpu-template
  nodeSelector:
    NodeGroupType: "gpu-dra"
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule


---
# Pod 2 - Training workload
apiVersion: v1
kind: Pod
metadata:
  name: training-pod-2
  namespace: timeslicing-gpu
  labels:
    app: gpu-training
spec:
  restartPolicy: Never
  containers:
  - name: training-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "-c"]
    args:
    - |
      import torch
      import time
      import os
```

```
      print(f"=== POD 2 STARTING ===")
      print(f"GPU available: {torch.cuda.is_available()}")
      print(f"GPU count: {torch.cuda.device_count()}")
      if torch.cuda.is_available():
          device = torch.cuda.current_device()
          print(f"Current GPU: {torch.cuda.get_device_name(device)}")
          print(f"GPU Memory:
{torch.cuda.get_device_properties(device).total_memory / 1024**3:.1f} GB")
          # Simulate training workload with heavier compute
          for i in range(15):
              x = torch.randn(2000, 2000).cuda()
              y = torch.mm(x, x.t())
              loss = torch.sum(y)
              print(f"Pod 2 - Training step {i+1}, Loss: {loss.item():.2f} at
{time.strftime('%H:%M:%S')}")
              time.sleep(5)
      else:
          print("No GPU available!")
          time.sleep(60)
    resources:
      claims:
      - name: shared-gpu-claim-2
  resourceClaims:
  - name: shared-gpu-claim-2
    resourceClaimTemplateName: timeslicing-gpu-template
  nodeSelector:
    NodeGroupType: "gpu-dra"
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
```

3. 템플릿과 포드를 적용합니다.

```
kubectl apply -f timeslicing-claim-template.yaml
kubectl apply -f timeslicing-pod.yaml
```

4. 리소스 클레임 모니터링:

```
kubectl get resourceclaims -n timeslicing-gpu -w
```

다음은 예 출력입니다.

```
NAME                                        STATE                AGE
inference-pod-1-shared-gpu-claim-9p97x      allocated,reserved   21s
training-pod-2-shared-gpu-claim-2-qghnb     pending              21s
inference-pod-1-shared-gpu-claim-9p97x      pending              105s
training-pod-2-shared-gpu-claim-2-qghnb     pending              105s
inference-pod-1-shared-gpu-claim-9p97x      pending              105s
training-pod-2-shared-gpu-claim-2-qghnb     allocated,reserved   105s
inference-pod-1-shared-gpu-claim-9p97x      pending              105s
```

첫 번째 포드(inference-pod-1)

- 상태: allocated,reserved
- 의미: DRA가 사용 가능한 GPU를 찾아이 포드용으로 예약했습니다.
- 포드 상태: 즉시 실행 시작

두 번째 포드(training-pod-2)

- 상태: pending
- 의미: DRA가 동일한 GPU에서 시간 분할을 구성할 때까지 대기
- 포드 상태: 예약 대기 중
- 상태는에서 pending 로 이동합니다allocated,reserved. running

MPS를 사용하여 GPU 워크로드 최적화

다중 프로세스 서비스(MPS)를 사용하면 시간 조각화보다 더 나은 격리를 통해 단일 GPU에서 여러 CUDA 컨텍스트를 동시에 실행할 수 있습니다.

다음 단계를 수행합니다.

1. 이름이 인 파일을 사용하여 MPSResourceClaimTemplate용을 정의합니다mps-claim-template.yaml.

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: mps-gpu
```

```
---
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: mps-gpu-template
  namespace: mps-gpu
spec:
  spec:
    devices:
      requests:
      - name: shared-gpu
        deviceClassName: gpu.nvidia.com
      config:
      - requests: ["shared-gpu"]
        opaque:
          driver: gpu.nvidia.com
          parameters:
            apiVersion: resource.nvidia.com/v1beta1
            kind: GpuConfig
            sharing:
              strategy: MPS
```

2. `mps-pod.yaml`이름이 인 파일을 사용하여 MPS를 사용하여 포드를 정의합니다.

```
---
# Single Pod with Multiple Containers sharing GPU via MPS
apiVersion: v1
kind: Pod
metadata:
  name: mps-multi-container-pod
  namespace: mps-gpu
  labels:
    app: mps-demo
spec:
  restartPolicy: Never
  containers:
  # Container 1 - Inference workload
  - name: inference-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "-c"]
    args:
    - |
      import torch
```

```
      import torch.nn as nn
      import time
      import os

      print(f"=== INFERENCE CONTAINER STARTING ===")
      print(f"Process ID: {os.getpid()}")
      print(f"GPU available: {torch.cuda.is_available()}")
      print(f"GPU count: {torch.cuda.device_count()}")

      if torch.cuda.is_available():
          device = torch.cuda.current_device()
          print(f"Current GPU: {torch.cuda.get_device_name(device)}")
          print(f"GPU Memory:
{torch.cuda.get_device_properties(device).total_memory / 1024**3:.1f} GB")

          # Create inference model
          model = nn.Sequential(
              nn.Linear(1000, 500),
              nn.ReLU(),
              nn.Linear(500, 100)
          ).cuda()

          # Run inference
          for i in range(1, 999999):
              with torch.no_grad():
                  x = torch.randn(128, 1000).cuda()
                  output = model(x)
                  result = torch.sum(output)
                  print(f"Inference Container PID {os.getpid()}: Batch {i}, Result:
{result.item():.2f} at {time.strftime('%H:%M:%S')}")
              time.sleep(2)
      else:
          print("No GPU available!")
          time.sleep(60)
    resources:
      claims:
      - name: shared-gpu-claim
        request: shared-gpu

  # Container 2 - Training workload
  - name: training-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "-c"]
    args:
```

```
    - |
      import torch
      import torch.nn as nn
      import time
      import os

      print(f"=== TRAINING CONTAINER STARTING ===")
      print(f"Process ID: {os.getpid()}")
      print(f"GPU available: {torch.cuda.is_available()}")
      print(f"GPU count: {torch.cuda.device_count()}")

      if torch.cuda.is_available():
          device = torch.cuda.current_device()
          print(f"Current GPU: {torch.cuda.get_device_name(device)}")
          print(f"GPU Memory:
{torch.cuda.get_device_properties(device).total_memory / 1024**3:.1f} GB")

          # Create training model
          model = nn.Sequential(
              nn.Linear(2000, 1000),
              nn.ReLU(),
              nn.Linear(1000, 500),
              nn.ReLU(),
              nn.Linear(500, 10)
          ).cuda()

          criterion = nn.MSELoss()
          optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

          # Run training
          for epoch in range(1, 999999):
              x = torch.randn(64, 2000).cuda()
              target = torch.randn(64, 10).cuda()

              optimizer.zero_grad()
              output = model(x)
              loss = criterion(output, target)
              loss.backward()
              optimizer.step()

              print(f"Training Container PID {os.getpid()}: Epoch {epoch}, Loss:
{loss.item():.4f} at {time.strftime('%H:%M:%S')}")
              time.sleep(3)
      else:
```

```
          print("No GPU available!")
          time.sleep(60)
    resources:
      claims:
      - name: shared-gpu-claim
        request: shared-gpu

  resourceClaims:
  - name: shared-gpu-claim
    resourceClaimTemplateName: mps-gpu-template

  nodeSelector:
    NodeGroupType: "gpu-dra"
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
```

3. 템플릿을 적용하고 여러 MPS 포드를 생성합니다.

```
kubectl apply -f mps-claim-template.yaml
kubectl apply -f mps-pod.yaml
```

4. 리소스 클레임을 모니터링합니다.

```
kubectl get resourceclaims -n mps-gpu -w
```

다음은 예 출력입니다.

```
NAME                                                STATE                AGE
mps-multi-container-pod-shared-gpu-claim-2p9kx   allocated,reserved   86s
```

이 구성은 동적 리소스 할당(DRA)을 통해 NVIDIA Multi-Process Service(MPS)를 사용한 실제 GPU 공유를 보여줍니다. GPU를 순차적으로 사용하여 워크로드가 회전하는 시간 조각화와 달리 MPS를 사용하면 두 컨테이너가 동일한 물리적 GPU에서 동시에 실행될 수 있습니다. 핵심 인사이트는 DRA MPS 공유에는 별도의 여러 포드가 아닌 단일 포드 내의 여러 컨테이너가 필요하다는 것입니다. 배포되면 DRA 드라이버는 포드ResourceClaim에 하나를 할당하고 추론 컨테이너와 훈련 컨테이너가 동시에 실행되도록 MPS를 자동으로 구성합니다.

각 컨테이너는 기본 하드웨어에 대한 액세스를 조정하는 MPS 데몬을 사용하여 자체 격리된 GPU 메모리 공간과 컴퓨팅 리소스를 가져옵니다. 다음을 수행하여 작동 중인지 확인할 수 있습니다.

- `nvidia-smi`를 확인하면 두 컨테이너가 모두 동일한 GPU 디바이스를 공유하는 M+C(`MPS + Compute`) 프로세스로 표시됩니다.
- 두 컨테이너의 로그를 모니터링하여 동시 실행을 입증하는 인터리브 타임스탬프를 표시합니다.

이 접근 방식은 보완 워크로드가 단일 프로세스로 활용도를 떨어뜨리지 않고 비용이 많이 드는 GPU 하드웨어를 효율적으로 공유할 수 있도록 하여 GPU 사용률을 극대화합니다.

Container1: **`inference-container`**

```
root@mps-multi-container-pod:/workspace# nvidia-smi
Wed Jul 16 21:09:30 2025
+-----------------------------------------------------------------------------------------
+
| NVIDIA-SMI 570.158.01             Driver Version: 570.158.01     CUDA Version: 12.9
   |
|-----------------------------------------+------------------------
+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr.
 ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute
 M. |
|                                         |                        |               MIG
 M. |
|=========================================+========================
+======================|
|   0  NVIDIA L4                      On  |   00000000:35:00.0 Off |
 0 |
| N/A   48C    P0             28W /   72W |     597MiB /  23034MiB |      0%   E.
 Process |
|                                         |                        |
 N/A |
+-----------------------------------------+------------------------
+----------------------+

+-----------------------------------------------------------------------------------------
+
| Processes:
   |
```

```
|  GPU   GI   CI              PID   Type   Process name                          GPU
 Memory |
|        ID   ID                                                               Usage
   |
|
=========================================================================================|
|    0   N/A  N/A               1    M+C   python
 246MiB |
+-----------------------------------------------------------------------------------------
+
```

Container2: **`training-container`**

```
root@mps-multi-container-pod:/workspace# nvidia-smi
Wed Jul 16 21:16:00 2025
+-----------------------------------------------------------------------------------------
+
| NVIDIA-SMI 570.158.01             Driver Version: 570.158.01     CUDA Version: 12.9
   |
|-----------------------------------------+------------------------
+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr.
 ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute
 M. |
|                                         |                        |               MIG
 M. |
|=========================================+========================
+======================|
|   0  NVIDIA L4                      On  |   00000000:35:00.0 Off |
 0 |
| N/A   51C    P0             28W /   72W |     597MiB /  23034MiB |      0%   E.
 Process |
|                                         |                        |
 N/A |
+-----------------------------------------+------------------------
+----------------------+

+-----------------------------------------------------------------------------------------
+
| Processes:
   |
```

```
|  GPU   GI   CI              PID   Type   Process name                            GPU
 Memory |
|        ID   ID                                                                   Usage
   |
|
=========================================================================================|
|    0   N/A  N/A               1    M+C   python
 314MiB |
+-----------------------------------------------------------------------------------------
+
```

다중 인스턴스 GPU를 사용하여 GPU 워크로드 최적화

다중 인스턴스 GPU(MIG)는 하드웨어 수준 파티셔닝을 제공하여 전용 컴퓨팅 및 메모리 리소스를 사용하여 격리된 GPU 인스턴스를 생성합니다.

다양한 프로필에서 동적 MIG 파티셔닝을 사용하려면 NVIDIA GPU 운영자가 필요합니다. NVIDIA GPU 운영자는 MIG Manager를 사용하여 MIG 프로파일을 생성하고 P4D, P4De, P5, P6 등과 같은 GPU 인스턴스를 재부팅하여 구성 변경 사항을 적용합니다. GPU 운영자에는 노드 레이블 변경을 감시하고 적절한 MIG 구성을 자동으로 적용하는 MIG Manager 구성 요소를 통한 포괄적인 MIG 관리 기능이 포함되어 있습니다. MIG 프로필 변경이 요청되면 연산자는 모든 GPU 클라이언트를 정상적으로 종료하고, 새 파티션 지오메트리를 적용하고, 영향을 받는 서비스를 다시 시작합니다. 이 프로세스를 수행하려면 GPU 인스턴스에 대한 노드 재부팅이 필요합니다. 따라서 성공적인 MIG 배포를 위해서는 MIG Manager 구성`WITH_REBOOT=true`에서를 활성화해야 합니다.

Amazon EKS에서 MIG를 사용하려면 NVIDIA DRA 드라이버와 NVIDIA GPU 운영자가 모두 필요합니다. NVIDIA 디바이스 플러그인 및 DCGM Exporter는 NVIDIA GPU Operator의 일부이므로이 외에도 필요하지 않습니다. EKS NVIDIA AMIs는 NVIDIA 드라이버가 사전 설치되어 제공되므로 충돌을 방지하고 인스턴스에 이미 있는 최적화된 드라이버를 활용하기 위해 GPU 운영자의 드라이버 배포를 비활성화했습니다. NVIDIA DRA 드라이버는 MIG 인스턴스에 대한 동적 리소스 할당을 처리하는 반면 GPU 운영자는 전체 GPU 수명 주기를 관리합니다. 여기에는 MIG 구성, 디바이스 플러그인 기능, DCGM을 통한 모니터링 및 노드 기능 검색이 포함됩니다. 이 통합 접근 방식은 하드웨어 수준 격리 및 동적 리소스 할당 기능을 갖춘 엔터프라이즈 GPU 관리를 위한 완전한 솔루션을 제공합니다.

1단계: NVIDIA GPU 운영자 배포

1. NVIDIA GPU Operator 리포지토리를 추가합니다.

```
helm repo add nvidia https://nvidia.github.io/gpu-operator
helm repo update
```

2. gpu-operator-values.yaml 파일 생성:

```
driver:
  enabled: false

mig:
  strategy: mixed

migManager:
  enabled: true
  env:
    - name: WITH_REBOOT
      value: "true"
  config:
    create: true
    name: custom-mig-parted-configs
    default: "all-disabled"
    data:
      config.yaml: |-
        version: v1
        mig-configs:
          all-disabled:
            - devices: all
              mig-enabled: false

          # P4D profiles (A100 40GB)
          p4d-half-balanced:
            - devices: [0, 1, 2, 3]
              mig-enabled: true
              mig-devices:
                "1g.5gb": 2
                "2g.10gb": 1
                "3g.20gb": 1
            - devices: [4, 5, 6, 7]
              mig-enabled: false

          # P4DE profiles (A100 80GB)
          p4de-half-balanced:
            - devices: [0, 1, 2, 3]
              mig-enabled: true
              mig-devices:
                "1g.10gb": 2
                "2g.20gb": 1
```

```
                "3g.40gb": 1
            - devices: [4, 5, 6, 7]
              mig-enabled: false

devicePlugin:
  enabled: true
  config:
    name: ""
    create: false
    default: ""

toolkit:
  enabled: true

nfd:
  enabled: true

gfd:
  enabled: true

dcgmExporter:
  enabled: true
  serviceMonitor:
    enabled: true
    interval: 15s
    honorLabels: false
    additionalLabels:
      release: kube-prometheus-stack

nodeStatusExporter:
  enabled: false

operator:
  defaultRuntime: containerd
  runtimeClass: nvidia
  resources:
    limits:
      cpu: 500m
      memory: 350Mi
    requests:
      cpu: 200m
      memory: 100Mi

daemonsets:
```

```
  tolerations:
    - key: "nvidia.com/gpu"
      operator: "Exists"
      effect: "NoSchedule"
  nodeSelector:
    accelerator: nvidia
  priorityClassName: system-node-critical
```

3. gpu-operator-values.yaml 파일을 사용하여 GPU 연산자를 설치합니다.

```
helm install gpu-operator nvidia/gpu-operator \
  --namespace gpu-operator \
  --create-namespace \
  --version v25.3.1 \
  --values gpu-operator-values.yaml
```

이 차트 Helm은 다음 구성 요소와 여러 MIG 프로파일을 배포합니다.

- 디바이스 플러그인(GPU 리소스 예약)
- DCGM Exporter(GPU 지표 및 모니터링)
- 노드 기능 검색(NFD - 하드웨어 레이블 지정)
- GPU 특성 검색(GFD - GPU별 레이블 지정)
- MIG Manager(다중 인스턴스 GPU 파티셔닝)
- Container Toolkit(GPU 컨테이너 런타임)
- 운영자 컨트롤러(수명 주기 관리)

4. 배포 포드를 확인합니다.

```
kubectl get pods -n gpu-operator
```

다음은 예 출력입니다.

```
NAME                                                          READY   STATUS
 RESTARTS        AGE
gpu-feature-discovery-27rdq                                   1/1     Running
 0               3h31m
gpu-operator-555774698d-48brn                                 1/1     Running
 0               4h8m
nvidia-container-toolkit-daemonset-sxmh9                      1/1     Running
 1 (3h32m ago)   4h1m
```

```
nvidia-cuda-validator-qb77g                                       0/1     Completed
 0               3h31m
nvidia-dcgm-exporter-cvzd7                                        1/1     Running
 0               3h31m
nvidia-device-plugin-daemonset-5ljm5                              1/1     Running
 0               3h31m
nvidia-gpu-operator-node-feature-discovery-gc-67f66fc557-q5wkt    1/1     Running
 0               4h8m
nvidia-gpu-operator-node-feature-discovery-master-5d8ffddcsl6s6   1/1     Running
 0               4h8m
nvidia-gpu-operator-node-feature-discovery-worker-6t4w7           1/1     Running
 1 (3h32m ago)   4h1m
nvidia-gpu-operator-node-feature-discovery-worker-9w7g8           1/1     Running
 0               4h8m
nvidia-gpu-operator-node-feature-discovery-worker-k5fgs           1/1     Running
 0               4h8m
nvidia-mig-manager-zvf54                                          1/1     Running
 1 (3h32m ago)   3h35m
```

5. MIG 예제를 테스트하기 위해 p4De 관리형 노드 그룹을 사용하여 Amazon EKS 클러스터를 생성합니다.

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: dra-eks-cluster
  region: us-east-1
  version: '1.33'

managedNodeGroups:
# P4DE MIG Node Group with Capacity Block Reservation
- name: p4de-mig-nodes
  amiFamily: AmazonLinux2023
  instanceType: p4de.24xlarge

  # Capacity settings
  desiredCapacity: 0
  minSize: 0
  maxSize: 1

  # Use specific subnet in us-east-1b for capacity reservation
  subnets:
```

```
    - us-east-1b

  # AL2023 NodeConfig for RAID0 local storage only
  nodeadmConfig:
    apiVersion: node.eks.aws/v1alpha1
    kind: NodeConfig
    spec:
      instance:
        localStorage:
          strategy: RAID0

  # Node labels for MIG configuration
  labels:
    nvidia.com/gpu.present: "true"
    nvidia.com/gpu.product: "A100-SXM4-80GB"
    nvidia.com/mig.config: "p4de-half-balanced"
    node-type: "p4de"
    vpc.amazonaws.com/efa.present: "true"
    accelerator: "nvidia"

  # Node taints
  taints:
    - key: nvidia.com/gpu
      value: "true"
      effect: NoSchedule

  # EFA support
  efaEnabled: true

  # Placement group for high-performance networking
  placementGroup:
    groupName: p4de-placement-group
    strategy: cluster

  # Capacity Block Reservation (CBR)
  # Ensure CBR ID matches the subnet AZ with the Nodegroup subnet
  spot: false
  capacityReservation:
    capacityReservationTarget:
      capacityReservationId: "cr-abcdefghij"  # Replace with your capacity
reservation ID
```

NVIDIA GPU 운영자는 노드에 추가된 레이블을 nvidia.com/mig.config: "p4de-half-balanced" 사용하고 지정된 프로필로 GPU를 분할합니다.

6. p4de 인스턴스에 로그인합니다.
7. 다음 명령을 실행합니다.

```
nvidia-smi -L
```

다음 예제 출력이 표시되어야 합니다.

```
[root@ip-100-64-173-145 bin]# nvidia-smi -L
GPU 0: NVIDIA A100-SXM4-80GB (UUID: GPU-ab52e33c-be48-38f2-119e-b62b9935925a)
  MIG 3g.40gb     Device  0: (UUID: MIG-da972af8-a20a-5f51-849f-bc0439f7970e)
  MIG 2g.20gb     Device  1: (UUID: MIG-7f9768b7-11a6-5de9-a8aa-e9c424400da4)
  MIG 1g.10gb     Device  2: (UUID: MIG-498adad6-6cf7-53af-9d1a-10cfd1fa53b2)
  MIG 1g.10gb     Device  3: (UUID: MIG-3f55ef65-1991-571a-ac50-0dbf50d80c5a)
GPU 1: NVIDIA A100-SXM4-80GB (UUID: GPU-0eabeccc-7498-c282-0ac7-d3c09f6af0c8)
  MIG 3g.40gb     Device  0: (UUID: MIG-80543849-ea3b-595b-b162-847568fe6e0e)
  MIG 2g.20gb     Device  1: (UUID: MIG-3af1958f-fac4-59f1-8477-9f8d08c55029)
  MIG 1g.10gb     Device  2: (UUID: MIG-401088d2-716f-527b-a970-b1fc7a4ac6b2)
  MIG 1g.10gb     Device  3: (UUID: MIG-8c56c75e-5141-501c-8f43-8cf22f422569)
GPU 2: NVIDIA A100-SXM4-80GB (UUID: GPU-1c7a1289-243f-7872-a35c-1d2d8af22dd0)
  MIG 3g.40gb     Device  0: (UUID: MIG-e9b44486-09fc-591a-b904-0d378caf2276)
  MIG 2g.20gb     Device  1: (UUID: MIG-ded93941-9f64-56a3-a9b1-a129c6edf6e4)
  MIG 1g.10gb     Device  2: (UUID: MIG-6c317d83-a078-5c25-9fa3-c8308b379aa1)
  MIG 1g.10gb     Device  3: (UUID: MIG-2b070d39-d4e9-5b11-bda6-e903372e3d08)
GPU 3: NVIDIA A100-SXM4-80GB (UUID: GPU-9a6250e2-5c59-10b7-2da8-b61d8a937233)
  MIG 3g.40gb     Device  0: (UUID: MIG-20e3cd87-7a57-5f1b-82e7-97b14ab1a5aa)
  MIG 2g.20gb     Device  1: (UUID: MIG-04430354-1575-5b42-95f4-bda6901f1ace)
  MIG 1g.10gb     Device  2: (UUID: MIG-d62ec8b6-e097-5e99-a60c-abf8eb906f91)
  MIG 1g.10gb     Device  3: (UUID: MIG-fce20069-2baa-5dd4-988a-cead08348ada)
GPU 4: NVIDIA A100-SXM4-80GB (UUID: GPU-5d09daf0-c2eb-75fd-3919-7ad8fafa5f86)
GPU 5: NVIDIA A100-SXM4-80GB (UUID: GPU-99194e04-ab2a-b519-4793-81cb2e8e9179)
GPU 6: NVIDIA A100-SXM4-80GB (UUID: GPU-c1a1910f-465a-e16f-5af1-c6aafe499cd6)
GPU 7: NVIDIA A100-SXM4-80GB (UUID: GPU-c2cfafbc-fd6e-2679-e955-2a9e09377f78)
```

NVIDIA GPU Operator가 P4DE 인스턴스에 p4de-half-balanced MIG 프로파일을 성공적으로 적용하여 구성된 하드웨어 수준 GPU 파티션을 생성했습니다. 파티셔닝 작동 방식은 다음과 같습니다.

GPU 운영자는 임베디드 MIG 프로파일에서이 구성을 적용했습니다.

```
p4de-half-balanced:
  - devices: [0, 1, 2, 3]        # First 4 GPUs: MIG enabled
    mig-enabled: true
    mig-devices:
      "1g.10gb": 2               # 2x small instances (10GB each)
      "2g.20gb": 1               # 1x medium instance (20GB)
      "3g.40gb": 1               # 1x large instance (40GB)
  - devices: [4, 5, 6, 7]        # Last 4 GPUs: Full GPUs
    mig-enabled: false
```

`nvidia-smi -L` 출력에서 GPU 운영자가 생성한 내용은 다음과 같습니다.

- MIG 지원 GPUs(0-3): 하드웨어 분할됨
  - GPU 0: NVIDIA A100-SXM4-80GB
    - MIG 3g.40gb 디바이스 0 - 대규모 워크로드(40GB 메모리, 42 SMs)
    - MIG 2g.20gb 디바이스 1 - 중간 워크로드(20GB 메모리, 28 SMs)
    - MIG 1g.10gb 디바이스 2 - 소규모 워크로드(10GB 메모리, 14 SMs)
    - MIG 1g.10gb 디바이스 3 - 소규모 워크로드(10GB 메모리, 14 SMs)
  - GPU 1: NVIDIA A100-SXM4-80GB
    - MIG 3g.40gb 디바이스 0 - 동일한 파티션 레이아웃
    - MIG 2g.20gb 디바이스 1
    - MIG 1g.10gb 디바이스 2
    - MIG 1g.10gb 디바이스 3
  - GPU 2 및 GPU 3 - GPU 0 및 GPU 1과 동일한 패턴
- 전체 GPUs(4~7): MIG 파티셔닝 없음
  - GPU 4: NVIDIA A100-SXM4-80GB – 전체 80GB GPU
  - GPU 5: NVIDIA A100-SXM4-80GB – 전체 80GB GPU
  - GPU 6: NVIDIA A100-SXM4-80GB – 전체 80GB GPU
  - GPU 7: NVIDIA A100-SXM4-80GB – 전체 80GB GPU

NVIDIA GPU 운영자가 MIG 파티션을 생성하면 NVIDIA DRA 드라이버는 이러한 하드웨어 격리 인스턴스를 자동으로 감지하여 Kubernetes에서 동적 리소스 할당에 사용할 수 있도록 합니다. DRA 드라이버는 특정 프로필(1g.10gb, 2g.20gb, 3g.40gb)이 있는 각 MIG 인스턴스를 검색하고 `mig.nvidia.com` 디바이스 클래스를 통해 예약 가능한 리소스로 노출합니다.

DRA 드라이버는 MIG 토폴로지를 지속적으로 모니터링하고 모든 GPUs에서 사용 가능한 인스턴스의 인벤토리를 유지합니다. 포드가를 통해 특정 MIG 프로파일을 요청하면 ResourceClaimTemplateDRA 드라이버는 사용 가능한 GPU에서 적절한 MIG 인스턴스를 지능적으로 선택하여 실제 하드웨어 수준 다중 테넌시를 활성화합니다. 이 동적 할당을 사용하면 엄격한 리소스 경계와 성능 보장을 유지하면서 동일한 물리적 GPU에서 여러 격리된 워크로드를 동시에 실행할 수 있습니다.

2단계: MIG 리소스 할당 테스트

이제 DRA가 다양한 워크로드에 MIG 인스턴스를 동적으로 할당하는 방법을 보여주는 몇 가지 예제를 실행해 보겠습니다. resourceclaimtemplates 및 테스트 포드를 배포하여 DRA 드라이버가 사용 가능한 MIG 파티션에 워크로드를 배치하는 방법을 확인하여 여러 컨테이너가 하드웨어 수준 격리로 GPU 리소스를 공유할 수 있도록 합니다.

1. MIGmig-claim-template.yaml를 포함하도록를 생성합니다resourceclaimtemplates.

```
apiVersion: v1
kind: Namespace
metadata:
  name: mig-gpu

---
# Template for 3g.40gb MIG instance (Large training)
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: mig-large-template
  namespace: mig-gpu
spec:
  spec:
    devices:
      requests:
      - name: mig-large
        deviceClassName: mig.nvidia.com
        selectors:
        - cel:
            expression: |
              device.attributes['gpu.nvidia.com'].profile == '3g.40gb'

---
# Template for 2g.20gb MIG instance (Medium training)
apiVersion: resource.k8s.io/v1beta1
```

```
kind: ResourceClaimTemplate
metadata:
  name: mig-medium-template
  namespace: mig-gpu
spec:
  spec:
    devices:
      requests:
      - name: mig-medium
        deviceClassName: mig.nvidia.com
        selectors:
        - cel:
            expression: |
              device.attributes['gpu.nvidia.com'].profile == '2g.20gb'

---
# Template for 1g.10gb MIG instance (Small inference)
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: mig-small-template
  namespace: mig-gpu
spec:
  spec:
    devices:
      requests:
      - name: mig-small
        deviceClassName: mig.nvidia.com
        selectors:
        - cel:
            expression: |
              device.attributes['gpu.nvidia.com'].profile == '1g.10gb'
```

2. 세 가지 템플릿을 적용합니다.

```
kubectl apply -f mig-claim-template.yaml
```

3. 다음 명령을 실행합니다.

```
kubectl get resourceclaimtemplates -n mig-gpu
```

다음은 예 출력입니다.

```
NAME                  AGE
mig-large-template    71m
mig-medium-template   71m
mig-small-template    71m
```

4. mig-pod.yaml를 생성하여이를 활용할 여러 작업을 예약resourceclaimtemplates합니다.

```
---
# ConfigMap containing Python scripts for MIG pods
apiVersion: v1
kind: ConfigMap
metadata:
  name: mig-scripts-configmap
  namespace: mig-gpu
data:
  large-training-script.py: |
    import torch
    import torch.nn as nn
    import torch.optim as optim
    import time
    import os

    print(f"=== LARGE TRAINING POD (3g.40gb) ===")
    print(f"Process ID: {os.getpid()}")
    print(f"GPU available: {torch.cuda.is_available()}")
    print(f"GPU count: {torch.cuda.device_count()}")

    if torch.cuda.is_available():
        device = torch.cuda.current_device()
        print(f"Using GPU: {torch.cuda.get_device_name(device)}")
        print(f"GPU Memory: {torch.cuda.get_device_properties(device).total_memory /
 1e9:.1f} GB")

        # Large model for 3g.40gb instance
        model = nn.Sequential(
            nn.Linear(2048, 1024),
            nn.ReLU(),
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
```

```
        ).cuda()

        optimizer = optim.Adam(model.parameters())
        criterion = nn.CrossEntropyLoss()

        print(f"Model parameters: {sum(p.numel() for p in model.parameters())}")

        # Training loop
        for epoch in range(100):
            # Large batch for 3g.40gb
            x = torch.randn(256, 2048).cuda()
            y = torch.randint(0, 10, (256,)).cuda()

            optimizer.zero_grad()
            output = model(x)
            loss = criterion(output, y)
            loss.backward()
            optimizer.step()

            if epoch % 10 == 0:
                print(f"Large Training - Epoch {epoch}, Loss: {loss.item():.4f}, GPU
Memory: {torch.cuda.memory_allocated()/1e9:.2f}GB")
            time.sleep(3)

        print("Large training completed on 3g.40gb MIG instance")

 medium-training-script.py: |
   import torch
   import torch.nn as nn
   import torch.optim as optim
   import time
   import os

   print(f"=== MEDIUM TRAINING POD (2g.20gb) ===")
   print(f"Process ID: {os.getpid()}")
   print(f"GPU available: {torch.cuda.is_available()}")
   print(f"GPU count: {torch.cuda.device_count()}")

   if torch.cuda.is_available():
       device = torch.cuda.current_device()
       print(f"Using GPU: {torch.cuda.get_device_name(device)}")
       print(f"GPU Memory: {torch.cuda.get_device_properties(device).total_memory /
1e9:.1f} GB")
```

```
        # Medium model for 2g.20gb instance
        model = nn.Sequential(
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
        ).cuda()

        optimizer = optim.Adam(model.parameters())
        criterion = nn.CrossEntropyLoss()

        print(f"Model parameters: {sum(p.numel() for p in model.parameters())}")

        # Training loop
        for epoch in range(100):
            # Medium batch for 2g.20gb
            x = torch.randn(128, 1024).cuda()
            y = torch.randint(0, 10, (128,)).cuda()

            optimizer.zero_grad()
            output = model(x)
            loss = criterion(output, y)
            loss.backward()
            optimizer.step()

            if epoch % 10 == 0:
                print(f"Medium Training - Epoch {epoch}, Loss: {loss.item():.4f}, GPU
Memory: {torch.cuda.memory_allocated()/1e9:.2f}GB")
            time.sleep(4)

        print("Medium training completed on 2g.20gb MIG instance")

 small-inference-script.py: |
   import torch
   import torch.nn as nn
   import time
   import os

   print(f"=== SMALL INFERENCE POD (1g.10gb) ===")
   print(f"Process ID: {os.getpid()}")
   print(f"GPU available: {torch.cuda.is_available()}")
   print(f"GPU count: {torch.cuda.device_count()}")
```

```
    if torch.cuda.is_available():
        device = torch.cuda.current_device()
        print(f"Using GPU: {torch.cuda.get_device_name(device)}")
        print(f"GPU Memory: {torch.cuda.get_device_properties(device).total_memory /
 1e9:.1f} GB")

        # Small model for 1g.10gb instance
        model = nn.Sequential(
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
        ).cuda()

        print(f"Model parameters: {sum(p.numel() for p in model.parameters())}")

        # Inference loop
        for i in range(200):
            with torch.no_grad():
                # Small batch for 1g.10gb
                x = torch.randn(32, 512).cuda()
                output = model(x)
                prediction = torch.argmax(output, dim=1)

                if i % 20 == 0:
                    print(f"Small Inference - Batch {i}, Predictions:
 {prediction[:5].tolist()}, GPU Memory: {torch.cuda.memory_allocated()/1e9:.2f}GB")
            time.sleep(2)

        print("Small inference completed on 1g.10gb MIG instance")

---
# Pod 1: Large training workload (3g.40gb)
apiVersion: v1
kind: Pod
metadata:
  name: mig-large-training-pod
  namespace: mig-gpu
  labels:
    app: mig-large-training
    workload-type: training
spec:
  restartPolicy: Never
  containers:
  - name: large-training-container
```

```
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "/scripts/large-training-script.py"]
    volumeMounts:
    - name: script-volume
      mountPath: /scripts
      readOnly: true
    resources:
      claims:
      - name: mig-large-claim
  resourceClaims:
  - name: mig-large-claim
    resourceClaimTemplateName: mig-large-template
  nodeSelector:
    node.kubernetes.io/instance-type: p4de.24xlarge
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
  volumes:
  - name: script-volume
    configMap:
      name: mig-scripts-configmap
      defaultMode: 0755

---
# Pod 2: Medium training workload (2g.20gb) - can run on SAME GPU as Pod 1
apiVersion: v1
kind: Pod
metadata:
  name: mig-medium-training-pod
  namespace: mig-gpu
  labels:
    app: mig-medium-training
    workload-type: training
spec:
  restartPolicy: Never
  containers:
  - name: medium-training-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "/scripts/medium-training-script.py"]
    volumeMounts:
    - name: script-volume
      mountPath: /scripts
```

```
      readOnly: true
    resources:
      claims:
      - name: mig-medium-claim
  resourceClaims:
  - name: mig-medium-claim
    resourceClaimTemplateName: mig-medium-template
  nodeSelector:
    node.kubernetes.io/instance-type: p4de.24xlarge
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
  volumes:
  - name: script-volume
    configMap:
      name: mig-scripts-configmap
      defaultMode: 0755

---
# Pod 3: Small inference workload (1g.10gb) - can run on SAME GPU as Pod 1 & 2
apiVersion: v1
kind: Pod
metadata:
  name: mig-small-inference-pod
  namespace: mig-gpu
  labels:
    app: mig-small-inference
    workload-type: inference
spec:
  restartPolicy: Never
  containers:
  - name: small-inference-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "/scripts/small-inference-script.py"]
    volumeMounts:
    - name: script-volume
      mountPath: /scripts
      readOnly: true
    resources:
      claims:
      - name: mig-small-claim
  resourceClaims:
```

```
  - name: mig-small-claim
    resourceClaimTemplateName: mig-small-template
  nodeSelector:
    node.kubernetes.io/instance-type: p4de.24xlarge
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
  volumes:
  - name: script-volume
    configMap:
      name: mig-scripts-configmap
      defaultMode: 0755
```

5. 세 개의 포드를 배포해야 하는이 사양을 적용합니다.

```
kubctl apply -f mig-pod.yaml
```

이러한 포드는 DRA 드라이버에서 예약해야 합니다.

6. DRA 드라이버 포드 로그를 확인하면 다음과 비슷한 출력이 표시됩니다.

```
I0717 21:50:22.925811 1 driver.go:87] NodePrepareResource is called: number of
 claims: 1
I0717 21:50:22.932499 1 driver.go:129] Returning newly prepared devices
 for claim '933e9c72-6fd6-49c5-933c-a896407dc6d1': [&Device{RequestNames:
[mig-large],PoolName:ip-100-64-173-145.ec2.internal,DeviceName:gpu-0-
mig-9-4-4,CDIDeviceIDs:[k8s.gpu.nvidia.com/device=**gpu-0-mig-9-4-4**],}]
I0717 21:50:23.186472 1 driver.go:87] NodePrepareResource is called: number of
 claims: 1
I0717 21:50:23.191226 1 driver.go:129] Returning newly prepared devices
 for claim '61e5ddd2-8c2e-4c19-93ae-d317fecb44a4': [&Device{RequestNames:
[mig-medium],PoolName:ip-100-64-173-145.ec2.internal,DeviceName:gpu-2-
mig-14-0-2,CDIDeviceIDs:[k8s.gpu.nvidia.com/device=**gpu-2-mig-14-0-2**],}]
I0717 21:50:23.450024 1 driver.go:87] NodePrepareResource is called: number of
 claims: 1
I0717 21:50:23.455991 1 driver.go:129] Returning newly prepared devices
 for claim '1eda9b2c-2ea6-401e-96d0-90e9b3c111b5': [&Device{RequestNames:
[mig-small],PoolName:ip-100-64-173-145.ec2.internal,DeviceName:gpu-1-
mig-19-2-1,CDIDeviceIDs:[k8s.gpu.nvidia.com/device=**gpu-1-mig-19-2-1**],}]
```

7. `resourceclaims`를 확인하여 포드 상태를 확인합니다.

```
kubectl get resourceclaims -n mig-gpu -w
```

다음은 예 출력입니다.

```
NAME                                                STATE                AGE
mig-large-training-pod-mig-large-claim-6dpn8     pending              0s
mig-large-training-pod-mig-large-claim-6dpn8     pending              0s
mig-large-training-pod-mig-large-claim-6dpn8     allocated,reserved   0s
mig-medium-training-pod-mig-medium-claim-bk596   pending              0s
mig-medium-training-pod-mig-medium-claim-bk596   pending              0s
mig-medium-training-pod-mig-medium-claim-bk596   allocated,reserved   0s
mig-small-inference-pod-mig-small-claim-d2t58    pending              0s
mig-small-inference-pod-mig-small-claim-d2t58    pending              0s
mig-small-inference-pod-mig-small-claim-d2t58    allocated,reserved   0s
```

보시다시피 모든 포드가 DRA 드라이버에 `allocated,reserved` 의해 보류 중에서 로 이동했습니다.

8. 노드`nvidia-smi`에서를 실행합니다. 세 개의 Python 프로세서가 실행 중인 것을 확인할 수 있습니다.

```
root@ip-100-64-173-145 bin]# nvidia-smi
+-----------------------------------------------------------------------------------------
+
| NVIDIA-SMI 570.158.01 Driver Version: 570.158.01 CUDA Version: 12.8 |
|-----------------------------------------+------------------------
+----------------------+
| GPU Name Persistence-M | Bus-Id Disp.A | Volatile Uncorr. ECC |
| Fan Temp Perf Pwr:Usage/Cap | Memory-Usage | GPU-Util Compute M. |
| | | MIG M. |
|=========================================+========================
+======================|
| 0 NVIDIA A100-SXM4-80GB On | 00000000:10:1C.0 Off | On |
| N/A 63C P0 127W / 400W | 569MiB / 81920MiB | N/A Default |
| | | Enabled |
+-----------------------------------------+------------------------
+----------------------+
| 1 NVIDIA A100-SXM4-80GB On | 00000000:10:1D.0 Off | On |
| N/A 56C P0 121W / 400W | 374MiB / 81920MiB | N/A Default |
| | | Enabled |
```

```
+-----------------------------------------+-----------------------
+----------------------+
| 2 NVIDIA A100-SXM4-80GB On | 00000000:20:1C.0 Off | On |
| N/A 63C P0 128W / 400W | 467MiB / 81920MiB | N/A Default |
| | | Enabled |
+-----------------------------------------+-----------------------
+----------------------+
| 3 NVIDIA A100-SXM4-80GB On | 00000000:20:1D.0 Off | On |
| N/A 57C P0 118W / 400W | 249MiB / 81920MiB | N/A Default |
| | | Enabled |
+-----------------------------------------+-----------------------
+----------------------+
| 4 NVIDIA A100-SXM4-80GB On | 00000000:90:1C.0 Off | 0 |
| N/A 51C P0 77W / 400W | 0MiB / 81920MiB | 0% Default |
| | | Disabled |
+-----------------------------------------+-----------------------
+----------------------+
| 5 NVIDIA A100-SXM4-80GB On | 00000000:90:1D.0 Off | 0 |
| N/A 46C P0 69W / 400W | 0MiB / 81920MiB | 0% Default |
| | | Disabled |
+-----------------------------------------+-----------------------
+----------------------+
| 6 NVIDIA A100-SXM4-80GB On | 00000000:A0:1C.0 Off | 0 |
| N/A 52C P0 74W / 400W | 0MiB / 81920MiB | 0% Default |
| | | Disabled |
+-----------------------------------------+-----------------------
+----------------------+
| 7 NVIDIA A100-SXM4-80GB On | 00000000:A0:1D.0 Off | 0 |
| N/A 47C P0 72W / 400W | 0MiB / 81920MiB | 0% Default |
| | | Disabled |
+-----------------------------------------+-----------------------
+----------------------+



+-----------------------------------------------------------------------------------------------
+
| MIG devices: |
+------------------+--------------------------------+-----------
+----------------------+
| GPU GI CI MIG | Memory-Usage | Vol| Shared |
| ID ID Dev | BAR1-Usage | SM Unc| CE ENC DEC OFA JPG |
| | | ECC| |
|==================+================================+===========
+======================|
```

```
| 0 2 0 0 | 428MiB / 40192MiB | 42 0 | 3 0 2 0 0 |
| | 2MiB / 32767MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 0 3 0 1 | 71MiB / 19968MiB | 28 0 | 2 0 1 0 0 |
| | 0MiB / 16383MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 0 9 0 2 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 0 10 0 3 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 1 1 0 0 | 107MiB / 40192MiB | 42 0 | 3 0 2 0 0 |
| | 0MiB / 32767MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 1 5 0 1 | 71MiB / 19968MiB | 28 0 | 2 0 1 0 0 |
| | 0MiB / 16383MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 1 13 0 2 | 161MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 2MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 1 14 0 3 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 2 1 0 0 | 107MiB / 40192MiB | 42 0 | 3 0 2 0 0 |
| | 0MiB / 32767MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 2 5 0 1 | 289MiB / 19968MiB | 28 0 | 2 0 1 0 0 |
| | 2MiB / 16383MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 2 13 0 2 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
```

```
| 2 14 0 3 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 3 1 0 0 | 107MiB / 40192MiB | 42 0 | 3 0 2 0 0 |
| | 0MiB / 32767MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 3 5 0 1 | 71MiB / 19968MiB | 28 0 | 2 0 1 0 0 |
| | 0MiB / 16383MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 3 13 0 2 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 3 14 0 3 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+



+-----------------------------------------------------------------------------------------
+
| Processes: |
| GPU GI CI PID Type Process name GPU Memory |
| ID ID Usage |
|
=========================================================================================|
**| 0 2 0 64080 C python 312MiB |
| 1 13 0 64085 C python 118MiB |
| 2 5 0 64073 C python 210MiB |**
+-----------------------------------------------------------------------------------------
+
```

GB200 P6e 인스턴스를 사용하여 IMEX로 GPU 워크로드 최적화

IMEX(Internode Memory Exchange)를 사용하면 NVIDIA GB200 UltraServers에서 분산 훈련을 위해 노드 간 메모리 일관성 통신을 사용할 수 있습니다.

다음 단계를 수행합니다.

1. imex-compute-domain.yaml이름이 인 파일을 사용하여 다중 노드 훈련을 ComputeDomain 위한를 정의합니다.

```
apiVersion: resource.nvidia.com/v1beta1
kind: ComputeDomain
metadata:
  name: distributed-training-domain
  namespace: default
spec:
  numNodes: 2
  channel:
    resourceClaimTemplate:
      name: imex-channel-template
```

2. 이름이 인 파일을 사용하여 IMEX 채널을 사용하여 포드를 정의합니다imex-pod.yaml.

```
apiVersion: v1
kind: Pod
metadata:
  name: imex-distributed-training
  namespace: default
  labels:
    app: imex-training
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: nvidia.com/gpu.clique
            operator: Exists
  containers:
  - name: distributed-training
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["bash", "-c"]
    args:
    - |
      echo "=== IMEX Channel Verification ==="
      ls -la /dev/nvidia-caps-imex-channels/
      echo ""

      echo "=== GPU Information ==="
      nvidia-smi
```

```
    echo ""

    echo "=== NCCL Test (if available) ==="
    python -c "
    import torch
    import torch.distributed as dist
    import os

    print(f'CUDA available: {torch.cuda.is_available()}')
    print(f'CUDA device count: {torch.cuda.device_count()}')

    if torch.cuda.is_available():
        for i in range(torch.cuda.device_count()):
            print(f'GPU {i}: {torch.cuda.get_device_name(i)}')

    # Check for IMEX environment variables
    imex_vars = [k for k in os.environ.keys() if 'IMEX' in k or 'NVLINK' in k]
    if imex_vars:
        print('IMEX Environment Variables:')
        for var in imex_vars:
            print(f'  {var}={os.environ[var]}')

    print('IMEX channel verification completed')
    "

    # Keep container running for inspection
    sleep 3600
  resources:
    claims:
    - name: imex-channel-0
    - name: imex-channel-1
resourceClaims:
- name: imex-channel-0
  resourceClaimTemplateName: imex-channel-template
- name: imex-channel-1
  resourceClaimTemplateName: imex-channel-template
tolerations:
- key: nvidia.com/gpu
  operator: Exists
  effect: NoSchedule
```

> **Note**
>
> 여기에는 P6e GB200 인스턴스가 필요합니다.

3. ComputeDomain 및 템플릿을 적용하여 IMEX를 배포합니다.

```
kubectl apply -f imex-claim-template.yaml
kubectl apply -f imex-compute-domain.yaml
kubectl apply -f imex-pod.yaml
```

4. ComputeDomain 상태를 확인합니다.

```
kubectl get computedomain distributed-training-domain
```

5. IMEX 데몬 배포를 모니터링합니다.

```
kubectl get pods -n nvidia-dra-driver -l resource.nvidia.com/computeDomain
```

6. 포드에서 IMEX 채널을 확인합니다.

```
kubectl exec imex-distributed-training -- ls -la /dev/nvidia-caps-imex-channels/
```

7. 포드 로그 보기:

```
kubectl logs imex-distributed-training
```

다음은 예상 출력의 예입니다.

```
=== IMEX Channel Verification ===
total 0
drwxr-xr-x. 2 root root 80 Jul 8 10:45 .
drwxr-xr-x. 6 root root 380 Jul 8 10:45 ..
crw-rw-rw-. 1 root root 241, 0 Jul 8 10:45 channel0
crw-rw-rw-. 1 root root 241, 1 Jul 8 10:45 channel1
```

자세한 내용은 GitHub의 NVIDIA 예제를 참조하세요.

네트워킹

노드 간 통신이 많은 애플리케이션의 경우 더 높은 네트워크 대역폭 또는 탄력적 패브릭 어댑터 고려

노드 간 통신 요구가 높은 Amazon EKS의 분산 훈련 워크로드의 경우 네트워크 대역폭 또는 EFA(Elastic Fabric Adapter)가 더 높은 인스턴스를 선택하는 것이 좋습니다. 네트워크 성능이 충분하지 않으면 데이터 전송에 병목 현상이 발생하여 분산 다중 GPU 훈련과 같은 기계 학습 작업이 느려질 수 있습니다. 추론 워크로드는 일반적으로 노드 간 통신이 높지 않습니다.

예제

예를 들어 Karpenter를 사용하는 경우:

```
apiVersion: v1
kind: Pod
metadata:
  name: ml-workload
spec:
  nodeSelector:
    karpenter.k8s.aws/instance-network-bandwidth: "100000"  # 100 Gbps in Mbps
    node.kubernetes.io/instance-type: p5.48xlarge  # EFA-enabled instance
  containers:
  - name: training-job
    image: `763104351884.dkr.ecr.us-west-2.amazonaws.com/pytorch-inference:2.6.0-gpu-
py312-cu124-ubuntu22.04-ec2-v1.6`
    resources:
      limits:
        vpc.amazonaws.com/efa: 1  # Requires EFA device plugin
```

훈련 작업에 EFA를 활용하려면 컨테이너 이미지에 MPI 및 NCCL과 같은 도구가 설치되어 있는지 확인합니다.

사용 가능한 IP 주소 수를 늘려 포드 시작 시간 단축

EKS에서 각 포드에는 VPC CIDR 블록의 IP 주소가 필요합니다. 클러스터가 더 많은 노드와 포드로 확장되면 IP 주소 소진 또는 느린 성능이 발생할 위험이 있지만 접두사 위임을 활성화하면 IP 범위를 미리 할당하고 EC2 API 호출을 줄여 이러한 문제를 완화하여 포드 시작 시간을 단축하고 확장성을 개선할 수 있습니다.

클러스터를 생성한 후 접두사 위임을 활성화하면 VPC 컨테이너 네트워크 인터페이스(CNI)가 EC2 인스턴스의 네트워크 인터페이스에 IP 접두사(/28, 각 IP 주소 16개 제공)를 할당할 수 있습니다. 즉, 각 노드가 더 많은 포드를 지원하여 IP 부족 위험을 줄일 수 있습니다. 예를 들어 `c5.4xlarge` 인스턴스에서 접두사 위임을 사용하여 최대 110개의 포드를 지원할 수 있습니다.

접두사 위임은 작은 포드가 많은 환경에서 IP 사용을 최적화하는 데 중요하지만 AI/ML 워크로드는 더 적고 큰 포드(예: GPU당 포드 1개)를 사용하는 경우가 많습니다. 접두사 위임을 활성화하면 VPC CNI가 웜 풀을 유지하여 더 빠른 포드 시작을 위해 접두사를 사전 할당할 수 있습니다. 즉, IP 주소를 즉시 사용할 수 있으므로 비접두사 모드에서 온디맨드 할당에 비해 포드 초기화에 필요한 시간이 줄어듭니다. 이러한 경우 접두사 위임을 활성화하여 IP를 절감하면 AI/ML 워크로드에 성능 이점이 있습니다. IP 주소 구성에 필요한 EC2 API 호출 수를 줄이고 IP 범위를 사전 할당함으로써 접두사 위임을 사용하면 포드 시작 시간을 단축할 수 있으며, 이는 AI/ML 워크로드를 빠르게 확장하는 데 특히 유용합니다.

접두사 위임을 활성화하려면:

```
kubectl set env daemonset/aws-node -n kube-system ENABLE_PREFIX_DELEGATION=true
```

특히 대규모 배포에서 IP 주소 소진을 방지하기 위해 VPC 서브넷을 적절하게 계획하고 VPCs 간 중복을 방지하기 위해 CIDR 블록을 관리합니다. 자세한 내용은 IP 주소 사용률 최적화 및 접두사가 있는 Amazon EKS 노드에 더 많은 IP 주소 할당을 참조하세요.

보안

보안 및 규정 준수

암호화 준수 스토리지를 위해 KMS가 포함된 S3 고려

달리 지정하지 않는 한 모든 S3 버킷은 기본적으로 SSE-S3를 사용하여 유휴 객체를 암호화합니다. 그러나 AWS Key Management Service(AWS KMS) 키(SSE-KMS)를 사용한 서버 측 암호화를 대신 사용하도록 버킷을 구성할 수 있습니다. AWS KMS 내 보안 관리가 암호화 관련 규정 준수 요구 사항을 충족하는 데 도움이 될 수 있습니다. 이러한 KMS 키를 사용하여 Amazon S3 버킷에서 데이터를 보호할 수 있습니다. S3 버킷에서 SSE-KMS 암호화를 사용하는 경우 AWS KMS 키는 버킷과 동일한 리전에 있어야 합니다.

SSE-KMS용 S3 버킷 키를 사용하도록 범용 버킷을 구성하여 Amazon S3에서 AWS KMS로의 요청 트래픽을 줄여 AWS KMS 요청 비용을 최대 99% 절감합니다. S3 버킷 키는 디렉터리 버킷의 및 작업에 대해 항상 활성화되며 비활성화할 수 없습니다. `GET` `PUT`

Amazon S3 Express One Zone은 S3 디렉터리 버킷이라는 특정 유형의 버킷을 사용합니다. 디렉터리 버킷은 S3 Express One Zone 스토리지 클래스 전용이며 지연 시간이 짧은 고성능 액세스를 지원합니다. S3 디렉터리 버킷에서 기본 버킷 암호화를 구성하려면 다음 예제와 같이 AWS CLI를 사용하고 별칭이 아닌 KMS 키 ID 또는 ARN을 지정합니다.

Example

```
aws s3api put-bucket-encryption --bucket my-directory-bucket --server-side-encryption-
configuration \
   '{"Rules": [{"ApplyServerSideEncryptionByDefault": {"SSEAlgorithm": "aws:kms",
 "KMSMasterKeyID": "1234abcd-12ab-34cd-56ef-1234567890ab"}}]}'
```

EKS 포드의 IAM 역할에 암호화된 객체에 액세스할 수 있는 KMS 권한(예: `kms:Decrypt`)이 있는지 확인합니다. 버킷에 샘플 모델을 업로드하고, 포드에 탑재하고(예: Mountpoint S3 CSI 드라이버를 통해), 포드가 암호화된 데이터를 오류 없이 읽을 수 있는지 확인하여 스테이징 환경에서 이를 테스트합니다. AWS CloudTrail을 통해 로그를 감사하여 암호화 요구 사항 준수를 확인합니다. 설정 세부 정보 및 키 관리는 KMS 설명서를 참조하세요.

스토리지

데이터 관리 및 스토리지

CSI 드라이버를 사용하여 포드에 AI 모델 배포

AI/ML 워크로드에는 대규모 모델 아티팩트(예: 훈련된 가중치, 구성)에 대한 액세스가 필요한 경우가 많으며, 포드에는 컨테이너 이미지에 임베딩하지 않고도 이러한 아티팩트에 액세스할 수 있는 안정적이고 확장 가능한 방법이 필요하므로 이미지 크기와 컨테이너 레지스트리 풀 타임이 증가할 수 있습니다. 볼륨 마운트 관리의 운영 오버헤드를 줄이려면 해당 CSI 드라이버를 사용하여 Amazon 스토리지 서비스(예: S3, FSx for Lustre, FSx for OpenZFS, EFS)를 영구 볼륨(PVs)으로 탑재하여 포드에 AI 모델을 배포하는 것이 좋습니다. 구현 세부 정보는이 섹션의 후속 주제를 참조하세요.

EKS의 ML 모델 캐시에 대한 스토리지 최적화

포드 및 애플리케이션 시작 지연 시간을 최소화하고, 메모리 사용량을 줄이고, 워크로드를 가속화하기 위해 원하는 수준의 성능을 얻고, ML 워크로드의 확장성을 보장하려면 최적의 스토리지 솔루션을 활용하는 것이 중요합니다. ML 워크로드는 종종 모델 파일(가중치)에 의존합니다. 모델 파일은 크기가 크고 포드 또는 노드 간에 데이터에 대한 공유 액세스가 필요할 수 있습니다. 최적의 스토리지 솔루션을 선택하는 것은 단일 노드 효율성, 다중 노드 액세스, 지연 시간 요구 사항, 비용 제약 및 데이터 통합 요구

사항(예: Amazon S3 데이터 리포지토리)과 같은 워크로드의 특성에 따라 달라집니다. 워크로드로 다양한 스토리지 솔루션을 벤치마킹하여 요구 사항을 충족하는 스토리지 솔루션을 파악하는 것이 좋습니다. 워크로드 요구 사항에 따라 평가하는 데 도움이 되는 다음 옵션을 제공합니다.

EKS CSI 드라이버는 다음과 같은 AWS 스토리지 서비스를 지원합니다. 각 서비스에는 자체 CSI 드라이버가 있으며 AI 및 ML 워크플로에 대한 자체 강점이 있습니다.

- Mountpoint for Amazon S3
- Amazon FSx for Lustre
- Amazon FSx for OpenZFS
- Amazon EFS
- Amazon EBS

AWS 스토리지 서비스의 선택은 배포 아키텍처, 규모, 성능 요구 사항 및 비용 전략에 따라 달라집니다. 스토리지 CSI 드라이버를 EKS 클러스터에 설치해야 합니다. 그러면 CSI 드라이버가 포드의 수명 주기 외부에서 영구 볼륨(PV)을 생성하고 관리할 수 있습니다. CSI 드라이버를 사용하여 지원되는 AWS 스토리지 서비스의 PV 정의를 EKS 클러스터 리소스로 생성할 수 있습니다. 그런 다음 포드는 PV에 대한 영구 볼륨 클레임(PVC)을 생성하여 데이터 볼륨에 대한 이러한 스토리지 볼륨에 액세스할 수 있습니다. AWS 스토리지 서비스 및 배포 시나리오에 따라 단일 PVC(및 관련 PV)를 워크로드의 여러 포드에 연결할 수 있습니다. 예를 들어 ML 훈련의 경우 공유 훈련 데이터는 PV에 저장되고 여러 포드가 액세스합니다. 실시간 온라인 추론의 경우 LLM 모델은 PV에 캐시되고 여러 포드가 액세스합니다. AWS 스토리지 서비스용 샘플 PV 및 PVC YAML 파일은 시작하는 데 도움이 되도록 아래에 나와 있습니다.

성능 모니터링 디스크 성능이 좋지 않으면 컨테이너 이미지 읽기가 지연되고 포드 시작 지연 시간이 증가하며 추론 또는 훈련 처리량이 저하될 수 있습니다. Amazon CloudWatch를 사용하여 AWS 스토리지 서비스의 성능 지표를 모니터링합니다. 성능 병목 현상을 식별하면 스토리지 구성 파라미터를 수정하여 성능을 최적화합니다.

시나리오: 여러 GPU 인스턴스 워크로드

Amazon FSx for Lustre: 분산 훈련 및 모델 서비스와 같이 지연 시간에 민감하고 대역폭 처리량이 높은 동적 워크로드가 있는 여러 EC2 GPU 컴퓨팅 인스턴스 환경이 있고 네이티브 Amazon S3 데이터 리포지토리 통합이 필요한 경우 Amazon FSx for Lustre를 사용하는 것이 좋습니다. FSx for Lustre는 고성능 컴퓨팅(HPC), Machine Learning과 같은 컴퓨팅 집약적인 워크로드를 위해 설계된 완전 관리형 고성능 병렬 파일 시스템을 제공합니다.

FSx for Lustre CSI 드라이버를 설치하여 EKS에 FSx 파일 시스템을 영구 볼륨(PV)으로 탑재한 다음 FSx for Lustre 파일 시스템을 독립형 고성능 캐시 또는 S3-linked 파일 시스템으로 배포하여 S3 데이터

에 대한 고성능 캐시 역할을 하여 GPU 컴퓨팅 인스턴스 전반에서 데이터 액세스를 위한 빠른 I/O와 높은 처리량을 제공할 수 있습니다. FSx for Lustre는 Scratch-SSD 또는 영구-SSD 스토리지 옵션을 사용하여 배포할 수 있습니다.

- Scratch-SSD 스토리지: TiB당 고정 처리량 용량이 프로비저닝된 임시 또는 단기(시간) 워크로드에 권장됩니다.
- 영구 SSD 스토리지: HPC 시뮬레이션, 빅 데이터 분석 또는 Machine Learning 훈련과 같이 최고 수준의 가용성이 필요한 미션 크리티컬 장기 실행 워크로드에 권장됩니다. 영구 SSD 스토리지를 사용하면 필요한 스토리지 용량과 처리량 용량(TiB당)을 모두 구성할 수 있습니다.

성능 고려 사항:

- FSx for Lustre 파일 시스템을 관리하기 위한 관리 포드: lustre 클라이언트가 설치되어 있고 FSx 파일 시스템이 탑재된 "관리" 포드를 구성합니다. 이렇게 하면 액세스 포인트가 FSx 파일 시스템을 미세 조정할 수 있으며, GPU 컴퓨팅 인스턴스를 시작하기 전에 ML 훈련 데이터 또는 LLM 모델로 FSx 파일 시스템을 사전 워밍해야 하는 상황도 가능합니다. 이는 아키텍처가 스팟 기반 Amazon EC2 GPU/컴퓨팅 인스턴스를 활용하는 경우 특히 중요합니다. 스팟 기반 Amazon EC2 인스턴스를 실행할 때 데이터를 처리할 준비가 되도록 관리 포드를 사용하여 FSx 파일 시스템에 원하는 데이터를 "웜" 또는 "사전 로드"할 수 있습니다.
- EFA(Elastic Fabric Adapter): 영구 SSD 스토리지 배포 유형은 EFA(Elastic Fabric Adapter)를 지원합니다. EFA를 사용하면 고성능 및 처리량 기반 GPU 기반 워크로드에 적합합니다. FSx for Lustre는 NVIDIA GPUDirect 스토리지(GDS)를 지원합니다. 여기서 GDS는 로컬 또는 원격 스토리지와 GPU 메모리 간에 직접 데이터 경로를 생성하여 데이터 액세스를 가속화하는 기술입니다.
- 압축: 압축할 수 있는 파일 유형이 있는 경우 파일 시스템에서 데이터 압축을 활성화합니다. 이렇게 하면 데이터 압축으로 FSx for Lustre 파일 서버와 스토리지 간에 전송되는 데이터의 양이 줄어들어 성능이 향상될 수 있습니다.
- Lustre 파일 시스템 스트라이핑 구성:
  - 데이터 스트라이핑: FSx for Luster가 Lustre 파일 시스템 내의 여러 객체 스토리지 대상(OSTs)에 파일 데이터를 분산하도록 허용하여 병렬 액세스 및 처리량, 특히 대규모 ML 훈련 작업의 경우 처리량을 극대화합니다.
  - 독립 실행형 파일 시스템 스트라이핑: 기본적으로 FSx for Lustre의 프로그레시브 파일 레이아웃(PFL) 기능을 통해 4 구성 요소 Lustre 스트라이핑 구성이 생성됩니다. 대부분의 시나리오에서는 기본 PFL Lustre 스트라이프 수/크기를 업데이트할 필요가 없습니다. Lustre 데이터 스트라이핑을 조정해야 하는 경우 FSx for Lustre 파일 시스템의 스트라이핑 파라미터를 참조하여 Lustre 스트라이핑을 수동으로 조정할 수 있습니다.

- S3-Linked 파일 시스템: 기본 Amazon S3 통합(데이터 리포지토리 연결 또는 DRA)을 사용하여 FSx 파일 시스템으로 가져온 파일은 기본 PFL 레이아웃을 사용하지 않고 대신 파일 시스템의 `ImportedFileChunkSize` 파라미터에 레이아웃을 사용합니다. 보다 큰 S3-imported 파일은 `ImportedFileChunkSize` 정의된 값(기본값 `ImportedFileChunkSize` 1GiB)을 기반으로 스트라이프 수와 함께 여러 OSTs에 저장됩니다. 파일이 큰 경우이 파라미터를 더 높은 값으로 조정하는 것이 좋습니다.
- 배치: 컴퓨팅 또는 GPU 노드와 동일한 가용 영역에 FSx for Lustre 파일 시스템을 배포하여 데이터에 대한 지연 시간을 최소화하고 교차 가용 영역 액세스 패턴을 방지합니다. 서로 다른 가용 영역에 여러 GPU 노드가 있는 경우 지연 시간이 짧은 데이터 액세스를 위해 각 가용 영역에 FSx 파일 시스템을 배포하는 것이 좋습니다.

예제

정적 프로비저닝(FSx 인스턴스가 이미 프로비저닝된 경우)을 사용하는 FSx for Lustre 파일 시스템에 대한 영구 볼륨(PV) 정의입니다.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: fsx-pv
spec:
  capacity:
    storage: 1200Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  mountOptions:
    - flock
  persistentVolumeReclaimPolicy: Recycle
  csi:
    driver: fsx.csi.aws.com
    volumeHandle: [FileSystemId of FSx instance]
    volumeAttributes:
      dnsname: [DNSName of FSx instance]
      mountname: [MountName of FSx instance]
```

예제

라는 PV에 대한 영구 볼륨 클레임 정의`fsx-pv`:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: fsx-claim
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: ""
  resources:
    requests:
      storage: 1200Gi
  volumeName: fsx-pv
```

예제

의 영구 볼륨 클레임을 사용하도록 포드를 구성합니다`fsx-claim`.

```
apiVersion: v1
kind: Pod
metadata:
  name: fsx-app
spec:
  containers:
  - name: app
    image: amazonlinux:2023
    command: ["/bin/sh"]
    volumeMounts:
    - name: persistent-storage
      mountPath: /data
  volumes:
  - name: persistent-storage
    persistentVolumeClaim:
      claimName: fsx-claim
```

전체 예제는 GitHub의 FSx for Lustre 드라이버 예제를 참조하세요. Amazon CloudWatch를 사용하여 Amazon FSx for Lustre 성능 지표를 모니터링합니다. Amazon CloudWatch 성능 병목 현상이 식별되면 필요에 따라 구성 파라미터를 조정합니다.

시나리오: 단일 GPU 인스턴스 워크로드

CSI 드라이버가 포함된 Mountpoint for Amazon S3: Mountpoint for Amazon S3 CSI 드라이버를 사용하여 포드에 볼륨으로 S3 버킷을 탑재할 수 있습니다. Amazon S3 이 방법을 사용하면 특정 S3 버킷

에 액세스할 수 있는 포드에 대한 세분화된 액세스 제어가 가능합니다. 각 포드에는 자체 탑재 지점 인스턴스와 로컬 캐시(5~10GB)가 있어 포드 간에 모델 로드 및 읽기 성능을 격리합니다. 이 설정은 서비스 계정에 대한 IAM 역할(IRSA)을 사용한 포드 수준 인증과 다양한 모델 또는 고객에 대한 독립적인 모델 버전 관리를 지원합니다. 각 포드가 S3 API 호출을 실행하고 자체 캐시를 유지 관리하므로 메모리 사용량과 API 트래픽이 증가합니다.

CSI 드라이버가 있는 포드 배포 YAML의 부분 예제:

```
# CSI driver dynamically mounts the S3 bucket for each pod

volumes:
  - name: s3-mount
    csi:
      driver: s3.csi.aws.com
      volumeAttributes:
        bucketName: your-s3-bucket-name
        mountOptions: "--allow-delete"  # Optional
        region: us-west-2

containers:
  - name: inference
    image: your-inference-image
    volumeMounts:
      - mountPath: /models
        name: s3-mount
volumeMounts:
  - name: model-cache
    mountPath: /models
volumes:
  - name: model-cache
    hostPath:
      path: /mnt/s3-model-cache
```

성능 고려 사항:

- 데이터 캐싱: Mountpoint for S3는 콘텐츠를 캐싱하여 비용을 절감하고 동일한 파일에 대한 반복 읽기 성능을 개선할 수 있습니다. 캐싱 옵션 및 파라미터는 캐싱 구성을 참조하세요.
- 객체 부분 크기: 크기가 72GB를 초과하는 파일을 저장하고 액세스하는 경우 Mountpoint 성능 구성을 참조하여 데이터 프로파일 및 워크로드 요구 사항에 맞게 `--read-part-size` 및 `--write-part-size` 명령줄 파라미터를 구성하는 방법을 이해합니다.

- 공유 캐시는 최대 1MB 크기의 객체를 위해 설계되었습니다. 큰 객체는 지원하지 않습니다. 로컬 캐시 옵션을 사용하여 EKS 노드의 NVMe 또는 EBS 볼륨에서 객체를 캐싱합니다.
- API 요청 요금: Mountpoint for S3를 사용하여 많은 수의 파일 작업을 수행할 때 API 요청 요금은 스토리지 비용의 일부가 될 수 있습니다. 이를 완화하려면 강력한 일관성이 필요하지 않은 경우 항상 메타데이터 캐싱을 활성화하고 `metadata-ttl` 기간을 설정하여 API 작업 수를 S3로 줄입니다.

자세한 내용은 Amazon EKS 공식 설명서의 Mountpoint for Amazon S3 CSI Driver를 참조하세요. 병목 현상이 발생하는 경우 Amazon CloudWatch 지표를 사용하여 Amazon Amazon S3의 성능 지표를 모니터링하고 필요한 경우 구성을 조정하는 것이 좋습니다.

Amazon FSx for OpenZFS 영구 공유 스토리지

여러 애플리케이션에 대해 고가용성, 고성능, 비용 민감도 및 여러 포드 배포가 필요한 지연 시간에 민감한 워크로드가 있는 여러 EC2 GPU 컴퓨팅 인스턴스를 포함하는 시나리오의 경우 Amazon FSx for OpenZFS를 사용하는 것이 좋습니다. 일부 워크로드 예제에는 실시간 추론, 강화 학습 및 생성적 적대적 네트워크 훈련이 포함됩니다. FSx for OpenZFS는 작은 IO 데이터 액세스 패턴을 사용하는 작은 파일이 있는 집중 디렉터리 구조에 대한 고성능 액세스가 필요한 워크로드에 특히 유용합니다. 또한 FSx for OpenZFS는 스토리지 용량과 독립적으로 성능을 확장할 수 있는 유연성을 제공하므로 필요한 성능 수준을 유지하면서 스토리지 크기를 실제 요구 사항에 맞춰 최적의 비용 효율성을 달성할 수 있습니다.

기본 FSx for OpenZFS CSI 드라이버를 사용하면 여러 볼륨을 생성하여 단일 파일 시스템에 여러 PVCs를 생성할 수 있습니다. 이렇게 하면 관리 오버헤드가 줄어들고 단일 파일 시스템의 통합 애플리케이션 포드 배포를 통해 파일 시스템의 처리량 및 IOPS 활용도가 극대화됩니다. 또한 제로 복사 스냅샷, 제로 복사 복제본, CSI 드라이버를 통해 동적으로 프로비저닝할 수 있는 사용자 및 그룹 할당량과 같은 엔터프라이즈 기능이 포함되어 있습니다.

FSx for OpenZFS는 생성 시 세 가지 배포 유형을 지원합니다.

- 단일 AZ: 지연 시간이 밀리초 미만인 최저 비용 옵션이지만 파일 시스템 또는 가용 영역 수준에서 고가용성을 제공하지 않습니다. 개발 및 테스트 워크로드 또는 애플리케이션 계층에서 고가용성이 있는 워크로드에 권장됩니다.
- 단일 AZ(HA): 파일 시스템 수준에서 밀리초 미만의 지연 시간으로 고가용성을 제공합니다. 고가용성이 필요한 고성능 워크로드에 권장됩니다.
- 다중 AZ: 파일 시스템 수준과 가용 영역 전체에서 고가용성을 제공합니다. 가용 영역 전체에서 추가 가용성이 필요한 고성능 워크로드에 권장됩니다.

성능 고려 사항:

- 배포 유형: 가용 영역 전반의 추가 가용성이 필요하지 않은 경우 단일 AZ(HA) 배포 유형을 사용하는 것이 좋습니다. 이 배포 유형은 쓰기 처리량의 최대 100%를 제공하고, 밀리초 미만의 지연 시간을 유지하며, Gen2 파일 시스템에는 최대 테라바이트의 자주 액세스하는 데이터를 저장할 수 있는 추가 NVMe 캐시가 있습니다. 다중 AZ 파일 시스템은 쓰기 처리량의 최대 75%를 제공하며 교차 AZ 트래픽에 대한 지연 시간이 증가합니다.
- 처리량 및 IOPS: 배포 후 파일 시스템에 대해 구성된 처리량 및 IOPS를 모두 확장하거나 축소할 수 있습니다. 최대 10GB/s의 디스크 처리량을 프로비저닝하여 최대 21GB/s의 캐시된 데이터 액세스를 제공할 수 있습니다. IOPS는 디스크에서 최대 400,000까지 확장할 수 있으며 캐시는 1백만 IOPS 이상을 제공할 수 있습니다. 단일 AZ 파일 시스템의 처리량 조정으로 인해 고가용성이 없으므로 파일 시스템이 잠시 중단됩니다. 단일 AZ(HA) 또는 다중 AZ 파일 시스템의 처리량 조정은 중단 없이 수행할 수 있습니다. SSD IOPS는 6시간마다 한 번씩 확장할 수 있습니다.
- 스토리지 클래스: FSx for OpenZFS는 SSD 스토리지 클래스와 Intelligent-Tiering 스토리지 클래스를 모두 지원합니다. AI/ML 워크로드의 경우 CPU/GPU의 사용량을 최대한으로 유지하면서 워크로드에 일관된 성능을 제공하는 SSD 스토리지 클래스를 사용하는 것이 좋습니다.
- 압축: 압축할 수 있는 워크로드가 있는 경우 LZ4 압축 알고리즘을 활성화합니다. 이렇게 하면 각 파일이 캐시에서 사용하는 데이터의 양이 줄어들어 캐시에서 직접 더 많은 데이터를 네트워크 처리량 및 IOPS로 제공할 수 있어 SSD 디스크의 부하가 줄어듭니다.
- 레코드 크기: 대부분의 AI/ML 워크로드는 기본 128KiB 레코드 크기를 그대로 두면 도움이 됩니다. 데이터 세트가 애플리케이션에서 128KiB 미만의 일관된 작은 블록 액세스 권한을 가진 큰 파일(10GiB 128KiB 이상)로 구성된 경우에만이 값을 줄여야 합니다.

파일 시스템이 생성되면 서비스에 의해 연결된 루트 볼륨이 자동으로 생성됩니다. 파일 시스템에서 루트 볼륨의 하위 볼륨 내에 데이터를 저장하는 것이 가장 좋습니다. FSx for OpenZFS CSI 드라이버를 사용하여 연결된 영구 볼륨 클레임을 생성하여 하위 볼륨을 동적으로 생성합니다.

예시:

기존 파일 시스템에서 루트 볼륨의 하위 볼륨($ROOT_VOL_ID)을 생성하고 NFS v4.2 프로토콜을 사용하여 볼륨을 VPC CIDR($VPC_CIDR)로 내보내는 데 사용되는 FSx for OpenZFS 볼륨에 대한 스토리지 클래스(SC) 정의입니다.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fsxz-vol-sc
provisioner: fsx.openzfs.csi.aws.com
parameters:
  ResourceType: "volume"
```

```
  ParentVolumeId: '"$ROOT_VOL_ID"'
  CopyTagsToSnapshots: 'false'
  DataCompressionType: '"LZ4"'
  NfsExports: '[{"ClientConfigurations": [{"Clients": "$VPC_CIDR", "Options":
 ["rw","crossmnt","no_root_squash"]}]}]'
  ReadOnly: 'false'
  RecordSizeKiB: '128'
  Tags: '[{"Key": "Name", "Value": "AI-ML"}]'
  OptionsOnDeletion: '["DELETE_CHILD_VOLUMES_AND_SNAPSHOTS"]'
reclaimPolicy: Delete
allowVolumeExpansion: false
mountOptions:
  - nfsvers=4.2
  - rsize=1048576
  - wsize=1048576
  - timeo=600
  - nconnect=16
  - async
```

위에서 생성한 fsxz-vol-sc 생성된 영구 볼륨 클레임(PVC)입니다. 할당된 스토리지 용량은 1Gi이며 CSI 드라이버 FAQ에 명시된 대로 FSx for OpenZFS 볼륨에 필요합니다. 볼륨에는이 구성을 사용하여 파일 시스템에 프로비저닝된 전체 용량이 제공됩니다. 볼륨 용량을 제한해야 하는 경우 사용자 또는 그룹 할당량을 사용하여 제한할 수 있습니다.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: dynamic-vol-pvc
  namespace: example
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: fsxz-vol-sc
  resources:
    requests:
      storage: 1Gi
```

dynamic-vol-pvc의 영구 볼륨 클레임(PVC)을 사용하여 볼륨을 탑재하도록 포드를 구성합니다.

```
kind: Pod
apiVersion: v1
metadata:
```

```
  name: fsx-app
  namespace: example
spec:
  volumes:
    - name: dynamic-vol-pv
      persistentVolumeClaim:
        claimName: dynamic-vol-pvc
  containers:
    - name: app
      image: amazonlinux:2023
      command: ["/bin/sh"]
      volumeMounts:
        - mountPath: "/mnt/fsxz"
          name: dynamic-vol-pv
```

공유 모델 캐시용 Amazon EFS

여러 EC2 GPU 컴퓨팅 인스턴스 환경이 있고 성능 및 확장성 요구 사항이 중간 수준인 여러 노드 및 가용 영역(예: Karpenter를 사용한 실시간 온라인 추론)에서 공유 모델 액세스가 필요한 동적 워크로드가 있는 경우 EFS CSI 드라이버를 통해 Amazon Elastic File System(EFS) 파일 시스템을 영구 볼륨으로 사용하는 것이 좋습니다. Amazon EFS는 완전 관리형, 고가용성 및 확장 가능한 클라우드 기반 NFS 파일 시스템으로, 공유 파일 스토리지가 있는 EC2 인스턴스 및 컨테이너를 일관된 성능으로 사용할 수 있으며 스토리지의 사전 프로비저닝이 필요하지 않습니다. EFS를 모델 볼륨으로 사용하고 EKS 클러스터에서 영구 볼륨을 정의하여 볼륨을 공유 파일 시스템으로 탑재합니다. EFS 파일 시스템에서 지원하는 각 영구 볼륨 클레임(PVC)은 EFS 파일 시스템에 대한 EFS 액세스 포인트로 생성됩니다. EFS를 사용하면 여러 노드와 포드가 동일한 모델 파일에 액세스할 수 있으므로 각 노드의 파일 시스템에 데이터를 동기화할 필요가 없습니다. EFS CSI 드라이버를 설치하여 EFS를 EKS와 통합합니다.

다음 처리량 모드로 Amazon EFS 파일 시스템을 배포할 수 있습니다.

- 버스팅 처리량: 가끔 버스트가 발생하는 다양한 워크로드에 적합한 파일 시스템 크기에 따라 처리량을 조정합니다.
- 프로비저닝된 처리량: 제한 내에서 예측 가능한 성능 요구 사항이 있는 일관된 ML 훈련 작업에 이상적인 전용 처리량입니다.
- 탄력적 처리량(ML에 권장): 다양한 ML 워크로드에 대한 비용 효율성, 워크로드에 따라 자동으로 확장됩니다.

성능 사양을 보려면 Amazon EFS 성능 사양을 참조하세요.

성능 고려 사항:

- 다양한 워크로드에 탄력적 처리량을 사용합니다.
- 활성 ML 워크로드에는 표준 스토리지 클래스를 사용합니다.

Amazon EFS 파일 시스템을 EKS 클러스터 및 포드 내의 영구 볼륨으로 사용하는 전체 예제는 GitHub의 EFS CSI 드라이버 예제를 참조하세요. Amazon CloudWatch를 사용하여 Amazon EFS 성능 지표를 모니터링합니다. Amazon CloudWatch 성능 병목 현상이 식별되면 필요에 따라 구성 파라미터를 조정합니다.

지연 시간에 민감한 객체 지향 워크플로에 S3 Express One Zone 사용

대규모 모델 훈련, 추론 또는 고성능 분석과 같은 Amazon EKS의 지연 시간에 민감한 AI/ML 워크로드의 경우 고성능 모델 스토리지 및 검색에 S3 Express One Zone을 사용하는 것이 좋습니다. S3 Express One Zone은 파일 시스템과 같은 계층적 네임스페이스를 제공합니다.이 네임스페이스는 고속을 유지하면서 "모든 것을 집어넣기"에 적합한 디렉터리 버킷에 업로드하기만 하면 됩니다. 이는 객체 지향 워크플로에 익숙한 경우에 특히 유용합니다. 또는 파일 시스템에 더 익숙한 경우(예: POSIX 준수) Amazon FSx for Lustre 또는 OpenZFS를 선호할 수 있습니다. Amazon S3 Express One Zone은 디렉터리 버킷을 사용하고 여러 AZ에 데이터를 분산하는 표준 S3 버킷보다 짧은 지연 시간을 제공하는 단일 가용 영역(AZs)에 데이터를 저장합니다. 최상의 결과를 얻으려면 Express One Zone 버킷과 동일한 AZ에 EKS 컴퓨팅을 공동 배치해야 합니다. S3 Express One Zone의 차이점에 대한 자세한 내용은 디렉터리 버킷의 차이점을 참조하세요.

파일 시스템 의미 체계를 사용하여 S3 Express One Zone에 액세스하려면 S3 버킷(Express One Zone 포함)을 로컬 파일 시스템으로 탑재하는 Mountpoint S3 CSI 드라이버를 사용하는 것이 좋습니다. S3 이렇게 하면 파일 작업(예: 열기, 읽기, 쓰기)이 S3 API 호출로 변환되어 여러 클라이언트의 읽기 중심 워크로드와 새 객체에 대한 순차 쓰기에 최적화된 높은 처리량의 액세스를 제공합니다. 지원되는 작업 및 제한 사항(예: 전체 POSIX 규정 준수는 없지만 Express One Zone에서 지원되는 추가 및 이름 변경)에 대한 자세한 내용은 Mountpoint 시맨틱 설명서를 참조하세요.

성능 이점

- S3 Standard보다 최대 10배 빠른 데이터 액세스를 제공하며, 일관된 한 자릿수 밀리초 지연 시간과 최대 80% 낮은 요청 비용을 제공합니다.
- 디렉터리 버킷당 초당 수십만 개에서 수백만 개의 요청을 처리하도록 규모를 조정하여 과도한 로드(예: 수십 개에서 수십만 개의 GPUs/CPUs 포화 네트워크가 있는 클러스터) 중에 표준 S3에서 볼 수 있는 제한 또는 브라운아웃을 방지합니다.
- 세션 기반 인증 메커니즘을 사용합니다. 한 번 인증하여 세션 토큰을 얻은 다음 요청당 인증 오버헤드 없이 고속으로 반복 작업을 수행합니다. 이는 빈번한 체크포인트 또는 데이터 로드와 같은 워크로드에 최적화되어 있습니다.

권장 사용 사례

- 캐싱: Mountpoint S3 CSI 드라이버를 S3 Express One Zone과 함께 사용하는 주요 사용 사례 중 하나는 캐싱입니다. 첫 번째 인스턴스는 S3 Standard(범용)에서 데이터를 읽고 지연 시간이 짧은 Express One Zone에서 캐싱합니다. 다른 클라이언트의 후속 읽기는 캐시된 데이터에 더 빠르게 액세스하므로 여러 EKS 노드가 동일한 데이터(예: 공유 훈련 데이터 세트)를 읽는 다중 노드 시나리오에 적합합니다. 이를 통해 반복 액세스의 성능을 최대 7배 개선하고 컴퓨팅 비용을 절감할 수 있습니다. 전체 POSIX 규정 준수(예: 파일 잠금 및 현재 위치 수정)가 필요한 워크로드의 경우 Amazon FSx for Lustre 또는 OpenZFS를 대안으로 고려하세요.
- 대규모 AI/ML 훈련 및 추론: 범용 S3 제한으로 인해 지연이 발생하여 비용이 많이 드는 컴퓨팅 리소스가 낭비될 수 있는 수백 또는 수천 개의 컴퓨팅 노드(예: EKS 클러스터의 GPUs)가 있는 워크로드에 적합합니다. 예를 들어 LLM 연구원 또는 일일 모델 테스트/체크포인트를 실행하는 조직은 리전 S3를 중단하지 않고 빠르고 안정적인 액세스의 이점을 누릴 수 있습니다. 소규모 워크로드(예: 노드 10개)의 경우 S3 Standard 또는 기타 스토리지 클래스로 충분할 수 있습니다.
- 데이터 파이프라인: 모델 로드/준비, 훈련 데이터 아카이브 또는 스트림 체크포인트. 팀이 기존 파일 시스템보다 객체 스토리지를 선호하는 경우(예: S3에 익숙함) FSx for Lustre와 같은 POSIX 호환 옵션에 대한 변경 사항을 엔지니어링하는 대신이 옵션을 사용합니다.

고려 사항

- 복원력: 단일 AZ 설계는 다중 AZ 클래스(99.99% 가용성)에 비해 99.999999999% 내구성(표준 S3와 동일, AZ 내의 중복성을 통해)을 제공하지만 가용성은 낮습니다(99.95% 설계, 99.9% SLA). AZ 장애에 대한 복원력이 떨어집니다. 다시 생성 가능하거나 캐시된 데이터에 사용합니다. 중요한 워크로드에 대해 다중 AZ 복제 또는 백업을 고려합니다.
- API 및 기능 지원: S3 APIs의 하위 집합을 지원합니다(예: 수명 주기 정책 또는 복제 없음). 세션 인증 또는 객체 처리를 위해 사소한 앱 변경이 필요할 수 있습니다.
- EKS 통합: 디렉터리 버킷과 동일한 AZ에 EKS 포드/노드를 공동 배치하여 네트워크 지연 시간을 최소화합니다. Kubernetes 네이티브 액세스를 위해 Mountpoint for Amazon S3 또는 CSI 드라이버를 사용합니다.
- 테스트: 비프로덕션 환경에서 검색 지연 시간을 테스트하여 성능 향상을 검증합니다. 표준 S3 시나리오(예: 높은 GPU 포화도)에서 제한을 모니터링하고 비교합니다.

S3 Express One Zone 스토리지 클래스는 여러 리전에서 사용할 수 있으며 스토리지를 기다리지 않고 객체 액세스가 필요한 워크로드에 대해 EKS와 통합됩니다. 자세한 내용은 S3 Express One Zone 시작하기를 참조하세요.

관찰성

모니터링 및 관찰성

GPU 지표 설명

GPU 사용률 지표는 샘플 기간 동안 GPU가 작업을 실행했는지 여부를 보여줍니다. 이 지표는 GPU가 하나 이상의 명령을 실행한 시간의 백분율을 캡처하지만 GPU가 하드웨어를 얼마나 효율적으로 사용했는지는 나타내지 않습니다. GPU에는 명령을 실행하는 병렬 처리 장치인 여러 스트리밍 멀티프로세서(SMs)가 포함되어 있습니다. 100% 사용률 판독값은 GPU가 모든 SMs에서 과중한 병렬 워크로드를 실행했음을 의미하거나 샘플 기간 동안 하나의 작은 명령이 GPU를 활성화했음을 의미할 수 있습니다. 실제 사용률을 이해하려면 하드웨어 아키텍처의 여러 수준에서 GPU 지표를 검사해야 합니다. 각 스트리밍 멀티프로세서는 서로 다른 코어 유형으로 빌드되며 각 계층은 서로 다른 성능 특성을 노출합니다. 최상위 지표(nvidia-smi를 통해 표시되는 GPU 사용률, 메모리 사용률, GPU 전력 및 GPU 온도)는 디바이스가 활성 상태인지 여부를 보여줍니다. 더 깊은 지표(SM 사용률, SM 활동 및 텐서 코어 사용량)는 GPU가 리소스를 얼마나 효율적으로 사용하는지 보여줍니다.

높은 GPU 전력 사용량 목표

사용률이 낮은 GPUs 워크로드가 모든 GPU 구성 요소를 동시에 참여시키지 못하기 때문에 컴퓨팅 용량을 낭비하고 비용을 높입니다. Amazon EKS의 AI/ML 워크로드의 경우 프록시로 GPU 전력 사용량을 추적하여 실제 GPU 활동을 식별합니다. GPU 사용률은 GPU가 커널을 실행하는 시간의 비율을 보고하지만 스트리밍 멀티프로세서, 메모리 컨트롤러 및 텐서 코어가 모두 동시에 활성 상태인지 여부는 드러나지 않습니다. 완전히 연결된 하드웨어는 경량 커널을 실행하거나 작업 사이에 유휴 상태로 있는 하드웨어보다 훨씬 더 많은 전력을 소비하기 때문에 전력 사용량이이 격차를 야기합니다. GPU의 열 설계 성능(TDP)과 전력 소비를 비교하여 사용률이 낮은 것을 확인한 다음 CPU 사전 처리, 네트워크 I/O 또는 비효율적인 배치 크기로 인해 워크로드에 병목 현상이 발생하는지 조사합니다.

Amazon EKS에서 CloudWatch Container Insights를 설정하여 GPU 전력 소비가 적은 포드, 노드 또는 워크로드를 식별합니다. 이 도구는 Amazon EKS와 직접 통합되며, GPU 전력 소비를 모니터링하고 전력 사용량이 목표 수준 아래로 떨어질 때 포드 스케줄링 또는 인스턴스 유형을 조정할 수 있습니다. 고급 시각화 또는 사용자 지정 대시보드가 필요한 경우 Kubernetes 네이티브 모니터링을 위해 NVIDIA의 DCGM-Exporter를 Prometheus 및 Grafana와 함께 사용합니다. 두 지표 모두 `nvidia_smi_power_draw` (GPU 전력 소비) 및 `nvidia_smi_temperature_gpu` (GPU 온도)와 같은 주요 NVIDIA 지표에 접근합니다. 지표 목록은 https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch-Agent-NVIDIA-GPU.htm 참조하세요. 특정 시간 동안 또는 특정 작업에 대해 지속적으로 낮은 전력 사용량과 같은 패턴을 찾습니다. 이러한 추세는 워크로드를 통합하거나 리소스 할당을 조정할 위치를 식별하는 데 도움이 됩니다.

Kubernetes의 정적 리소스 제한(예: CPU, 메모리 및 GPU 수)은 특히 수요가 변동하는 추론과 같은 동적 AI/ML 워크로드의 경우 과다 프로비저닝 또는 과소 사용으로 이어지는 경우가 많습니다. 사용률 추세를 분석하고 워크로드를 더 적은 수의 GPUs. 추가 GPU를 할당하기 전에 각 GPU가 전체 사용률에 도달했는지 확인합니다. 이 접근 방식은 낭비를 줄이고 비용을 절감합니다. 예약 및 공유 전략 최적화에 대한 자세한 지침은 EKS 컴퓨팅 및 오토 스케일링 모범 사례를 참조하세요.

관찰성 및 지표

AI/ML 워크로드에 모니터링 및 관찰성 도구 사용

최신 AI/ML 서비스에는 인프라, 모델링 및 애플리케이션 로직 간의 조정이 필요합니다. 플랫폼 엔지니어는 인프라 및 관찰성 스택을 관리합니다. 지표를 수집, 저장 및 시각화합니다. AI/ML 엔지니어는 모델별 지표를 정의하고 다양한 로드 및 데이터 분포에서 성능을 모니터링합니다. 애플리케이션 개발자는 APIs 사용하고, 요청을 라우팅하고, 서비스 수준 지표 및 사용자 상호 작용을 추적합니다. 통합 관찰성 관행이 없으면 이러한 팀은 사일로에서 작업하고 시스템 상태 및 성능에 대한 중요한 신호를 놓치게 됩니다. 환경 간에 공유 가시성을 설정하면 모든 이해관계자가 문제를 조기에 감지하고 안정적인 서비스를 유지할 수 있습니다.

AI/ML 워크로드에 Amazon EKS 클러스터를 최적화하면 특히 GPU 메모리 관리와 관련된 고유한 모니터링 문제가 발생합니다. 적절한 모니터링이 없으면 조직은 out-of-memory(OOM) 오류, 리소스 비효율성 및 불필요한 비용에 직면하게 됩니다. 효과적인 모니터링을 통해 EKS 고객의 성능, 복원력을 높이고 비용을 절감할 수 있습니다. 세 가지 모니터링 계층을 결합하는 전체적인 접근 방식을 사용합니다. 먼저 NVIDIA DCGM Exporter를 사용하여 세분화된 GPU 지표를 모니터링하여 GPU 전력 사용량, GPU 온도, SM 활동, SM 점유율 및 XID 오류를 추적합니다. 둘째, Ray 및 vLLM과 같은 추론 서비스 프레임워크를 모니터링하여 네이티브 지표를 통해 분산 워크로드 인사이트를 얻습니다. 셋째, 애플리케이션 수준 인사이트를 수집하여 워크로드와 관련된 사용자 지정 지표를 추적합니다. 이 계층화된 접근 방식을 사용하면 하드웨어 사용률부터 애플리케이션 성능까지 파악할 수 있습니다.

도구 및 프레임워크

여러 도구와 프레임워크는 AI/ML 워크로드를 모니터링하기 위한 기본 out-of-the-box 지표를 제공합니다. 이러한 기본 제공 지표를 사용하면 사용자 지정 계측이 필요하지 않으며 설정 시간이 단축됩니다. 지표는 추론 서비스 및 벤치마킹에 중요한 지연 시간, 처리량 및 토큰 생성과 같은 성능 측면에 중점을 둡니다. 네이티브 지표를 사용하면 사용자 지정 컬렉션 파이프라인을 빌드하지 않고도 즉시 모니터링을 시작할 수 있습니다.

- vLLM: 요청 지연 시간 및 메모리 사용량과 같은 기본 지표를 제공하는 대규모 언어 모델(LLMs)을 위한 고처리량 서비스 엔진입니다.

- Ray: 작업 실행 시간 및 리소스 사용률을 포함하여 확장 가능한 AI 워크로드에 대한 지표를 내보내는 분산 컴퓨팅 프레임워크입니다.
- Hugging Face Text Generation Inference(TGI): 추론 성능에 대한 지표가 내장된 LLMs을 배포하고 제공하기 위한 도구 키트입니다.
- NVIDIA genai-perf: 생성형 AI 모델을 벤치마킹하고 처리량, 지연 시간 및 특정 시간 간격으로 완료된 요청과 같은 LLM별 지표를 측정하는 명령줄 도구입니다.

관찰성 방법

다음 방법 중 하나로 추가 관찰성 메커니즘을 구현하는 것이 좋습니다.

CloudWatch Container Insights 조직에서 최소한의 설정으로 AWS 네이티브 도구를 선호하는 경우 CloudWatch Container Insights를 사용하는 것이 좋습니다. NVIDIA DCGM Exporter와 통합되어 GPU 지표를 수집하고 빠른 인사이트를 위해 사전 구축된 대시보드를 제공합니다. 클러스터에 CloudWatch Observability 추가 기능을 설치하여 활성화되는 Container Insights는 NVIDIA DCGM Exporter의 수명 주기를 배포하고 관리합니다.이 수명 주기는 Nvidia 드라이버에서 GPU 지표를 수집하여 CloudWatch에 노출합니다.

Container Insights를 설치한 후 CloudWatch는 환경에서 NVIDIA GPUs 자동으로 감지하고 중요한 상태 및 성능 지표를 수집합니다. 이러한 지표는 큐레이션된 out-of-the-box 대시보드에 표시됩니다. 통합 CloudWatch 에이전트를 사용하여 Ray 및 vLLM을 CloudWatch와 통합하여 기본 지표를 전송할 수도 있습니다. CloudWatch 이 통합 접근 방식은 EKS 환경의 관찰성을 간소화하고 팀이 모니터링 인프라를 구축하는 대신 성능 튜닝 및 비용 최적화에 집중할 수 있도록 합니다.

사용 가능한 지표의 전체 목록은 Amazon EKS 및 Kubernetes Container Insights 지표를 참조하세요. GPU 모니터링 구현에 대한 step-by-step 지침은 Amazon CloudWatch Container Insights를 사용하여 NVIDIA GPU 워크로드에 대한 운영 인사이트 얻기를 참조하세요. 추론 지연 시간 최적화의 실제 예는 AI 응답성 최적화: Amazon Bedrock 지연 시간 최적화 추론에 대한 실제 가이드를 참조하세요.

Managed Prometheus 및 Grafana 조직에 사용자 지정 대시보드와 고급 시각화 기능이 필요한 경우 Kubernetes용 NVIDIA DCGM-Exporter 및 Grafana 네이티브 모니터링을 사용하여 Prometheus를 배포합니다. Prometheus는 DCGM-Exporter에서 GPU 지표를 스크레이프하고 저장하는 반면, Grafana는 유연한 시각화 및 알림 기능을 제공합니다. 이 접근 방식을 사용하면 CloudWatch Container Insights에 비해 대시보드 설계 및 지표 보존을 더 잘 제어할 수 있습니다.

Ray 및 vLLM Ray 및 vLLM과 같은 오픈 소스 프레임워크https://awslabs.github.io/ai-on-eks/docs/blueprints/inference/GPUs/vLLM-rayserve를 통합하여 기본 지표를 Prometheus로 내보내이 모니터링 스택을 확장할 수 있습니다. 또한 Grafana를 AWS X-Ray 데이터 소스에 연결하여 분산 트레이스를 시

각화하고 추론 파이프라인 전반의 성능 병목 현상을 식별할 수 있습니다. 이 조합은 애플리케이션 수준 요청 흐름을 통해 GPU 수준 지표의 end-to-end 가시성을 제공합니다.

이 모니터링 스택 배포에 대한 step-by-step 지침은 AWS 관리형 오픈 소스 서비스를 사용하여 Amazon EKS에서 GPU 워크로드 모니터링을 참조하세요.

코어 훈련 및 미세 조정 지표 모니터링 고려

핵심 훈련 지표를 모니터링하여 Amazon EKS 클러스터와 클러스터에서 실행되는 기계 학습 워크로드의 상태와 성능을 추적합니다. 훈련 워크로드는 장기간 실행되고 리소스를 다르게 소비하며 모델 수렴 및 데이터 파이프라인 효율성에 대한 가시성이 필요하기 때문에 추론 워크로드와 모니터링 요구 사항이 다릅니다. 아래 지표는 병목 현상을 식별하고, 리소스 할당을 최적화하고, 훈련 작업이 성공적으로 완료되도록 하는 데 도움이 됩니다. 이 모니터링 접근 방식 구현에 대한 step-by-step 지침은 Amazon EKS에서 기계 학습 워크로드 관찰 소개를 참조하세요.

리소스 사용 지표

리소스 사용량 지표를 모니터링하여 리소스가 제대로 사용되고 있는지 확인합니다. 이러한 지표는 병목 현상과 근본 원인 성능 문제를 식별하는 데 도움이 됩니다.

- CPU, 메모리, 네트워크, GPU 전력 및 GPU 온도 - 할당된 리소스가 워크로드 수요를 충족하고 최적화 기회를 식별할 수 있도록 이러한 지표를 모니터링합니다. gpu_memory_usage_bytes와 같은 지표를 추적하여 메모리 소비 패턴을 식별하고 최대 사용량을 감지합니다. 훈련 중 가장 높은 메모리 수요를 이해하기 위해 95번째 백분위수(P95)와 같은 백분위수를 계산합니다. 이 분석을 통해 모델과 인프라를 최적화하여 OOM 오류를 방지하고 비용을 절감할 수 있습니다.
- SM 점유, SM 활동, FPxx 활동 - 이러한 지표를 모니터링하여 GPU의 기본 리소스가 어떻게 사용되고 있는지 파악합니다. SM 활동에 대한 대상 0.8은 경험 규칙입니다.
- 노드 및 포드 리소스 사용률 - 노드 및 포드 수준에서 리소스 사용량을 추적하여 리소스 경합과 잠재적 병목 현상을 식별합니다. 노드가 용량 제한에 근접하는지 모니터링하여 포드 스케줄링을 지연시키고 훈련 작업을 늦출 수 있습니다.
- 리소스 사용률 요청 및 한도와 비교 - 실제 리소스 사용량을 구성된 요청 및 한도와 비교하여 클러스터가 현재 워크로드를 처리하고 향후 워크로드를 수용할 수 있는지 확인합니다. 이 비교는 OOM 오류 또는 리소스 낭비를 방지하기 위해 리소스 할당을 조정해야 하는지 여부를 보여줍니다.
- ML 프레임워크의 내부 지표 - TensorFlow 및 PyTorch와 같은 ML 프레임워크에서 내부 훈련 및 수렴 지표를 캡처합니다. 이러한 지표에는 손실 곡선, 학습률, 배치 처리 시간 및 훈련 단계 기간이 포함됩니다. TensorBoard 또는 유사한 도구를 사용하여 이러한 지표를 시각화하여 모델 수렴을 추적하고 훈련 비효율성을 식별합니다.

모델 성능 지표

모델 성능 지표를 모니터링하여 훈련 프로세스가 정확도 및 비즈니스 요구 사항을 충족하는 모델을 생성하는지 확인합니다. 이러한 지표는 훈련을 중지하고, 모델 버전을 비교하고, 성능 저하를 식별할 시기를 결정하는 데 도움이 됩니다.

- 정확도, 정밀도, 재현율 및 F1-score 이러한 지표를 추적하여 모델이 검증 데이터에서 얼마나 잘 작동하는지 파악합니다. 각 훈련 에포크 후 검증 세트의 F1-score 계산하여 모델이 개선되고 있는지 여부와 허용 가능한 성능 수준에 도달했을 때를 평가합니다.
- 비즈니스별 지표 및 KPIs - AI/ML 이니셔티브의 비즈니스 가치를 직접 측정하는 지표를 정의하고 추적합니다. 권장 시스템의 경우 클릭률 또는 전환율과 같은 지표를 추적하여 모델이 의도한 비즈니스 성과를 주도하는지 확인합니다.
- 시간 경과에 따른 성능 - 모델 버전과 훈련 실행의 성능 지표를 비교하여 추세를 식별하고 성능 저하를 감지합니다. 최신 모델 버전이 기준 모델과 비교하여 성능을 유지하는지 또는 개선하는지 추적합니다. 이 기록 비교는 새 모델을 배포할지 아니면 훈련 문제를 조사할지 결정하는 데 도움이 됩니다.

데이터 품질 및 드리프트 지표

데이터 품질 및 드리프트 지표를 모니터링하여 훈련 데이터가 일관되고 대표적인지 확인합니다. 데이터 드리프트는 시간이 지남에 따라 모델 성능이 저하될 수 있는 반면, 데이터 품질 문제는 모델이 수렴하거나 신뢰할 수 없는 결과를 생성하는 것을 방지할 수 있습니다.

- 입력 데이터의 통계 속성 - 평균, 표준 편차 및 시간 경과에 따른 입력 특성 분포와 같은 통계 속성을 추적하여 데이터 드리프트 또는 이상을 감지합니다. 특성 분포가 기준 훈련 데이터에서 크게 이동하는지 모니터링합니다. 예를 들어 중요한 기능의 평균이 표준 편차 2개를 초과하여 변경되는 경우 데이터 파이프라인이 변경되었는지 또는 기본 데이터 소스가 변경되었는지 조사합니다.
- 데이터 드리프트 감지 및 알림 - 훈련에 영향을 미치기 전에 데이터 품질 문제를 감지하고 경고하는 자동화된 메커니즘을 구현합니다. Kolmogorov-Smirnov 테스트 또는 카이 제곱 테스트와 같은 통계 테스트를 사용하여 현재 데이터 분포를 원래 훈련 데이터와 비교합니다. 테스트에서 상당한 드리프트가 감지되면 알림을 설정하여 업데이트된 데이터로 모델을 재훈련하거나 데이터 파이프라인 문제를 조사할 수 있습니다.

지연 시간 및 처리량 지표

지연 시간 및 처리량 지표를 모니터링하여 훈련 파이프라인의 병목 현상을 식별하고 리소스 사용률을 최적화합니다. 이러한 지표는 훈련 중에 시간이 소비되는 부분과 최적화 작업에 집중할 부분을 이해하는 데 도움이 됩니다.

- ML 훈련 파이프라인의 End-to-End 지연 시간 - 데이터 수집부터 모델 업데이트까지 전체 훈련 파이프라인을 통해 데이터가 흐르는 총 시간을 측정합니다. 훈련 실행 전반에 걸쳐이 지표를 추적하여 파이프라인 변경으로 성능이 향상되는지 또는 저하되는지 확인합니다. 지연 시간이 길면 노드 간 데이터 로드, 전처리 또는 네트워크 통신의 병목 현상이 발생하는 경우가 많습니다.
- 훈련 처리량 및 처리 속도 - 훈련 파이프라인이 시간 단위당 처리하는 데이터의 양을 추적하여 효율적인 리소스 활용을 보장합니다. 초당 처리된 샘플 또는 분당 완료된 배치와 같은 지표를 모니터링합니다. 하드웨어 용량에 비해 처리량이 낮으면 GPU 주기를 낭비하는 데이터 로드, 전처리 또는 모델 계산의 비효율성을 시사합니다.
- 체크포인트 저장 및 복원 지연 시간 - 작업을 재개하거나 장애로부터 복구할 때 모델 체크포인트를 스토리지(S3, EFS, FSx)에 저장하고 GPU 또는 CPU 메모리에 복원하는 데 필요한 시간을 모니터링합니다. 느린 체크포인트 작업은 작업 복구 시간을 연장하고 비용을 높입니다. 체크포인트 크기, 저장 기간, 복원 기간 및 실패 수를 추적하여 압축 또는 더 빠른 스토리지 계층과 같은 최적화 기회를 식별합니다.
- 데이터 로드 및 사전 처리 시간 - 스토리지에서 데이터를 로드하고 사전 처리 변환을 적용하는 데 소요된 시간을 측정합니다. 이 시간을 모델 계산 시간과 비교하여 훈련이 데이터 바인딩인지 컴퓨팅 바인딩인지 확인합니다. 데이터 로드가 총 훈련 시간의 20% 이상을 소비하는 경우 캐싱, 미리 가져오기 또는 더 빠른 스토리지로 데이터 파이프라인을 최적화하는 것이 좋습니다.

오류 발생률 및 실패

훈련 파이프라인 전체에서 오류율과 실패를 모니터링하여 신뢰성을 유지하고 컴퓨팅 리소스 낭비를 방지합니다. 감지되지 않은 오류로 인해 훈련 작업이 자동으로 실패하거나, 잘못된 모델을 생성하거나, 문제가 발견되기 전에 GPU 시간을 낭비할 수 있습니다.

- 파이프라인 오류 모니터링 - 데이터 사전 처리, 모델 훈련 및 체크포인트 작업을 포함하여 ML 파이프라인의 모든 단계에서 오류를 추적합니다. 오류 유형, 빈도 및 영향을 받는 구성 요소를 로깅하여 문제를 빠르게 식별합니다. 일반적인 오류로는 데이터 형식 불일치, 사전 처리 중 out-of-memory 실패, 스토리지 제한으로 인한 체크포인트 저장 실패 등이 있습니다. 오류 발생률이 기준 임계값을 초과할 때 알림을 설정하여 오류 캐스케이드 전에 조사할 수 있습니다.
- 반복 오류 분석 - 반복 오류의 패턴을 식별 및 조사하여 향후 장애를 방지하고 파이프라인 신뢰성을 개선합니다. 로그를 분석하여 특정 데이터 샘플, 배치 크기 또는 훈련 구성이 일관되게 실패를 유발하는지 확인합니다. 예를 들어 특정 입력 데이터 유형이 사전 처리 오류를 트리거하는 경우 파이프라인 앞부분에서 검증 검사를 추가하거나 데이터 정리 로직을 업데이트합니다. 평균 실패 간격(MTBF)을 추적하여 시간이 지남에 따라 파이프라인 신뢰성이 개선되는지 여부를 측정합니다."

Kubernetes 및 EKS 특정 지표

Kubernetes 및 EKS 지표를 모니터링하여 클러스터 인프라가 정상 상태를 유지하고 훈련 워크로드를 지원할 수 있는지 확인합니다. 이러한 지표는 훈련 작업 실패 또는 성능 저하를 유발하기 전에 인프라 문제를 감지하는 데 도움이 됩니다.

- Kubernetes 클러스터 상태 지표 - 포드, 노드, 배포 및 서비스를 포함한 Kubernetes 객체의 상태와 상태를 모니터링합니다. 포드 상태를 추적하여 보류 중, 실패 또는 충돌 루프 상태에 멈춘 포드를 식별합니다. 노드 조건을 모니터링하여 디스크 압력, 메모리 압력 또는 네트워크 사용 불가와 같은 문제를 감지합니다. kubectl 또는 모니터링 도구를 사용하여 이러한 지표를 지속적으로 확인하고 포드가 시작되지 않거나 노드를 예약할 수 없게 되면 알림을 설정합니다.
- 훈련 파이프라인 실행 지표 - 성공 및 실패한 파이프라인 실행, 작업 기간, 단계 완료 시간 및 오케스트레이션 오류를 추적합니다. 훈련 작업이 예상 기간 내에 완료되는지 여부와 시간이 지남에 따라 실패율이 증가하는지 모니터링합니다. 작업 성공률, 평균 작업 기간, 실패까지의 시간과 같은 지표를 추적합니다. 이러한 지표는 인프라 문제, 구성 문제 또는 데이터 품질 문제로 인해 훈련 실패가 발생하는지 여부를 식별하는 데 도움이 됩니다.
- AWS 서비스 지표 - EKS 인프라 및 훈련 워크로드를 지원하는 AWS 서비스에 대한 지표를 추적합니다. 요청 지연 시간, 오류율 및 처리량과 같은 S3 지표를 모니터링하여 데이터 로드 성능이 일관되게 유지되는지 확인합니다. IOPS, 처리량 및 대기열 길이를 포함한 EBS 볼륨 지표를 추적하여 스토리지 병목 현상을 탐지합니다. VPC 흐름 로그 및 네트워크 지표를 모니터링하여 노드 간 또는 외부 서비스에 대한 연결 문제를 식별합니다.
- Kubernetes 컨트롤 플레인 지표 - API 서버, 스케줄러, 컨트롤러 관리자 및 기타 데이터베이스를 모니터링하여 클러스터 작업에 영향을 미치는 성능 문제 또는 장애를 감지합니다. API 서버 요청 지연 시간, 요청 속도 및 오류율을 추적하여 컨트롤 플레인이 예약 요청에 신속하게 응답하도록 합니다. etcd 데이터베이스 크기, 커밋 기간 및 리더 변경 사항을 모니터링하여 안정성 문제를 감지합니다. API 서버 지연 시간이 길거나 etcd 리더를 자주 변경하면 포드 일정이 지연되고 훈련 작업 시작 시간이 연장될 수 있습니다.

애플리케이션 및 인스턴스 로그

애플리케이션 및 인스턴스 로그를 수집하고 분석하여 지표만으로는 설명할 수 없는 문제를 진단합니다. 로그는 오류, 상태 변경 및 시스템 이벤트에 대한 자세한 컨텍스트를 제공하여 훈련 작업이 실패하거나 제대로 수행되지 않는 이유를 이해하는 데 도움이 됩니다. 로그를 지표와 연결하면 근본 원인을 더 빠르게 파악할 수 있습니다.

- 애플리케이션 로그 - 훈련 작업, 데이터 파이프라인 및 ML 프레임워크에서 애플리케이션 로그를 수집하여 병목 현상을 식별하고 장애를 진단합니다. 이러한 로그는 데이터 로드 오류, 모델 초기화 실패, 체크포인트 저장 오류, 프레임워크별 경고 등 작업 실행에 대한 자세한 정보를 캡처합니다. 로그 타임스탬프를 지표 스파이크와 연관시켜 성능 저하 또는 장애의 원인을 파악합니다. 예를 들어 GPU

사용률이 갑자기 떨어지면 애플리케이션 로그에서 데이터 파이프라인이 중단되거나 사전 처리 실패를 나타내는 오류가 있는지 확인합니다. CloudWatch Logs 또는 Fluent Bit와 같은 중앙 집중식 로깅 도구를 사용하여 모든 포드의 로그를 집계하고 검색할 수 있도록 합니다.

- 인스턴스 로그 - 시스템 저널 로그 및 dmesg 출력과 같은 인스턴스 수준 로그를 수집하여 하드웨어 문제 및 커널 수준 문제를 감지합니다. 이러한 로그는 GPU 드라이버 오류, 메모리 할당 실패, 디스크 I/O 오류, 애플리케이션 로그에 나타나지 않을 수 있는 네트워크 인터페이스 문제와 같은 문제를 보여줍니다. 인스턴스 로그를 애플리케이션 로그 및 지표와 연관시켜 훈련 실패가 하드웨어 문제 또는 애플리케이션 문제로 인한 것인지 확인합니다. 예를 들어 out-of-memory 오류로 인해 훈련 작업이 실패하는 경우 시스템에서 메모리가 부족했는지 또는 애플리케이션이 컨테이너 제한을 초과했는지 여부를 나타내는 커널 OOM 킬러 메시지가 있는지 dmesg 로그를 확인합니다. GPU XID 오류 또는 디스크 장애와 같은 중요한 하드웨어 오류에 대한 알림을 설정하여 장애가 발생한 인스턴스가 광범위한 훈련 중단을 일으키기 전에 교체할 수 있습니다.

다음 섹션에서는 CloudWatch Container Insights 및 Amazon Managed Prometheus와 Amazon Managed Grafana의 두 가지 AWS 권장 접근 방식을 사용하여 위에서 설명한 지표를 수집하는 방법을 보여줍니다. 최소한의 설정과 사전 구축된 대시보드로 AWS 네이티브 도구를 선호하는 경우 CloudWatch Container Insights를 선택합니다. 사용자 지정 대시보드, 고급 시각화 기능이 필요하거나 기존 Prometheus 기반 모니터링 인프라와 통합하려는 경우 Amazon Managed Grafana에서 Amazon Managed Prometheus를 선택합니다. 사용 가능한 Container Insights 지표의 전체 목록은 Amazon EKS 및 Kubernetes Container Insights 지표를 참조하세요.

실시간 온라인 추론 지표 모니터링 고려

실시간 시스템에서는 사용자 또는 기타 종속 시스템에 적시에 응답을 제공하려면 짧은 지연 시간이 중요합니다. 지연 시간이 길면 사용자 경험이 저하되거나 성능 요구 사항을 위반할 수 있습니다. 추론 지연 시간에 영향을 미치는 구성 요소에는 모델 로드 시간, 사전 처리 시간, 실제 예측 시간, 사후 처리 시간, 네트워크 전송 시간이 포함됩니다. 추론 지연 시간을 모니터링하여 서비스 수준 계약(SLAs)을 충족하는 지연 시간이 짧은 응답을 보장하고 다음에 대한 사용자 지정 지표를 개발하는 것이 좋습니다. 예상 로드 상태에서 테스트하고, 네트워크 지연 시간을 포함하고, 동시 요청을 고려하고, 다양한 배치 크기로 테스트합니다.

- 첫 번째 토큰까지의 시간(TTFT) - 사용자가 요청을 제출한 시점부터 응답의 시작(첫 번째 단어, 토큰 또는 청크)을 수신할 때까지의 시간입니다. 예를 들어 챗봇에서는 사용자가 질문한 후 첫 번째 출력(토큰)을 생성하는 데 걸리는 시간을 확인합니다.
- End-to-End 지연 시간 - 요청이 수신된 시점부터 응답이 다시 전송된 시점까지의 총 시간입니다. 예를 들어 요청에서 응답까지의 시간을 측정합니다.

- 초당 출력 토큰(TPS) - 모델이 응답을 시작한 후 새 토큰을 생성하는 속도를 나타냅니다. 예를 들어 챗봇에서는 기준 텍스트에 대한 언어 모델의 생성 속도를 추적합니다.
- 오류율 - 성능 문제를 나타낼 수 있는 실패한 요청을 추적합니다. 예를 들어 대용량 문서 또는 특정 문자에 대한 실패한 요청을 모니터링합니다.
- 처리량 - 시스템이 시간 단위당 처리할 수 있는 요청 또는 작업 수를 측정합니다. 예를 들어 최대 부하를 처리하기 위해 초당 요청을 추적합니다.

K/V(Key/Value) 캐시는 추론 지연 시간을 위한 강력한 최적화 기술일 수 있으며, 특히 변환기 기반 모델과 관련이 있습니다. K/V 캐시는 이전 변환기 계층 계산의 키 및 값 텐서를 저장하여 자동 회귀 추론, 특히 대규모 언어 모델(LLMs)에서 중복 계산을 줄입니다. 캐시 효율성 지표(특히 K/V 또는 세션 캐시 사용):

- 캐시 적중률/누락률 - 캐싱(K/V 또는 캐시 임베딩)을 활용하는 추론 설정의 경우 캐시가 얼마나 자주 도움이 되는지 측정합니다. 낮은 적중률은 최적화되지 않은 캐시 구성 또는 워크로드 변경을 나타낼 수 있으며, 둘 다 지연 시간을 증가시킬 수 있습니다.

이후 주제에서는 위에서 언급한 몇 가지 지표에 대한 데이터 수집을 보여줍니다. 두 가지 AWS 권장 접근 방식인 AWS 네이티브 CloudWatch Container Insights와 Amazon Managed Grafana를 사용하는 오픈 소스 Amazon Managed Prometheus의 예를 제공합니다. https://docs.aws.amazon.com/grafana/latest/userguide/getting-started-with-AMG.html 전반적인 관찰성 요구 사항에 따라 이러한 솔루션 중 하나를 선택합니다. Container Insights 지표의 전체 목록은 Amazon EKS 및 Kubernetes Container Insights 지표를 참조하세요.

GPU 메모리 사용량 추적

the section called “코어 훈련 및 미세 조정 지표 모니터링 고려” 주제에서 설명한 대로 out-of-memory(OOM) 오류를 방지하고 효율적인 리소스 사용률을 보장하려면 GPU 메모리 사용이 필수적입니다. 다음 예제에서는 사용자 지정 히스토그램 지표를 노출하도록 훈련 애플리케이션을 계측`gpu_memory_usage_bytes`하고 P95 메모리 사용량을 계산하여 최대 소비를 식별하는 방법을 보여줍니다. 스테이징 환경에서 샘플 훈련 작업(예: 변환기 모델 미세 조정)으로 테스트해야 합니다.

AWS-Native CloudWatch Container Insights 예제

이 샘플은 AWS 네이티브 접근 방식을 사용하여를 히스토그램`gpu_memory_usage_bytes`으로 노출하도록 훈련 애플리케이션을 계측하는 방법을 보여줍니다. CloudWatch 임베디드 지표 형식(EMF) 형식으로 구조화된 로그를 내보내도록 AI/ML 컨테이너를 구성해야 합니다. CloudWatch 로그는 EMF를

구문 분석하고 지표를 게시합니다. 훈련 애플리케이션에서 aws_embedded_metrics를 사용하여 EMF 형식의 구조화된 로그를 CloudWatch Logs로 전송하여 GPU 지표를 추출합니다.

```
from aws_embedded_metrics import metric_scope
import torch
import numpy as np

memory_usage = []

@metric_scope
def log_gpu_memory(metrics):
    # Record current GPU memory usage
    mem = torch.cuda.memory_allocated()
    memory_usage.append(mem)

    # Log as histogram metric
    metrics.set_namespace("MLTraining/GPUMemory")
    metrics.put_metric("gpu_memory_usage_bytes", mem, "Bytes", "Histogram")

    # Calculate and log P95 if we have enough data points
    if len(memory_usage) >= 10:
        p95 = np.percentile(memory_usage, 95)
        metrics.put_metric("gpu_memory_p95_bytes", p95, "Bytes")
        print(f"Current memory: {mem} bytes, P95: {p95} bytes")

# Example usage in training loop
for epoch in range(20):
    # Your model training code would go here
    log_gpu_memory()
```

Prometheus 및 Grafana 예제

이 샘플은 Python의 Prometheus 클라이언트 라이브러리를 사용하여 훈련 애플리케이션을 히스토그램gpu_memory_usage_bytes`으로 노출하도록 계측하는 방법을 보여줍니다.

```
from prometheus_client import Histogram
from prometheus_client import start_http_server
import pynvml

start_http_server(8080)
memory_usage = Histogram(
    'gpu_memory_usage_bytes',
    'GPU memory usage during training',
```

```
    ['gpu_index'],
    buckets=[1e9, 2e9, 4e9, 8e9, 16e9, 32e9]
)

# Function to get GPU memory usage
def get_gpu_memory_usage():
    if torch.cuda.is_available():
        # Get the current GPU device
        device = torch.cuda.current_device()

        # Get memory usage in bytes
        memory_allocated = torch.cuda.memory_allocated(device)
        memory_reserved = torch.cuda.memory_reserved(device)

        # Total memory usage (allocated + reserved)
        total_memory = memory_allocated + memory_reserved

        return device, total_memory
    else:
        return None, 0

# Get GPU memory usage
gpu_index, memory_used = get_gpu_memory_usage()
```

실시간 온라인 추론에 대한 추론 요청 기간 추적

the section called “코어 훈련 및 미세 조정 지표 모니터링 고려” 주제에서 설명한 것처럼 사용자 또는 기타 종속 시스템에 적시에 응답을 제공하려면 지연 시간이 짧아야 합니다. 다음 예제에서는 추론 애플리케이션에서 노출된 사용자 지정 히스토그램 지표 `inference_request_duration_seconds`를 추적하는 방법을 보여줍니다. 최악의 시나리오에 초점을 맞추기 위해 95번째 백분위수(P95) 지연 시간을 계산하고, 스테이징 환경에서 합성 추론 요청(예: Locust를 통해)으로 테스트하고, SLA 위반을 감지하기 위해 알림 임계값(예: >500ms)을 설정합니다.

AWS-Native CloudWatch Container Insights 예제

이 샘플은 AWS CloudWatch 임베디드 지표 형식을 사용하여 추론 애플리케이션에서 추론 _request_duration_seconds에 대한 사용자 지정 히스토그램 지표를 생성하는 방법을 보여줍니다.

```
import boto3
import time
from aws_embedded_metrics import metric_scope, MetricsLogger
```

```
cloudwatch = boto3.client('cloudwatch')

@metric_scope
def log_inference_duration(metrics: MetricsLogger, duration: float):
    metrics.set_namespace("ML/Inference")
    metrics.put_metric("inference_request_duration_seconds", duration, "Seconds",
 "Histogram")
    metrics.set_property("Buckets", [0.1, 0.5, 1, 2, 5])

@metric_scope
def process_inference_request(metrics: MetricsLogger):
    start_time = time.time()

    # Your inference processing code here
    # For example:
    # result = model.predict(input_data)

    duration = time.time() - start_time
    log_inference_duration(metrics, duration)

    print(f"Inference request processed in {duration} seconds")

# Example usage
process_inference_request()
```

Prometheus 및 Grafana 예제

이 샘플은 Python의 Prometheus 클라이언트 라이브러리를 사용하여 추론 애플리케이션에서 추론 _request_duration_seconds에 대한 사용자 지정 히스토그램 지표를 생성하는 방법을 보여줍니다.

```
from prometheus_client import Histogram
from prometheus_client import start_http_server
import time

start_http_server(8080)
request_duration = Histogram(
    'inference_request_duration_seconds',
    'Inference request latency',
    buckets=[0.1, 0.5, 1, 2, 5]
)
start_time = time.time()
# Process inference request
request_duration.observe(time.time() - start_time)
```

Grafana에서 쿼리를 사용하여 P95 지연 시간 추세`histogram_quantile(0.95, sum(rate(inference_request_duration_seconds_bucket[5m])) by (le, pod))`를 시각화합니다. 자세한 내용은 Prometheus 히스토그램 설명서 및 Prometheus 클라이언트 설명서를 참조하세요.

실시간 온라인 추론을 위한 토큰 처리량 추적

the section called "코어 훈련 및 미세 조정 지표 모니터링 고려" 주제에서 설명한 대로 토큰 처리 시간을 모니터링하여 모델 성능을 측정하고 조정 결정을 최적화하는 것이 좋습니다. 다음 예제에서는 추론 애플리케이션에서 노출된 사용자 지정 히스토그램 지표 `token_processing_duration_seconds`를 추적하는 방법을 보여줍니다. 95번째 백분위수(P95) 지속 시간을 계산하여 처리 효율성을 분석하고, 비프로덕션 클러스터에서 시뮬레이션된 요청 로드(예: 초당 100~1000개의 요청)로 테스트하고, 조정을 최적화하도록 KEDA 트리거를 조정합니다.

AWS-Native CloudWatch Container Insights 예제

이 샘플은 AWS CloudWatch 임베디드 지표 형식을 사용하여 token_processing_duration_seconds에 대한 추론 애플리케이션에서 사용자 지정 히스토그램 지표를 생성하는 방법을 보여줍니다. 차원(`set_dimension) with a custom `get_duration_bucket 함수)을 사용하여 기간을 버킷으로 분류합니다(예: """0.01", ">1").

```
import boto3
import time
from aws_embedded_metrics import metric_scope, MetricsLogger

cloudwatch = boto3.client('cloudwatch')

@metric_scope
def log_token_processing(metrics: MetricsLogger, duration: float, token_count: int):
    metrics.set_namespace("ML/TokenProcessing")
    metrics.put_metric("token_processing_duration_seconds", duration, "Seconds")
    metrics.set_dimension("ProcessingBucket", get_duration_bucket(duration))
    metrics.set_property("TokenCount", token_count)

def get_duration_bucket(duration):
    buckets = [0.01, 0.05, 0.1, 0.5, 1]
    for bucket in buckets:
        if duration <= bucket:
            return f"<={bucket}"
    return f">{buckets[-1]}"

@metric_scope
```

```
def process_tokens(input_text: str, model, tokenizer, metrics: MetricsLogger):
    tokens = tokenizer.encode(input_text)
    token_count = len(tokens)

    start_time = time.time()
    # Process tokens (replace with your actual processing logic)
    output = model(tokens)
    duration = time.time() - start_time

    log_token_processing(metrics, duration, token_count)
    print(f"Processed {token_count} tokens in {duration} seconds")
    return output
```

Prometheus 및 Grafana 예제

이 샘플은 Python의 Prometheus 클라이언트 라이브러리를 사용하여 token_processing_duration_seconds에 대한 추론 애플리케이션에서 사용자 지정 히스토그램 지표를 생성하는 방법을 보여줍니다.

```
from prometheus_client import Histogram
from prometheus_client import start_http_server
import time

start_http_server(8080)
token_duration = Histogram(
    'token_processing_duration_seconds',
    'Token processing time per request',
    buckets=[0.01, 0.05, 0.1, 0.5, 1]
)
start_time = time.time()
# Process tokens
token_duration.observe(time.time() - start_time)
```

Grafana에서 쿼리를 사용하여 P95 처리 시간 추세`histogram_quantile(0.95, sum(rate(token_processing_duration_seconds_bucket[5m])) by (le, pod))`를 시각화합니다. 자세한 내용은 Prometheus 히스토그램 설명서 및 Prometheus 클라이언트 설명서를 참조하세요.

체크포인트 복원 지연 시간 측정

the section called “코어 훈련 및 미세 조정 지표 모니터링 고려” 주제에서 설명한 것처럼 체크포인트 지연 시간은 모델 수명 주기의 여러 단계에서 중요한 지표입니다. 다음 예제에서는 애플리케이션에서

checkpoint_restore_duration_seconds`노출되는 사용자 지정 히스토그램 지표를 추적하는 방법을 보여줍니다. 95번째 백분위수(P95) 지속 시간을 계산하여 복원 성능을 모니터링하고, 비프로덕션 클러스터에서 스팟 중단을 테스트하고, 알림 임계값(예: <30초)을 설정하여 지연을 감지합니다.

AWS-Native CloudWatch Container Insights 예제

이 샘플은 CloudWatch Insights를 사용하여 checkpoint_restore_duration_seconds를 히스토그램으로 노출하도록 배치 애플리케이션을 계측하는 방법을 보여줍니다.

```
import boto3
import time
import torch
from aws_embedded_metrics import metric_scope, MetricsLogger

@metric_scope
def log_checkpoint_restore(metrics: MetricsLogger, duration: float):
    metrics.set_namespace("ML/ModelOperations")
    metrics.put_metric("checkpoint_restore_duration_seconds", duration, "Seconds",
 "Histogram")
    metrics.set_property("Buckets", [1, 5, 10, 30, 60])
    metrics.set_property("CheckpointSource", "s3://my-bucket/checkpoint.pt")

@metric_scope
def load_checkpoint(model, checkpoint_path: str, metrics: MetricsLogger):
    start_time = time.time()

    # Load model checkpoint
    model.load_state_dict(torch.load(checkpoint_path))

    duration = time.time() - start_time
    log_checkpoint_restore(metrics, duration)

    print(f"Checkpoint restored in {duration} seconds")
```

Prometheus 및 Grafana 예제

이 샘플은 Python의 Prometheus 클라이언트 라이브러리를 사용하여 배치 애플리케이션을 히스토그램checkpoint_restore_duration_seconds으로 노출하도록 계측하는 방법을 보여줍니다.

```
from prometheus_client import Histogram
from prometheus_client import start_http_server
import torch
```

```
start_http_server(8080)
restore_duration = Histogram(
    'checkpoint_restore_duration_seconds',
    'Time to restore checkpoint',
    buckets=[1, 5, 10, 30, 60]
)
with restore_duration.time():
    model.load_state_dict(torch.load("s3://my-bucket/checkpoint.pt"))
```

Grafana에서 쿼리를 사용하여 P95 복원 지연 시간 추세`histogram_quantile(0.95, sum(rate(checkpoint_restore_duration_seconds_bucket[5m]) by (le))`를 시각화합니다. 자세한 내용은 Prometheus 히스토그램 설명서 및 Prometheus 클라이언트 설명서를 참조하세요.

애플리케이션 조정 및 성능

ML 아티팩트, 서비스 프레임워크 및 시작 최적화 관리

Amazon EKS에 기계 학습(ML) 모델을 배포하려면 모델을 컨테이너 이미지 및 런타임 환경에 통합하는 방법을 신중하게 고려해야 합니다. 이렇게 하면 확장성, 재현성 및 효율적인 리소스 사용률이 보장됩니다. 이 주제에서는 ML 모델 아티팩트를 처리하고, 프레임워크 제공을 선택하고, 사전 캐싱과 같은 기술을 통해 컨테이너 시작 시간을 최적화하는 다양한 접근 방식을 설명합니다.이 모든 방법은 컨테이너 시작 시간을 줄이도록 조정되었습니다.

배포에서 ML 모델 아티팩트 처리

핵심 결정은 ML 모델 아티팩트(예: 가중치 및 구성)를 직접 처리하는 방법입니다. 선택 사항은 이미지 크기, 배포 속도, 모델 업데이트 빈도 및 운영 오버헤드에 영향을 미칩니다. "모델" 저장을 참조할 때는 모델 아티팩트(예: 훈련된 파라미터 및 모델 가중치)를 참조합니다. Amazon EKS에서 ML 모델 아티팩트를 처리하는 방법에는 여러 가지가 있습니다. 각 에는 장단점이 있으며, 최상의 장단점은 모델의 크기, 업데이트 주기 및 인프라 요구 사항에 따라 달라집니다. 최소 권장 방법부터 가장 권장하는 방법까지 다음 접근 방식을 고려하세요.

- 모델을 컨테이너 이미지로 베이킹: 이미지 빌드 중에 모델 파일(예: .safetensors, .pth, .h5)을 컨테이너 이미지(예: Dockerfile)에 복사합니다. 모델은 변경 불가능한 이미지의 일부입니다. 업데이트 빈도가 낮은 소형 모델에는이 접근 방식을 사용하는 것이 좋습니다. 이렇게 하면 일관성과 재현성이 보장되고, 로드 지연이 없으며, 종속성 관리를 간소화할 수 있지만 이미지 크기가 커져 빌드 및 푸시 속도가 느려지고, 모델 업데이트를 위해 재구축 및 재배포가 필요하며, 레지스트리 풀 처리량으로 인해 대규모 모델에 적합하지 않습니다.

- 런타임 시 모델 다운로드: 컨테이너 시작 시 애플리케이션은 init 컨테이너 또는 진입점의 스크립트를 통해 외부 스토리지(예: Mountpoint for S3 CSI 드라이버, Amazon S3 S3)에서 모델을 다운로드합니다. S3 업데이트가 빈번한 대규모 모델의 경우이 접근 방식부터 시작하는 것이 좋습니다. 이렇게 하면 컨테이너 이미지가 코드/런타임에 초점을 맞추고, 재구축 없이 모델을 쉽게 업데이트할 수 있으며, 스토리지 메타데이터를 통한 버전 관리를 지원하지만, 인증 및 캐싱이 필요한 잠재적 네트워크 장애(재시도 로직 필요)가 발생합니다.

자세한 내용은 AI on EKS 워크숍의 풀 프로세스 가속화를 참조하세요.

ML 모델 제공

Amazon EKS에서 기계 학습(ML) 모델을 배포하고 제공하려면 지연 시간, 처리량, 확장성 및 운영 간소화를 최적화할 수 있는 적절한 모델 제공 접근 방식을 선택해야 합니다. 선택은 모델 유형(예: 언어, 비전 모델), 워크로드 요구 사항(예: 실시간 추론) 및 팀 전문 지식에 따라 달라집니다. 일반적인 접근 방식에는 프로토타이핑을 위한 Python 기반 설정, 프로덕션급 기능을 위한 전용 모델 서버, 고성능 및 효율성을 위한 특수 추론 엔진이 포함됩니다. 각 방법에는 설정 복잡성, 성능 및 리소스 사용률의 장단점이 포함됩니다. 제공 프레임워크는 종속성으로 인해 컨테이너 이미지 크기(여러 GBs 늘릴 수 있으며, 이는 시작 시간에 영향을 미칠 수 있습니다. 이를 완화하기 위해 아티팩트 처리 기술을 사용한 분리를 고려하세요. 옵션은 최소 권장 사항부터 가장 권장 사항까지 나열됩니다.

Python 프레임워크 사용(예: FastAPI, PyTorch를 사용하는 HuggingFace 변환기) 컨테이너화된 노드 설정 내에 모델 파일(가중치, 구성, 토큰화기)을 포함하는 Python 프레임워크를 사용하여 사용자 지정 애플리케이션을 개발합니다.

- 장점: 간편한 프로토타이핑, 추가 인프라 없이 Python 전용, 모든 HuggingFace 모델과 호환, 간단한 Kubernetes 배포.
- 단점: 단일 요청/단순 일괄 처리, 느린 토큰 생성(최적화된 커널 없음), 메모리 비효율성, 규모 조정/모니터링 부족, 긴 시작 시간으로 제한합니다.
- 권장 사항: 사용자 지정 로직 통합이 필요한 초기 프로토타이핑 또는 단일 노드 작업에 사용합니다.

전용 모델 서비스 프레임워크(예: TensorRT-LLM, TGI) 사용 ML 추론을 위해 TensorRT-LLM 또는 TGI와 같은 특수 서버를 채택하고 모델 로드, 라우팅 및 최적화를 관리합니다. 이러한는 선택 사항인 컴파일 또는 플러그인과 함께 Safetensor와 같은 형식을 지원합니다.

- 장점: 일괄 처리(정적/진행 중 또는 연속), 양자화(INT8, FP8, GPTQ), 하드웨어 최적화(NVIDIA, AMD, Intel, Inferentia) 및 다중 GPU 지원(Tensor/Pipeline Parallelism)을 제공합니다. TensorRT-

LLM은 다양한 모델(LLMs, Encoder-Decoder)을 지원하는 반면, TGI는 HuggingFace 통합을 활용합니다.

- 단점: TensorRT-LLM은 컴파일이 필요하며 NVIDIA 전용입니다. TGI는 일괄 처리에서 덜 효율적일 수 있습니다. 둘 다 구성 오버헤드를 추가하고 일부 모델 유형(예: 비변환기)에 맞지 않을 수 있습니다.
- 권장 사항: A/B 테스트 또는 호환 하드웨어를 사용한 높은 처리량과 같은 프로덕션 기능이 필요한 PyTorch/TensorFlow 모델에 적합합니다.

특수한 고처리량 추론 엔진(예: vLLM) 사용 vLLM과 같은 고급 추론 엔진을 활용하여 PagedAttention을 사용한 LLM 서비스, 진행 중인 배치 및 양자화(INT8, FP8-KV, AWQ)를 최적화하고 EKS Auto Scaling과 통합 가능합니다.

- 장점: 높은 처리량 및 메모리 효율성(40~60% VRAM 절감), 동적 요청 처리, 토큰 스트리밍, 단일 노드 Tensor 병렬 다중 GPU 지원, 광범위한 하드웨어 호환성.
- 단점: 디코더 전용 변환기(예: LLaMA)에 최적화되어 있고 비변환기 모델에 덜 효과적이며 호환되는 하드웨어(예: NVIDIA GPUs) 및 설정 작업이 필요합니다.
- 권장 사항: EKS에서 지연 시간이 짧은 대용량 LLM 추론을 위한 최고의 선택으로 확장성과 성능을 극대화합니다.

컨테이너 이미지 풀 타임 최적화

컨테이너 이미지가 크면 콜드 스타트 지연이 발생하여 포드 시작 지연 시간에 영향을 미칠 수 있습니다. 가로로 확장되는 실시간 추론 워크로드와 같이 지연 시간에 민감한 워크로드의 경우 빠른 포드 시작이 중요합니다. 컨테이너 이미지 풀 타임을 최적화하려면 다음 접근 방식을 고려하세요.

컨테이너 이미지 크기 줄이기

시작 시 컨테이너 이미지 크기를 줄이는 것은 이미지를 더 작게 만드는 또 다른 방법입니다. 컨테이너 이미지 빌드 프로세스의 모든 단계에서 줄일 수 있습니다. 시작하려면 필요한 종속성 수가 가장 적은 기본 이미지를 선택합니다. 이미지 빌드 중에는 필요한 필수 라이브러리와 아티팩트만 포함합니다. 이미지를 빌드할 때 여러 `RUN` 또는 `COPY` 명령을 결합하여 더 많은 계층을 생성해 보세요. AI/ML 프레임워크의 경우 다단계 빌드를 사용하여 빌드와 런타임을 분리하고, 필요한 아티팩트만 복사하고(예: 레지스트리 또는 로컬 컨텍스트`COPY –from=`의 경우를 통해), 런타임 전용 이미지와 같은 변형을 선택합니다(예: 3.03GB`pytorch/pytorch:2.7.1-cuda11.8-cudnn9-runtime`에서 대 6.66GB에서 개발). 자세한 내용은 AI on EKS 워크숍의 컨테이너 이미지 크기 줄이기를 참조하세요.

SOCI 스냅샷 생성기를 사용하여 이미지 미리 가져오기

쉽게 최소화할 수 없는 매우 큰 이미지의 경우 병렬 풀 및 언패킹 모드로 구성된 오픈 소스 SOCI(Sekable OCI) 스냅샷 생성기를 사용할 수 있습니다. 이 솔루션을 사용하면 빌드 파이프라인을 재구축하거나 수정하지 않고도 기존 이미지를 사용할 수 있습니다. 이 옵션은 매우 큰 이미지가 있는 워크로드를 고성능 EC2 컴퓨팅 인스턴스에 배포할 때 특히 효과적입니다. 대규모 AI/ML 워크로드와 마찬가지로 처리량이 많은 네트워킹 및 고성능 스토리지 구성에서도 잘 작동합니다.

SOCI 병렬 풀/언팩 모드는 구성 가능한 병렬화 전략을 통해 end-to-end 이미지 풀 성능을 개선합니다. 이미지 가져오기 및 준비 속도가 빨라지면 새 워크로드를 배포하고 클러스터를 효율적으로 확장할 수 있는 속도에 직접적인 영향을 미칩니다. 이미지 풀에는 두 가지 주요 단계가 있습니다.

1: 레지스트리에서 노드로 계층 가져오기

계층 가져오기 최적화를 위해 SOCI는 계층당 여러 동시 HTTP 연결을 생성하여 단일 연결 제한을 초과하는 다운로드 처리량을 곱합니다. 큰 계층을 청크로 분할하고 여러 연결에서 동시에 다운로드합니다. 이 접근 방식은 사용 가능한 네트워크 대역폭을 포화시키고 다운로드 시간을 크게 줄이는 데 도움이 됩니다. 이는 단일 계층이 몇 기가바이트일 수 있는 AI/ML 워크로드에 특히 유용합니다.

2. 컨테이너를 생성하기 위해 해당 계층의 압축을 풀고 준비

계층 언패킹 최적화를 위해 SOCI는 여러 계층을 동시에 처리합니다. 다음 단계를 시작하기 전에 각 계층의 압축이 완전히 풀릴 때까지 기다리는 대신 사용 가능한 CPU 코어를 사용하여 여러 계층의 압축을 풀고 동시에 추출합니다. 이 병렬 처리는 기존의 I/O 바운드 언패킹 단계를 사용 가능한 코어에 따라 확장되는 CPU 최적화 작업으로 변환합니다. 시스템은이 병렬화를 신중하게 오케스트레이션하여 파일 시스템 일관성을 유지하면서 처리량을 극대화합니다.

SOCI 병렬 풀 모드는 동시성 다운로드와 병렬성 압축 해제 모두에 대해 구성 가능한 파라미터가 있는 이중 임계값 제어 시스템을 사용합니다. 이 세분화된 제어를 사용하면 특정 성능 요구 사항 및 환경 조건을 충족하도록 SOCI의 동작을 미세 조정할 수 있습니다. 이러한 파라미터를 이해하면 최상의 풀 성능을 위해 런타임을 최적화하는 데 도움이 됩니다.

참조

- 솔루션 및 튜닝 장단점에 대한 자세한 내용은 GitHub의 SOCI 프로젝트 리포지토리에 있는 기능 설명서를 참조하세요.
- Amazon EKS에서 Karpenter를 사용하는 실습 예제는 SOCI 스냅샷 생성기 병렬 풀/언팩 모드를 사용하는 Karpenter 블루프린트를 참조하세요.

- 병렬 풀을 위한 Bottlerocket 구성에 대한 자세한 내용은 Bottlerocket 설명서의 soci-snapshotter 병렬 풀 언팩 모드를 참조하세요.o

EBS 스냅샷을 사용하여 이미지 미리 가져오기

캐시된 컨테이너 이미지의 Amazon Elastic Block Store(EBS) 스냅샷을 만들고이 스냅샷을 EKS 작업자 노드에 재사용할 수 있습니다. 이렇게 하면 노드 시작 시 로컬에서 이미지를 미리 가져와 포드 초기화 시간을 줄일 수 있습니다. 관리형 노드 그룹용 Karpenter 및 EKS Terraform 블루프린트를 사용한 자세한 내용은 Bottlerocket 데이터 볼륨을 사용하여 Amazon EKS에서 컨테이너 시작 시간 단축을 참조하세요. https://aws-ia.github.io/terraform-aws-eks-blueprints/patterns/machine-learning/ml-container-cache/

자세한 내용은 AI on EKS 워크숍의 컨테이너 스냅샷 생성기 사용 및 EBS 스냅샷을 사용하여 Bottlerocket 데이터 볼륨에 컨테이너 이미지 사전 로드를 참조하세요. https://awslabs.github.io/ai-on-eks/docs/guidance/container-startup-time/accelerate-pull-process/prefecthing-images-on-br

컨테이너 런타임 캐시를 사용하여 이미지 미리 가져오기

Kubernetes 리소스(예: DaemonSet 또는 배포)를 사용하여 노드에 컨테이너 이미지를 미리 가져와 노드의 컨테이너 런타임 캐시를 채울 수 있습니다. 컨테이너 런타임 캐시는 컨테이너 런타임에서 관리하는 로컬 스토리지입니다(예: 레지스트리에서 가져온 후 이미지가 저장되는 컨테이너형 스토리지입니다. 사전 풀링을 통해 로컬에서 이미지를 사용할 수 있으므로 포드 시작 중에 다운로드 지연이 발생하지 않습니다. 이 접근 방식은 이미지가 자주 변경되거나(예: 자주 업데이트), EBS 스냅샷이 미리 구성되지 않거나, EBS 볼륨을 빌드하는 데 컨테이너 레지스트리에서 직접 가져오는 것보다 시간이 더 많이 걸리거나, 노드가 이미 클러스터에 있고 여러 가능한 이미지 중 하나를 사용하여 온디맨드로 포드를 가동해야 하는 경우에 특히 유용합니다.

모든 변형을 미리 풀링하면 필요한 이미지에 관계없이 빠르게 시작할 수 있습니다. 예를 들어 10가지 기술을 사용하여 구축된 100,000개의 소형 모델이 필요한 대량 병렬 ML 워크로드에서 대규모 클러스터(예: 수천 개의 노드)에서 DaemonSet를 통해 10개의 이미지를 미리 풀링하면 포드 시작 시간이 최소화되어 온디맨드 풀을 방지하여 10초 이내에 완료할 수 있습니다. 컨테이너 런타임 캐시 접근 방식을 사용하면 EBS 스냅샷을 관리할 필요가 없으며, 항상 DaemonSets를 사용하여 최신 컨테이너 이미지 버전을 얻을 수 있지만 노드가 스케일 인/아웃되는 실시간 추론 워크로드의 경우 Cluster Autoscaler와 같은 도구가 추가한 새 노드는 미리 풀링된 DaemonSet가 이미지 풀링을 완료하기 전에 워크로드 포드를 예약할 수 있습니다. 이로 인해 새 노드의 초기 포드가 풀을 트리거하여 시작이 지연되고 지연 시간이 짧은 요구 사항에 영향을 미칠 수 있습니다. 또한 kubelet 이미지 가비지 수집은 디스크 사용량이 특정 임계값을 초과하거나 구성된 최대 미사용 기간을 초과하는 경우 미사용 이미지를 제거하여 미리 풀링된 이미지에 영향을 미칠 수 있습니다. 스케일 인/아웃 패턴에서는 유휴 노드에서 이미지를 제거할

수 있으므로 후속 스케일 업 중에 다시 풀링하고 버스트 워크로드에 대한 캐시의 신뢰성을 줄여야 합니다.

컨테이너 런타임 캐시로 이미지를 미리 풀링하는 예제는 AWS GitHub 리포지토리를 참조하세요.

kubelet 및 컨테이너형 스토리지용 NVMe 고려

디스크 성능을 높이기 `containerd` 위해 임시 NVMe 인스턴스 스토리지 디스크를 사용하도록 `kubelet` 및를 구성하는 것이 좋습니다. 컨테이너 풀 프로세스에는 레지스트리에서 컨테이너 이미지를 다운로드하고 해당 계층을 사용 가능한 형식으로 압축 해제하는 작업이 포함됩니다. 압축 해제 중에 I/O 작업을 최적화하려면 컨테이너 호스트의 인스턴스 유형에 더 높은 수준의 I/O 성능과 처리량, 즉 로컬 스토리지가 있는 NVMe 지원 인스턴스와 EBS 볼륨 IOPS/처리량을 제공하는 항목을 평가해야 합니다. NVMe 로컬 스토리지가 있는 EC2 인스턴스의 경우 더 높은 수준의 I/O 성능과 처리량을 위해 임시 NVMe 인스턴스 스토리지 디스크를 사용하도록 노드의 기본 파일 시스템을 kubelet(`/var/lib/kubelet`), containerd(`/var/lib/containerd`) 및 Pod 로그(`/var/log/pods`)로 구성하는 것이 좋습니다.

노드에 다운로드되는 임시 스토리지 및 컨테이너 이미지를 요청하는 포드 간에 노드의 임시 스토리지를 공유할 수 있습니다. Bottlerocket 또는 AL2023 EKS 최적화 AMIs와 함께 Karpenter를 사용하는 경우 instanceStorePolicy를 RAID0으로 설정하거나 관리형 노드 그룹을 사용하는 경우 NodeConfig의 localStoragePolicy를 사용자 데이터의 일부로 설정하여 EC2NodeClass에서 구성할 수 있습니다. NodeConfig

이 가이드에 기여

누구나 모범 사례 가이드에 기여할 수 있습니다. EKS 모범 사례 가이드는 GitHub에서 AsciiDoc 형식으로 작성되었습니다.

기존 기여자에 대한 요약

- VS Code`bpg-docs.code-workspace`를 사용하여를 열어 AsciiDoc 확장을 자동으로 설치합니다.
  - Visual Studio Marketplace의 AsciiDoc 확장에 대해 자세히 알아보세요.
- AWS Docs 웹 사이트의 소스 파일은에 저장됩니다. `latest/bpg`
- 구문은 마크다운과 매우 유사합니다.
  - AsciiDoctor 문서의 구문 참조를 검토합니다.
- 문서 플랫폼은 만 배포합니다`latest/bpg/images`. 각 가이드 섹션에는이 디렉터리로 돌아가는 심볼 링크가 있습니다. 예를 들어는를 `latest/bpg/networking/images` 가리킵니다`latest/bpg/images`.

로컬 편집 환경 설정

가이드를 자주 편집하려는 경우 로컬 편집 환경을 설정합니다.

리포지토리 포크 및 복제

`git`, `github`및 텍스트 편집기에 익숙해야 합니다. `git` 및 시작하기에 대한 자세한 내용은 GitHub 문서의 GitHub 계정 시작하기를 `github`참조하세요. GitHub

1. GitHub에서 EKS 모범 사례 가이드를 확인하세요.
2. 프로젝트 리포지토리의 포크를 생성합니다. GitHub 문서에서 리포지토리를 포크하는 방법을 알아봅니다.
3. 프로젝트 리포지토리의 포크를 복제합니다. 포크된 리포지토리를 복제하는 방법을 알아봅니다.

VS Code Workspace 열기

AWS는 Microsoft의 Visual Studio 코드를 사용하여 가이드를 편집할 것을 권장합니다. VS Code에 대한 자세한 내용은 Visual Studio Code 설명서의 Visual Studio Code 다운로드 및 Visual Studio Code 시작하기를 참조하세요.

1. VS Code를 엽니다.
2. 복제된 리포지토리에서 `bpg-docs.code-workspace` 파일을 엽니다.
3. 이 워크스페이스를 처음 여는 경우 프롬프트를 수락하여 AsciiDoc 확장을 설치합니다. 이 확장은 AsciiDoc 파일의 구문을 확인하고 라이브 미리 보기를 생성합니다.
4. `latest/bpg` 디렉터리로 이동합니다. 이 디렉터리에는 AWS 설명서 사이트에 배포되는 소스 파일이 들어 있습니다. 소스 파일은 "보안" 또는 "네트워크"와 같은 가이드 섹션별로 구성됩니다.

파일 편집

1. 편집기에서 파일을 엽니다.
   - AsciiDoc 구문을 보고 제목, 링크 및 목록을 생성하는 방법을 알아봅니다.
   - 마크다운 구문을 사용하여 텍스트, 목록 및 제목의 형식을 지정할 수 있습니다. 마크다운 구문을 사용하여 링크를 생성할 수 없습니다.
2. 페이지의 라이브 미리 보기를 엽니다.
   - 먼저 `ctrl-k` 또는 `cmd-k` (키보드에 따라 다름)를 누릅니다. 둘째,를 누릅니다v. 그러면 분할 보기에서 미리 보기가 열립니다.

AWS는 기능 브랜치를 사용하여 변경 사항을 구성할 것을 제안합니다. git을 사용하여 브랜치를 생성하는 방법을 알아봅니다.

풀 요청 제출

GitHub 웹 사이트 또는 GitHub cli에서 풀 요청을 생성할 수 있습니다.

GitHub 웹 사이트를 사용하여 포크에서 풀 요청을 생성하는 방법을 알아봅니다.

GitHub cli를 사용하여 풀 요청을 생성하는 방법을 알아봅니다.

github.dev 웹 기반 편집기 사용

`github.dev` 웹 기반 편집기는 VS Code를 기반으로 합니다. 이는 설정 없이 여러 파일을 편집하고 콘텐츠를 미리 볼 수 있는 좋은 방법입니다.

AsciiDoc 확장을 지원합니다. GUI를 사용하여 git 작업을 수행할 수 있습니다. 웹 기반 편집기에는 명령을 실행하기 위한 셸 또는 터미널이 없습니다.

GitHub 계정이 있어야 합니다. 필요한 경우 로그인하라는 메시지가 표시됩니다.

GitHub 웹 기반 편집기를 시작합니다.

단일 페이지 편집

GitHub를 사용하여 개별 페이지를 빠르게 업데이트할 수 있습니다. 각 페이지에는 하단에 “#GitHub에서이 페이지 편집” 링크가 있습니다.

1. 이 가이드에서 편집하려는 페이지로 이동합니다.
2. 하단의 "GitHub에서이 페이지 편집" 링크를 클릭합니다.
3. GitHub 파일 뷰어의 오른쪽 상단에 있는 연필 편집 아이콘을 클릭하거나 e
4. 파일 편집
5. "변경 사항 커밋... " 버튼을 사용하여 변경 사항을 제출합니다. 이 버튼은 GitHub 풀 요청을 생성합니다. 가이드 유지 관리자는이 풀 요청을 검토합니다. 검토자가 풀 요청을 승인하거나 변경을 요청합니다.

페이지 ID 보기 및 설정

이 페이지에서는 페이지 ID를 보고 설정하는 방법을 설명합니다.

페이지 ID는 설명서 사이트의 각 페이지를 식별하는 고유한 문자열입니다. 특정 페이지에 있을 때 브라우저의 주소 표시줄에서 페이지 ID를 볼 수 있습니다. 페이지 ID는 URL, 파일 이름 및 상호 참조 링크를 생성하는 데 사용됩니다.

예를 들어이 페이지를 보는 경우 브라우저 주소 표시줄의 URL은 다음과 같습니다.

```
https://docs.aws.amazon.com/view-set-page-id.html
```

URL의 마지막 부분(`view-set-page-id`)은 페이지 ID입니다.

페이지 ID 설정

새 페이지를 생성할 때 소스 파일에서 페이지 ID를 설정해야 합니다. 페이지 ID는 페이지 콘텐츠를 설명하는 간결한 하이픈 문자열이어야 합니다.

1. 텍스트 편집기에서 새 페이지의 소스 파일을 엽니다.
2. 파일 상단에 다음 줄을 추가합니다. 첫 번째 제목 위에 있어야 합니다.

```
[#my-new-page]
```

를 새 페이지의 페이지 IDmy-new-page로 바꿉니다.

3. 파일을 저장합니다.

> **Note**
>
> 페이지 IDs 전체 설명서 사이트에서 고유해야 합니다. 기존 페이지 ID를 사용하려고 하면 빌드 오류가 발생합니다.

새 페이지 생성

새 페이지를 생성하고 가이드 목차를 업데이트하는 방법을 알아봅니다.

페이지 메타데이터 생성

1. 페이지 제목과 페이지 짧은 제목을 결정합니다. 페이지 짧은 제목은 선택 사항이지만 페이지 제목이 몇 단어 이상인 경우 권장됩니다.
2. 페이지의 ID를 확인합니다. EKS 모범 사례 가이드 내에서 고유해야 합니다. 규칙은 소문자를 모두 사용하고 로 단어를 구분하는 것입니다-.
3. 필요한 경우 폴더에 새 asciidoc 파일을 생성하고 파일에 다음 텍스트를 추가합니다.

   Example

   ["topic"] [#<page-id>] = <page-title> :info_titleabbrev: <page-short-title>

   예:

   Example

   ["topic"] [#scalability] = EKS Scalability 모범 사례: info_titleabbrev: Scalability

목차에 추가

1. 목차에서 상위 페이지의 파일을 엽니다. 새로운 최상위 가이드 섹션의 경우 상위 파일은 입니다book.adoc.
2. 상위 파일 하단에서 다음 지시문을 업데이트하고 삽입합니다.

   Example

   포함::<new-filename>[leveloffset=+1]

   예를 들어,

   Example

   include::dataplane.adoc[leveloffset=+1]

이미지 삽입

1. 편집하려는 페이지의 이미지 접두사를 찾습니다. 파일 제목의 :imagesdir: 속성을 검토합니다. 예를 들어, `:imagesdir: images/reliability/
2. 와 같이이 경로에 이미지를 배치합니다. latest/bpg/images/reliability
3. 이미지에 적합한 alt-text를 결정합니다. 이미지에 대한 간략한 설명을 작성합니다. 예를 들어 "3개의 가용 영역이 있는 VPC의 다이어그램"은 적절한 alt-text입니다.
4. 다음 예제를 alt-text 및 이미지 파일 이름으로 업데이트합니다. 원하는 위치에를 삽입합니다.

   Example

   image::<image-filename>[<image-alt-text>]

   예:

   Example

   image::eks-data-plane-connectivity.jpeg[네트워크 다이어그램]

Vale을 사용하여 스타일 확인

1. Vale CLI를 설치합니다.

2. `vale sync` 실행
3. Visual Studio Marketplace에서 Vale 확장을 설치합니다.
4. VS Code를 다시 시작하고 AsciiDoc 파일을 엽니다.
5. VS 코드는 문제가 있는 텍스트를 강조 표시합니다. VS Code 문서에서 오류 및 경고를 사용하는 방법을 알아봅니다.

로컬 미리 보기 빌드

1. `brew` Linux 또는 MacOS에서를 사용하여 `asciidoctor` 도구 설치
   - AsciiDoctor 문서에 asciidoctor cli를 설치하는 방법을 알아봅니다. AsciiDoctor
   - brew 패키지 관리자를 설치하는 방법을 알아봅니다.
2. 터미널을 열고 로 이동합니다. `latest/bpg/`
3. `asciidoctor book.adoc` 실행
   - 구문 경고 및 오류 검토
4. `book.html` 출력 파일을 엽니다.
   - MacOS에서는를 실행`open book.html`하여 기본 브라우저에서 미리 보기를 열 수 있습니다.

AsciiDoc 치트 시트

기본 형식 지정

```
*bold text*
_italic text_
`monospace text`
```

헤더

```
= Document Title (Header 1)
== Header 2
=== Header 3
==== Header 4
===== Header 5
====== Header 6
```

Lists

정렬되지 않은 목록:

```
- Item 1
- Item 2
-- Subitem 2.1
-- Subitem 2.2
- Item 3
```

정렬된 목록:

```
. First item
. Second item
.. Subitem 2.1
.. Subitem 2.2
. Third item
```

링크

```
External link:  https://example.com[Link text]
Internal link: <<page-id>>
Internal link: xref:page-id[Link text]
```

이미지

```
image::image-file.jpg[Alt text]
```

코드 블록

```
 [source,python]
 ----
 def hello_world():
     print("Hello, World!")
 ----
```

테이블

기본 테이블을 빌드하는 방법을 알아봅니다.

```
[cols="1,1"]
|===
|Cell in column 1, row 1
|Cell in column 2, row 1

|Cell in column 1, row 2
|Cell in column 2, row 2

|Cell in column 1, row 3
|Cell in column 2, row 3
|===
```

주의 문구

```
NOTE: This is a note admonition.

WARNING: This is a warning admonition.

TIP: This is a tip admonition.

IMPORTANT: This is an important admonition.

CAUTION: This is a caution admonition.
```

미리 보기:

> **Note**
>
> 참고에 관한 주의 문구입니다.

포함 파일

```
include::filename.adoc[]
```

기계 번역으로 제공되는 번역입니다. 제공된 번역과 원본 영어의 내용이 상충하는 경우에는 영어 버전이 우선합니다.