다중 계정 DevOps 환경을 위한 Git 분기 전략 선택

AWS 권장 가이드

AWS 권장 가이드: 다중 계정 DevOps 환경을 위한 Git 분기 전략 선택

Table of Contents

다중 계정 DevOps 환경을 위한 Git 분기 전략 선택

Amazon Web Services(기여자)

2024년 2월(문서 기록)

클라우드 기반 접근 방식으로 전환하고에서 소프트웨어 솔루션을 제공하는 것은 혁신적일 AWS 수 있습니다. 소프트웨어 개발 수명 주기 프로세스를 변경해야 할 수 있습니다. 일반적으로의 개발 프로세스 중에 여러가 AWS 계정 사용됩니다 AWS 클라우드. DevOps 프로세스와 페어링할 호환 가능한 Git 분기 전략을 선택하는 것은 성공에 필수적입니다. 조직에 적합한 Git 분기 전략을 선택하면 개발 팀 간에 DevOps 표준과 모범 사례를 간결하게 전달하는 데 도움이 됩니다. Git 분기는 단일 환경에서 간단할 수 있지만 샌드박스, 개발, 테스트, 스테이징 및 프로덕션 환경과 같은 여러 환경에 적용할 경우 혼란스러울 수 있습니다. 여러 환경이 있으면 DevOps 구현의 복잡성이 증가합니다.

이 가이드에서는 조직이 다중 계정 DevOps 프로세스를 구현하는 방법을 보여주는 Git 분기 전략의 시각적 다이어그램을 제공합니다. 시각적 가이드는 팀이 Git 분기 전략을 DevOps 관행과 병합하는 방법을 이해하는 데 도움이 됩니다. 소스 코드 리포지토리를 관리하기 위해 Gitflow, GitHub Flow 또는 Trunk와 같은 표준 분기 모델을 사용하면 개발 팀이 작업을 조정하는 데 도움이 됩니다. 또한 이러한 팀은 인터넷에서 표준 Git 훈련 리소스를 사용하여 해당 모델과 전략을 이해하고 구현할 수 있습니다.

의 DevOps 모범 사례는 AWS Well-Architected의 DevOps 지침을 AWS참조하세요. 이 가이드를 검토할 때 실사를 사용하여 조직에 적합한 분기 전략을 선택합니다. 일부 전략은 사용 사례에 더 적합할 수 있습니다.

목표

이 가이드는 여러가 있는 조직의 DevOps 분기 전략을 선택하고 구현하는 방법에 대한 설명서 시리즈의 일부입니다 AWS 계정. 이 시리즈는 처음부터 요구 사항, 목표 및 모범 사례를 가장 잘 충족하는 전략을 적용하여에서 경험을 간소화하는 데 도움이 되도록 설계되었습니다 AWS 클라우드. 이 가이드에는 조직에서 사용하는 지속적 통합 및 지속적 전달(CI/CD) 엔진과 기술 프레임워크에 따라 달라지기 때문에 DevOps 실행 파일 스크립트가 포함되어 있지 않습니다.

이 가이드에서는 세 가지 일반적인 Git 분기 전략인 GitHub Flow, Gitflow 및 Trunk의 차이점을 설명합니다. 이 가이드의 권장 사항은 팀이 조직의 목표에 맞는 분기 전략을 식별하는 데 도움이 됩니다. 이 가이드를 검토한 후 조직의 분기 전략을 선택할 수 있습니다. 전략을 선택한 후 다음 패턴 중 하나를 사용하여 개발 팀과 함께 전략을 구현할 수 있습니다.

- 다중 계정 DevOps 환경을 위한 트렁크 분기 전략 구현

- 다중 계정 DevOps 환경을 위한 GitHub Flow 분기 전략 구현
- 다중 계정 DevOps 환경을 위한 Gitflow 분기 전략 구현

한 조직, 팀 또는 프로젝트에 적합한 것이 다른 조직, 팀 또는 프로젝트에 적합하지 않을 수 있다는 점에 유의해야 합니다. Git 분기 전략 중에서 선택하는 것은 팀 규모, 프로젝트 요구 사항, 협업, 통합 빈도 및 릴리스 관리 간의 원하는 균형 등 다양한 요인에 따라 달라집니다.

CI/CD 사례 사용

AWS 에서는 소프트웨어 릴리스 수명 주기를 자동화하는 프로세스인 지속적 통합 및 지속적 전달(CI/CD)을 구현할 것을 권장합니다. 기존에는 개발에서 프로덕션으로 새 코드를 가져오는 데 필요한 수동 DevOps 프로세스의 대부분 또는 전부를 자동화합니다. CI/CD 파이프라인에는 샌드박스, 개발, 테스트, 스테이징 및 프로덕션 환경이 포함됩니다. 각 환경에서 CI/CD 파이프라인은 코드를 배포하거나 테스트하는 데 필요한 모든 인프라를 프로비저닝합니다. CI/CD를 사용하면 개발 팀이 코드를 변경하여 자동으로 테스트하고 배포할 수 있습니다. CI/CD 파이프라인은 개발 팀을 위한 거버넌스 및 가드레일도 제공합니다. 기능 수락 및 배포를 위해 일관성, 표준, 모범 사례 및 최소 수락 수준을 적용합니다. 자세한 내용은 지속적 통합 및 지속적 전송 실행을 참조하세요 AWS.

이 가이드에서 설명하는 모든 분기 전략은 CI/CD 관행에 매우 적합합니다. CI/CD 파이프라인의 복잡성은 분기 전략의 복잡성에 따라 증가합니다. 예를 들어 Gitflow는이 가이드에서 설명하는 가장 복잡한 분기 전략입니다. 이 전략의 CI/CD 파이프라인에는 더 많은 단계(예: 규정 준수 이유)가 필요하며 여러 개의 동시 프로덕션 릴리스를 지원해야 합니다. 분기 전략의 복잡성이 증가함에 따라 CI/CD 사용도 더 중요해집니다. 이는 CI/CD가 개발자가 정의된 프로세스를 의도적으로 또는 의도하지 않게 우회하지 못하도록 하는 개발 팀을 위한 가드레일과 메커니즘을 설정하기 때문입니다.

AWS 는 CI/CD 파이프라인을 구축하는 데 도움이 되도록 설계된 개발자 서비스 제품군을 제공합니다. 예를 들어 AWS CodePipeline는 빠르고 안정적인 애플리케이션 및 인프라 업데이트를 위해 릴리스 파이프라인을 자동화하는 데 도움이 되는 완전관리형 지속적 제공 서비스입니다.는 소스 코드를 AWS CodeBuild컴파일하고 테스트를 실행하며 ready-to-deploy 있는 소프트웨어 패키지를 생성합니다. 자세한 내용은 AWS의 개발자 도구를 참조하세요.

DevOps 환경 이해

분기 전략을 이해하려면 각 환경에서 발생하는 목적과 활동을 이해해야 합니다. 여러 환경을 설정하면 개발 활동을 단계로 분리하고, 해당 활동을 모니터링하고, 승인되지 않은 기능의 의도하지 않은 릴리스를 방지하는 데 도움이 됩니다. 각 환경에 하나 이상의가 있을 수 AWS 계정 있습니다.

대부분의 조직에는 몇 가지 사용 환경이 요약되어 있습니다. 그러나 환경 수는 조직 및 소프트웨어 개발 정책에 따라 다를 수 있습니다. 이 설명서 시리즈에서는 개발 파이프라인을 포괄하는 다음과 같은 5가지 공통 환경이 있다고 가정합니다. 이러한 환경은 서로 다른 이름으로 호출될 수 있습니다.

- 샌드박스 - 개발자가 코드를 작성하고, 실수를 하고, 개념 증명 작업을 수행하는 환경입니다.
- 개발 - 개발자가 코드를 통합하여 모든 것이 하나의 응집력 있는 애플리케이션으로 작동하는지 확인하는 환경입니다.
- 테스트 - QA 팀 또는 수락 테스트가 수행되는 환경입니다. 팀은이 환경에서 성능 또는 통합 테스트를 수행하는 경우가 많습니다.
- 스테이징 - 프로덕션과 동등한 상황에서 코드와 인프라가 예상대로 작동하는지 검증하는 사전 프로덕션 환경입니다. 이 환경은 프로덕션 환경과 최대한 비슷하도록 구성되어 있습니다.
- 프로덕션 - 최종 사용자 및 고객의 트래픽을 처리하는 환경입니다.

이 섹션에서는 각 환경에 대해 자세히 설명합니다. 또한 다음 단계로 진행할 수 있도록 각 환경의 빌드 단계, 배포 단계 및 종료 기준에 대해서도 설명합니다. 다음 이미지는 이러한 환경을 순서대로 보여줍니다.

이 섹션의 주제:

샌드박스 환경

샌드박스 환경은 개발자가 코드를 작성하고, 실수를 하고, 개념 증명 작업을 수행하는 곳입니다. 로컬 워크스테이션에서 또는 로컬 워크스테이션의 스크립트를 통해 샌드박스 환경에 배포할 수 있습니다.

액세스

개발자는 샌드박스 환경에 대한 전체 액세스 권한을 가져야 합니다.

빌드 단계

개발자는 샌드박스 환경에 변경 사항을 배포할 준비가 되면 로컬 워크스테이션에서 빌드를 수동으로 실행합니다.

1. git-secrets(GitHub)를 사용하여 민감한 정보 스캔
2. 소스 코드 린트
3. 해당하는 경우 소스 코드 빌드 및 컴파일
4. 단위 테스트 수행
5. 코드 적용 범위 분석 수행
6. 정적 코드 분석 수행
7. 코드형 인프라 구축(IaC)
8. IaC 보안 분석 수행
9. 오픈 소스 라이선스 추출
10. 빌드 아티팩트 게시

배포 단계

Gitflow 또는 Trunk 모델을 사용하는 경우 샌드박스 환경에 `feature`브랜치가 성공적으로 빌드되면 배포 단계가 자동으로 시작됩니다. GitHub Flow 모델을 사용하는 경우 다음 배포 단계를 수동으로 수행합니다. 다음은 샌드박스 환경의 배포 단계입니다.

1. 게시된 아티팩트 다운로드
2. 데이터베이스 버전 관리 수행
3. IaC 배포 수행

4. 통합 테스트 수행

개발 환경으로 이동하기 전 기대 사항

- 샌드박스 환경에서 `feature`브랜치 빌드 성공
- 개발자가 샌드박스 환경에서 기능을 수동으로 배포하고 테스트했습니다.

개발 환경

개발 환경은 개발자가 코드를 통합하여 모든 것이 하나의 응집력 있는 애플리케이션으로 작동하도록 하는 곳입니다. Gitflow에서 개발 환경에는 병합 요청에 포함된 최신 기능이 포함되어 있으며 릴리스할 준비가 되었습니다. GitHub Flow 및 Trunk 전략에서 개발 환경은 테스트 환경으로 간주되며 코드 베이스는 불안정하고 프로덕션에 배포하기에 적합하지 않을 수 있습니다.

액세스

최소 권한 원칙에 따라 권한을 할당합니다. 최소 권한은 작업을 수행하는 데 필요한 최소 권한을 부여하는 보안 모범 사례입니다. 개발자는 샌드박스 환경보다 개발 환경에 대한 액세스 권한이 적어야 합니다.

빌드 단계

`develop` 브랜치(Gitflow) 또는 브`main`랜치(Trunk 또는 GitHub Flow)에 대한 병합 요청을 생성하면 빌드가 자동으로 시작됩니다.

1. git-secrets(GitHub)를 사용하여 민감한 정보 스캔
2. 소스 코드 린트
3. 해당하는 경우 소스 코드 빌드 및 컴파일
4. 단위 테스트 수행
5. 코드 적용 범위 분석 수행
6. 정적 코드 분석 수행
7. 빌드 IaC
8. IaC 보안 분석 수행
9. 오픈 소스 라이선스 추출

배포 단계

Gitflow 모델을 사용하는 경우 개발 환경에 develop브랜치가 성공적으로 빌드되면 배포 단계가 자동으로 시작됩니다. GitHub Flow 모델 또는 Trunk 모델을 사용하는 경우 main브랜치에 대해 병합 요청이 생성되면 배포 단계가 자동으로 시작됩니다. 다음은 개발 환경의 배포 단계입니다.

1. 빌드 단계에서 게시된 아티팩트 다운로드
2. 데이터베이스 버전 관리 수행
3. IaC 배포 수행
4. 통합 테스트 수행

테스트 환경으로 이동하기 전 기대 사항

- 개발 환경에서 develop브랜치(Gitflow) 또는 브main랜치(Trunk 또는 GitHub Flow)의 성공적인 빌드 및 배포
- 100%에서 단위 테스트 통과
- 성공적인 IaC 빌드
- 배포 아티팩트가 성공적으로 생성되었습니다.
- 개발자가 수동 확인을 수행하여 기능이 예상대로 작동하는지 확인했습니다.

테스트 환경

품질 보증(QA) 담당자는 테스트 환경을 사용하여 기능을 검증합니다. 테스트를 완료한 후 변경 사항을 승인합니다. 승인하면 브랜치는 다음 환경인 스테이징으로 이동합니다. Gitflow에서이 환경 및 그 이상의 다른 환경은 release브랜치에서만 배포할 수 있습니다. release 브랜치는 계획된 기능이 포함된 develop브랜치를 기반으로 합니다.

액세스

최소 권한 원칙에 따라 권한을 할당합니다. 개발자는 개발 환경보다 테스트 환경에 대한 액세스 권한이 적어야 합니다. QA 담당자는 기능을 테스트할 수 있는 충분한 권한이 필요합니다.

빌드 단계

이 환경의 빌드 프로세스는 Gitflow 전략을 사용할 때 버그 수정에만 적용됩니다. bugfix 브랜치에 대한 병합 요청을 생성하면 빌드가 자동으로 시작됩니다.

1. git-secrets(GitHub)를 사용하여 민감한 정보 스캔
2. 소스 코드 린트
3. 해당하는 경우 소스 코드 빌드 및 컴파일
4. 단위 테스트 수행
5. 코드 적용 범위 분석 수행
6. 정적 코드 분석 수행
7. 빌드 IaC
8. IaC 보안 분석 수행
9. 오픈 소스 라이선스 추출

배포 단계

개발 환경에 배포한 후 테스트 환경에서 `release`브랜치(Gitflow) 또는 `main`브랜치(Trunk 또는 GitHub Flow)의 배포를 자동으로 시작합니다. 다음은 테스트 환경의 배포 단계입니다.

1. 테스트 환경에 `release`브랜치(Gitflow) 또는 `main`브랜치(Trunk 또는 GitHub Flow) 배포
2. 지정된 직원의 수동 승인을 위해 일시 중지
3. 게시된 아티팩트 다운로드
4. 데이터베이스 버전 관리 수행
5. IaC 배포 수행
6. 통합 테스트 수행
7. 성능 테스트 수행
8. 품질 보증 승인

스테이징 환경으로 이동하기 전 기대 사항

- 개발 및 QA 팀은 조직의 요구 사항을 충족하기에 충분한 테스트를 수행했습니다.
- 개발 팀이 `bugfix`브랜치를 통해 발견된 버그를 해결했습니다.

스테이징 환경

스테이징 환경은 프로덕션 환경과 동일하게 구성됩니다. 예를 들어 데이터 설정의 범위와 크기는 프로덕션 워크로드와 비슷해야 합니다. 스테이징 환경을 사용하여 코드와 인프라가 예상대로 작동하는지 확인합니다. 이 환경은 미리 보기 또는 고객 데모와 같은 비즈니스 사용 사례에도 선호됩니다.

액세스

최소 권한 원칙에 따라 권한을 할당합니다. 개발자는 프로덕션 환경과 동일한 스테이징 환경에 액세스할 수 있어야 합니다.

빌드 단계

없음. 테스트 환경에서 사용된 것과 동일한 아티팩트가 스테이징 환경에서 재사용됩니다.

배포 단계

테스트 환경에서 승인 및 배포 후 스테이징 환경에서 `release`브랜치(Gitflow) 또는 `main`브랜치(Trunk 또는 GitHub Flow)의 배포를 자동으로 시작합니다. 다음은 스테이징 환경의 배포 단계입니다.

1. 스테이징 환경에 `release`브랜치(Gitflow) 또는 `main`브랜치(Trunk 또는 GitHub Flow) 배포
2. 지정된 직원의 수동 승인을 위해 일시 중지
3. 게시된 아티팩트 다운로드
4. 데이터베이스 버전 관리 수행
5. IaC 배포 수행
6. (선택 사항) 통합 테스트 수행
7. (선택 사항) 로드 테스트 수행
8. 필요한 개발, QA, 제품 또는 비즈니스 승인자의 승인을 받습니다.

프로덕션 환경으로 이동하기 전 기대 사항

- 프로덕션에 상응하는 릴리스가 스테이징 환경에 성공적으로 배포되었습니다.
- (선택 사항) 통합 및 로드 테스트 성공

프로덕션 환경

프로덕션 환경은 출시된 제품을 지원하여 실제 클라이언트가 실제 데이터를 처리합니다. 최소 권한으로 액세스 권한이 할당된 보호된 환경이며, 승격된 액세스는 제한된 기간 동안 감사된 예외 프로세스를 통해서만 허용되어야 합니다.

액세스

프로덕션 환경에서 개발자는 AWS Management Console에서 제한된 읽기 전용 액세스 권한을 가져야 합니다. 예를 들어 개발자는 day-to-day 작업을 위해 로그 데이터에 액세스할 수 있어야 합니다. 프로덕션에 대한 모든 릴리스는 배포 전에 승인 단계를 거쳐 게이트되어야 합니다.

빌드 단계

없음. 테스트 및 스테이징 환경에 사용된 것과 동일한 아티팩트가 프로덕션 환경에서 재사용됩니다.

배포 단계

스테이징 환경에서 승인 및 배포 후 프로덕션 환경에서 `release`브랜치(Gitflow) 또는 `main`브랜치(Trunk 또는 GitHub Flow)의 배포를 자동으로 시작합니다. 다음은 프로덕션 환경의 배포 단계입니다.

1. 프로덕션 환경에 `release`브랜치(Gitflow) 또는 `main`브랜치(Trunk 또는 GitHub Flow) 배포
2. 지정된 직원의 수동 승인을 위해 일시 중지
3. 게시된 아티팩트 다운로드
4. 데이터베이스 버전 관리 수행
5. IaC 배포 수행

Git 기반 개발 모범 사례

Git 기반 개발을 성공적으로 채택하려면 협업을 촉진하고 코드 품질을 유지하며 지속적 통합 및 지속적 전달(CI/CD)을 지원하는 일련의 모범 사례를 따르는 것이 중요합니다. 이 가이드의 모범 사례 외에도 AWS Well-Architected DevOps 지침을 검토하세요. 다음은 Git 기반 개발을 위한 몇 가지 주요 모범 사례입니다 AWS.

- 변경 사항을 작고 자주 유지 - 개발자에게 작은 증분 변경 사항 또는 기능을 커밋하도록 장려합니다. 이렇게 하면 병합 충돌의 위험이 줄어들고 문제를 빠르게 식별하고 수정할 수 있습니다.
- 기능 토글 사용 - 불완전하거나 실험적인 기능의 릴리스를 관리하려면 기능 토글 또는 기능 플래그를 사용합니다. 이렇게 하면 메인 브랜치의 안정성에 영향을 주지 않고 프로덕션에서 특정 기능을 숨기거나 활성화하거나 비활성화할 수 있습니다.
- 강력한 테스트 제품군 유지 - 포괄적이고 잘 유지 관리된 테스트 제품군은 문제를 조기에 감지하고 코드 베이스가 안정적으로 유지되는지 확인하는 데 매우 중요합니다. 테스트 자동화에 투자하고 실패한 테스트를 수정하는 데 우선순위를 둡니다.
- 지속적 통합 수용 - 지속적 통합 도구 및 사례를 사용하여 코드 변경 사항을 자동으로 빌드, 테스트하고 `develop`브랜치(Gitflow) 또는 브`main`랜치(Trunk 또는 GitHub Flow)에 통합합니다. 이를 통해 문제를 조기에 파악하고 개발 프로세스를 간소화할 수 있습니다.
- 코드 검토 수행 - 코드를 `main`브랜치에 통합하기 전에 코드에 대한 동료 검토를 장려하여 품질을 유지하고, 지식을 공유하고, 잠재적 문제를 파악합니다. 풀 요청 또는 기타 코드 검토 도구를 사용하여 이 프로세스를 용이하게 합니다.
- 손상된 빌드 모니터링 및 수정 - 빌드가 중단되거나 테스트가 실패하면 가능한 한 빨리 문제를 해결하는 데 우선순위를 둡니다. 이렇게 하면 `develop`브랜치(Gitflow) 또는 `main`브랜치(Trunk 또는 GitHub Flow)가 릴리스 가능한 상태로 유지되고 다른 개발자에게 미치는 영향이 최소화됩니다.
- 커뮤니케이션 및 협업 - 팀원 간에 열린 커뮤니케이션과 협업을 장려합니다. 개발자가 코드 베이스에 대해 진행 중인 작업과 변경 사항을 알고 있는지 확인합니다.
- 지속적으로 리팩터링 - 코드 베이스를 정기적으로 리팩터링하여 유지 관리를 개선하고 기술 부채를 줄입니다. 개발자에게 코드를 찾은 상태보다 더 나은 상태로 두도록 권장합니다.
- 복잡한 작업에 단기 브랜치 사용 - 규모가 크거나 복잡한 작업의 경우 단기 브랜치(작업 브랜치라고도 함)를 사용하여 변경 사항을 처리합니다. 그러나 브랜치 수명을 일반적으로 하루 미만으로 짧게 유지해야 합니다. 변경 사항을 가능한 한 빨리 `develop`브랜치(Gitflow) 또는 브`main`랜치(Trunk 또는 GitHub Flow)에 다시 병합합니다. 더 작고 빈번한 병합 및 검토는 팀이 하나의 대규모 병합 요청보다 더 쉽게 사용하고 처리할 수 있습니다.

- 팀 교육 및 지원 - Git 기반 개발을 처음 사용하거나 모범 사례 채택에 지침이 필요한 개발자에게 교육 및 지원을 제공합니다.

Git 분기 전략

이 가이드에서는 최소에서 가장 복잡한 순서로 다음 Git 기반 분기 전략을 자세히 설명합니다.

- 트렁크 - 트렁크 기반 개발은 모든 개발자가 일반적으로 `trunk` 또는 브랜치라고 하는 단일 `main`브랜치에서 작업하는 소프트웨어 개발 사례입니다. 이 접근 방식의 기본 개념은 코드 변경 사항을 자주 통합하고 자동화된 테스트 및 지속적인 통합에 의존하여 코드 기반을 지속적으로 릴리스 가능한 상태로 유지하는 것입니다.
- GitHub Flow – GitHub Flow는 GitHub에서 개발한 경량 브랜치 기반 워크플로입니다. 수명이 짧은 `feature`브랜치에 대한 아이디어를 기반으로 합니다. 기능이 완료되고 배포할 준비가 되면 기능이 `main`브랜치에 병합됩니다.
- Gitflow - Gitflow 접근 방식을 사용하면 개별 기능 브랜치에서 개발이 완료됩니다. 승인 후 `feature`브랜치를 일반적으로 이름이 인 통합 브랜치에 병합합니다`develop`. `develop` 브랜치에 충분한 기능이 누적되면 브`release`랜치가 생성되어 상위 환경에 기능을 배포합니다.

각 분기 전략에는 장단점이 있습니다. 모두 동일한 환경을 사용하지만 모두 동일한 브랜치 또는 수동 승인 단계를 사용하지는 않습니다. 가이드의이 섹션에서는 각 분기 전략을 자세히 검토하여 뉘앙스를 숙지하고 조직의 사용 사례에 맞는지 평가할 수 있습니다.

이 섹션의 주제:

- 트렁크 분기 전략
- GitHub Flow 분기 전략
- Gitflow 분기 전략

트렁크 분기 전략

트렁크 기반 개발은 모든 개발자가 일반적으로 `trunk` 또는 브랜치라고 하는 단일 `main`브랜치에서 작업하는 소프트웨어 개발 사례입니다. 이 접근 방식의 기본 개념은 코드 변경 사항을 자주 통합하고 자동화된 테스트 및 지속적인 통합에 의존하여 코드 기반을 지속적으로 릴리스 가능한 상태로 유지하는 것입니다.

트렁크 기반 개발에서 개발자는 작은 증분 업데이트를 목표로 하루에 여러 번 `main`브랜치에 변경 사항을 커밋합니다. 이를 통해 빠른 피드백 루프를 활성화하고, 병합 충돌의 위험을 줄이고, 팀원 간의 협업을 촉진할 수 있습니다. 이 관행은 자동 테스트에 의존하여 잠재적 문제를 조기에 포착하고 코드 베

이스가 안정적이고 릴리스 가능하도록 하기 때문에 잘 유지 관리된 테스트 제품군의 중요성을 강조합니다.

트렁크 기반 개발은 종종 기능 기반 개발(기능 분기 또는 기능 기반 개발이라고도 함)과 대조되며, 여기서 각각의 새로운 기능 또는 버그 수정은 기본 분기와 별도로 자체 전용 분기에서 개발됩니다. 트렁크 기반 개발과 기능 기반 개발 중에서 선택하는 것은 팀 크기, 프로젝트 요구 사항, 협업, 통합 빈도 및 릴리스 관리 간의 원하는 균형과 같은 요인에 따라 달라집니다.

트렁크 분기 전략에 대한 자세한 내용은 다음 리소스를 참조하세요.

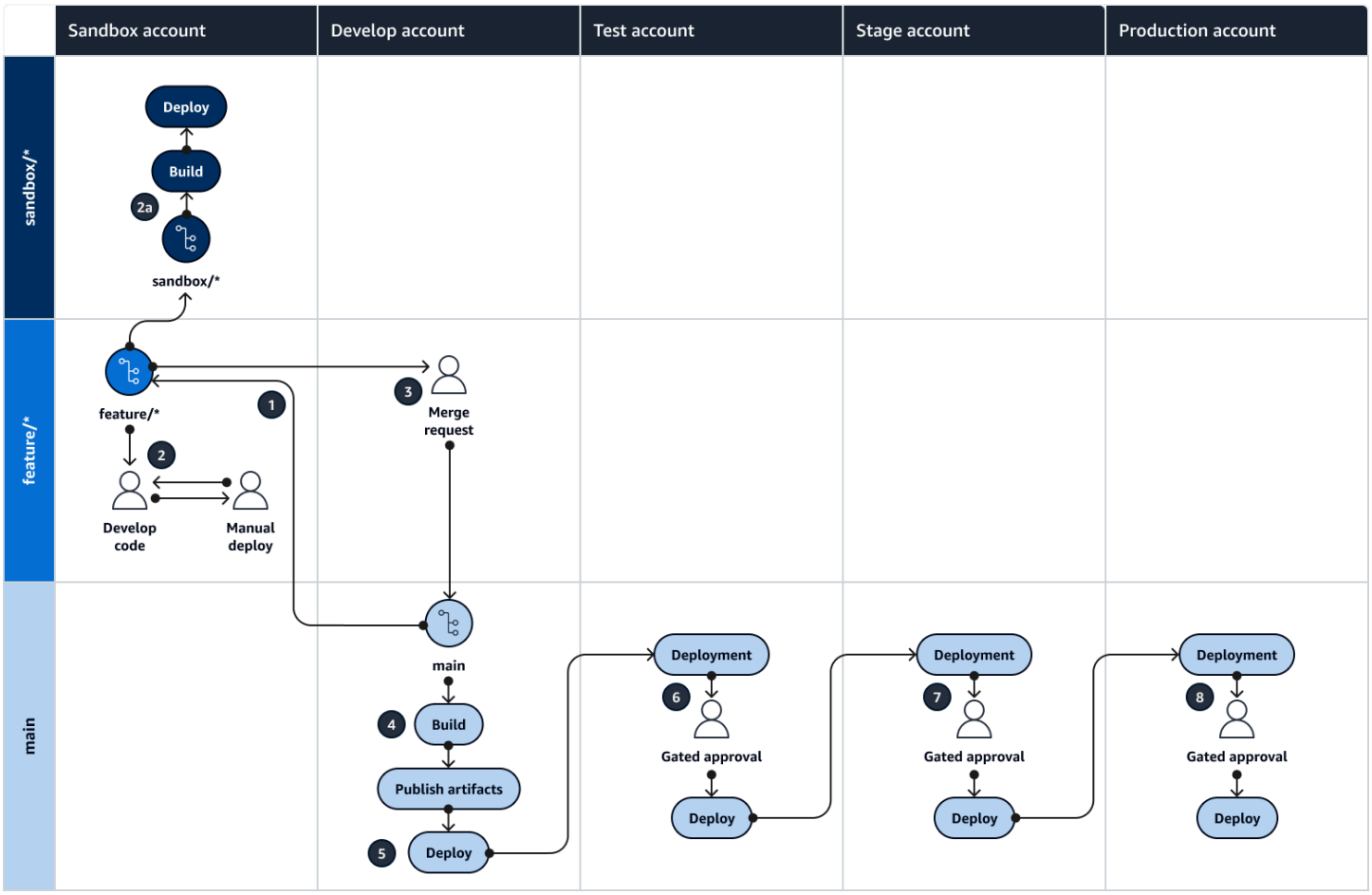
- 다중 계정 DevOps 환경을 위한 트렁크 분기 전략 구현(AWS 권고 가이드)
- 트렁크 기반 개발 소개(트렁크 기반 개발 웹 사이트)

이 섹션의 주제:

- 트렁크 전략의 시각적 개요
- 트렁크 전략의 브랜치
- 트렁크 전략의 장단점

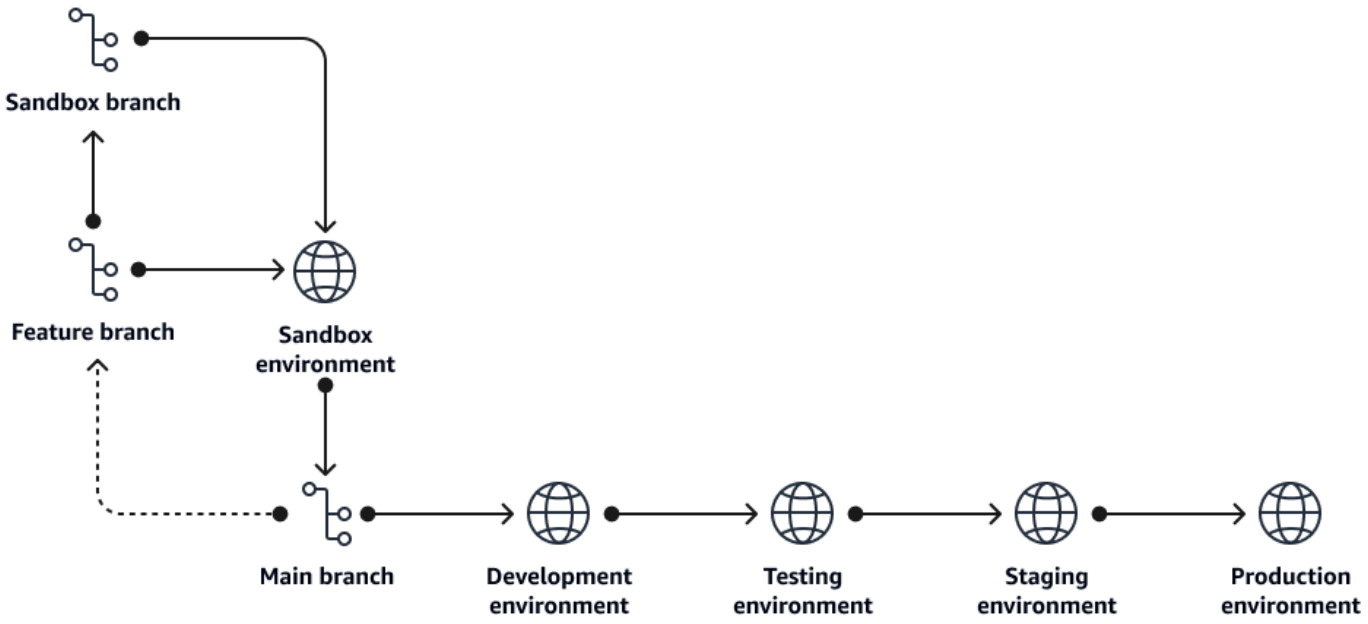
트렁크 전략의 시각적 개요

다음 다이어그램은 Punnett 사각형(Wikipedia)처럼 Trunk 분기 전략을 이해하는 데 사용할 수 있습니다. 세로 축의 브랜치를 가로 축의 AWS 환경과 정렬하여 각 시나리오에서 수행할 작업을 결정합니다. 원으로 표시된 숫자는 다이어그램에 표시된 작업 순서를 안내합니다. 이 다이어그램은 샌드박스 환경의 브랜치에서 브`feature`랜치의 프로덕션 릴리스에 이르기까지 Trunk `main`브랜치 전략의 개발 워크플로를 보여줍니다. 각 환경에서 발생하는 활동에 대한 자세한 내용은이 설명서의 DevOps 환경을 참조하세요.



트렁크 전략의 브랜치

트렁크 분기 전략에는 일반적으로 다음과 같은 분기가 있습니다.



기능 브랜치

feature 브랜치에서 기능을 개발하거나 핫픽스를 생성합니다. feature 브랜치를 생성하려면 브랜치에서 브main랜치합니다. 개발자는 feature브랜치에서 코드를 반복, 커밋 및 테스트합니다. 기능이 완료되면 개발자는 해당 기능을 승격합니다. feature 브랜치에서 전달되는 경로는 두 개뿐입니다.

- sandbox 브랜치에 병합
- main 브랜치에 병합 요청 생성

명명 규칙:	`feature/<story number>_<developer initials>_<descriptor>`
명명 규칙 예제:	`feature/123456_MS_Implement_Feature_A`

샌드박스 브랜치

이 브랜치는 비표준 트렁크 브랜치이지만 CI/CD 파이프라인 개발에 유용합니다. sandbox 브랜치는 주로 다음과 같은 목적으로 사용됩니다.

- CI/CD 파이프라인을 사용하여 샌드박스 환경에 대한 전체 배포 수행

- 개발 또는 테스트와 같은 낮은 환경에서 전체 테스트를 위한 병합 요청을 제출하기 전에 파이프라인을 개발하고 테스트합니다.

Sandbox 브랜치는 본질적으로 일시적이며 수명이 짧습니다. 특정 테스트가 완료된 후 삭제해야 합니다.

명명 규칙:	`sandbox/<story number>_<developer initials>_<descriptor>`
명명 규칙 예제:	`sandbox/123456_MS_Test_Pipeline_Deploy`

기본 브랜치

`main` 브랜치는 항상 프로덕션에서 실행 중인 코드를 나타냅니다. 코드는에서 분기되어 `main`개발되었다가 다시에 병합됩니다`main`. 의 배포는 모든 환경을 대상으로 할 `main` 수 있습니다. 삭제를 방지하려면 브랜치에 대해 `main`브랜치 보호를 활성화합니다.

명명 규칙:	`main`

핫픽스 브랜치

트렁크 기반 워크플로에는 전용 `hotfix`브랜치가 없습니다. 핫픽스는 브`feature`랜치를 사용합니다.

트렁크 전략의 장단점

Trunk 분기 전략은 강력한 커뮤니케이션 기술을 갖춘 소규모의 성숙한 개발 팀에 매우 적합합니다. 또한 애플리케이션에 대한 지속적인 롤링 기능 릴리스가 있는 경우에도 잘 작동합니다. 대규모 또는 조각화된 개발 팀이 있거나 대규모의 예약된 기능 릴리스가 있는 경우에는 적합하지 않습니다. 병합 충돌은 이 모델에서 발생하므로 병합 충돌 해결은 핵심 기술입니다. 모든 팀원은 그에 따라 교육을 받아야 합니다.

장점

트렁크 기반 개발은 개발 프로세스를 개선하고, 협업을 간소화하고, 소프트웨어의 전반적인 품질을 향상시킬 수 있는 몇 가지 이점을 제공합니다. 다음은 몇 가지 주요 이점입니다.

- 더 빠른 피드백 루프 - 트렁크 기반 개발을 통해 개발자는 코드 변경 사항을 자주, 종종 하루에 여러 번 통합합니다. 이를 통해 잠재적 문제에 대한 더 빠른 피드백을 얻을 수 있으며 개발자는 기능 기반 개발 모델에서보다 더 빠르게 문제를 식별하고 수정할 수 있습니다.
- 병합 충돌 감소 - 트렁크 기반 개발에서는 변경 사항이 지속적으로 통합되므로 크고 복잡한 병합 충돌의 위험이 최소화됩니다. 이렇게 하면 더 깨끗한 코드 기반을 유지하고 충돌 해결에 소요되는 시간을 줄일 수 있습니다. 충돌 해결은 기능 기반 개발에서 시간이 많이 걸리고 오류가 발생하기 쉽습니다.
- 협업 개선 - 트렁크 기반 개발을 통해 개발자는 동일한 브랜치에서 함께 작업하여 팀 내에서 더 나은 커뮤니케이션과 협업을 촉진할 수 있습니다. 이로 인해 문제 해결 속도가 빨라지고 팀 역량이 향상될 수 있습니다.
- 더 쉬운 코드 검토 - 트렁크 기반 개발에서 코드 변경이 더 작고 빈번하기 때문에 철저한 코드 검토를 더 쉽게 수행할 수 있습니다. 변경 사항이 작을수록 일반적으로 더 쉽게 이해하고 검토할 수 있으므로 잠재적 문제와 개선 사항을 보다 효과적으로 식별할 수 있습니다.
- 지속적 통합 및 제공 - 트렁크 기반 개발은 지속적 통합 및 지속적 제공(CI/CD)의 원칙을 지원합니다. 코드 기반을 릴리스 가능한 상태로 유지하고 변경 사항을 자주 통합하면 팀은 CI/CD 관행을 더 쉽게 채택할 수 있으므로 배포 주기가 빨라지고 소프트웨어 품질이 향상됩니다.
- 향상된 코드 품질 - 빈번한 통합, 테스트 및 코드 검토를 통해 트렁크 기반 개발은 전반적인 코드 품질 향상에 기여할 수 있습니다. 개발자는 문제를 더 빠르게 포착하고 수정할 수 있으므로 시간이 지남에 따라 기술 부채가 누적될 가능성을 줄일 수 있습니다.
- 간소화된 분기 전략 - 트렁크 기반 개발은 수명이 긴 분기 수를 줄여 분기 전략을 간소화합니다. 이렇게 하면 특히 대규모 프로젝트 또는 팀의 경우 코드 기반을 더 쉽게 관리하고 유지 관리할 수 있습니다.

단점

트렁크 기반 개발에는 개발 프로세스와 팀 역학에 영향을 미칠 수 있는 몇 가지 단점이 있습니다. 다음은 몇 가지 주목할 만한 단점입니다.

- 제한된 격리 - 모든 개발자가 동일한 브랜치에서 작업하기 때문에 팀의 모든 사람이 변경 사항을 즉시 볼 수 있습니다. 이로 인해 간섭 또는 충돌이 발생하여 의도하지 않은 부작용이 발생하거나 빌드가 중단될 수 있습니다. 반대로 기능 기반 개발은 변경 사항을 더 잘 격리하여 개발자가 더 독립적으로 작업할 수 있도록 합니다.
- 테스트에 대한 부담 증가 - 트렁크 기반 개발은 지속적인 통합과 자동화된 테스트를 통해 문제를 신속하게 포착합니다. 그러나이 접근 방식은 테스트 인프라에 많은 부담을 줄 수 있으며 잘 유지 관리

된 테스트 제품군이 필요합니다. 테스트가 포괄적이거나 신뢰할 수 없는 경우 기본 브랜치에서 감지되지 않은 문제가 발생할 수 있습니다.

- 릴리스에 대한 제어 감소 - 트렁크 기반 개발은 코드 기반을 지속적으로 릴리스 가능한 상태로 유지하는 것을 목표로 합니다. 이는 유리할 수 있지만 엄격한 릴리스 일정이 있거나 특정 기능을 함께 릴리스해야 하는 프로젝트에는 항상 적합하지 않을 수 있습니다. 기능 기반 개발을 통해 기능이 릴리스되는 시기와 방법을 더 잘 제어할 수 있습니다.
- 코드 이탈 - 개발자가 변경 사항을 기본 브랜치에 지속적으로 통합하면 트렁크 기반 개발로 인해 코드 이탈이 증가할 수 있습니다. 이로 인해 개발자가 코드 베이스의 현재 상태를 추적하기 어렵고 최근 변경의 영향을 이해하려고 할 때 혼란이 발생할 수 있습니다.
- 강력한 팀 문화 필요 - 트렁크 기반 개발에는 팀원 간의 높은 수준의 원칙, 커뮤니케이션 및 협업이 필요합니다. 이는 특히 대규모 팀이나이 접근 방식에 경험이 부족한 개발자와 협력할 때 유지 관리하기 어려울 수 있습니다.
- 확장성 문제 - 개발 팀의 규모가 증가함에 따라 기본 브랜치에 통합되는 코드 변경 횟수가 빠르게 증가할 수 있습니다. 이로 인해 빌드 중단 및 테스트 실패가 더 자주 발생하여 코드 베이스를 해제 가능한 상태로 유지하기 어려울 수 있습니다.

GitHub Flow 분기 전략

GitHub Flow는 GitHub에서 개발한 경량 브랜치 기반 워크플로입니다. GitHub Flow는 기능이 완료되고 배포할 준비가 되면 기본 브랜치에 병합되는 수명이 짧은 기능 브랜치에 대한 아이디어를 기반으로 합니다. GitHub Flow의 주요 원칙은 다음과 같습니다.

- 브랜칭은 간단합니다. 개발자는 클릭 몇 번으로 작업에 사용할 특성 브랜치를 생성하여 기본 브랜치에 영향을 주지 않고 협업하고 실험하는 기능을 개선할 수 있습니다.
- 지속적 배포 - 변경 사항이 기본 브랜치에 병합되는 즉시 배포되므로 신속한 피드백과 반복이 가능합니다.
- 병합 요청 - 개발자는 병합 요청을 사용하여 변경 사항을 기본 브랜치에 병합하기 전에 논의 및 검토 프로세스를 시작합니다.

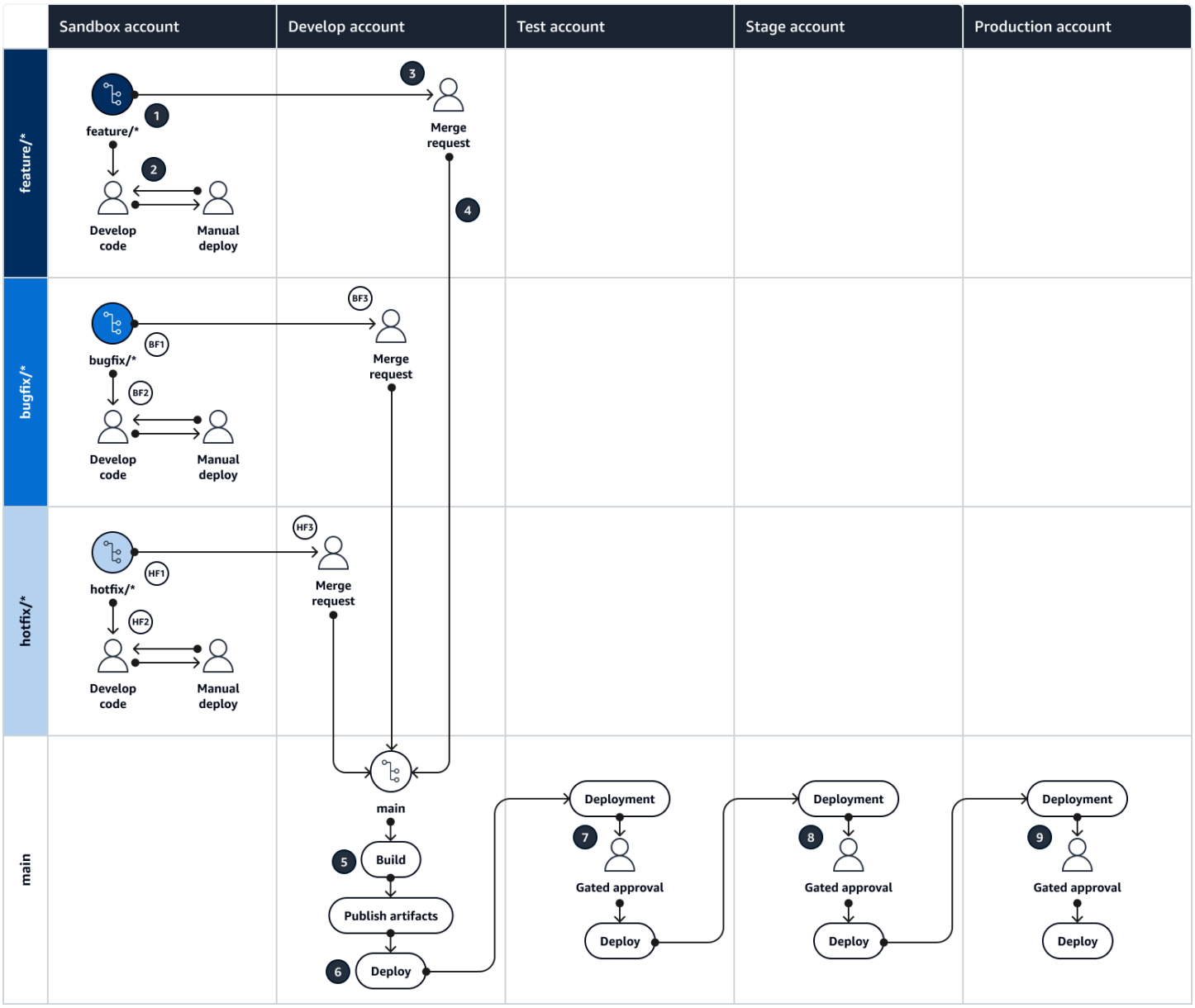
GitHub 흐름에 대한 자세한 내용은 다음 리소스를 참조하세요.

- 다중 계정 DevOps 환경을 위한 GitHub Flow 분기 전략 구현(AWS 권장 가이드)
- GitHub Flow Quickstart(GitHub 설명서)
- GitHub Flow를 사용해야 하는 이유 (GitHub Flow 웹 사이트)

이 섹션의 주제:

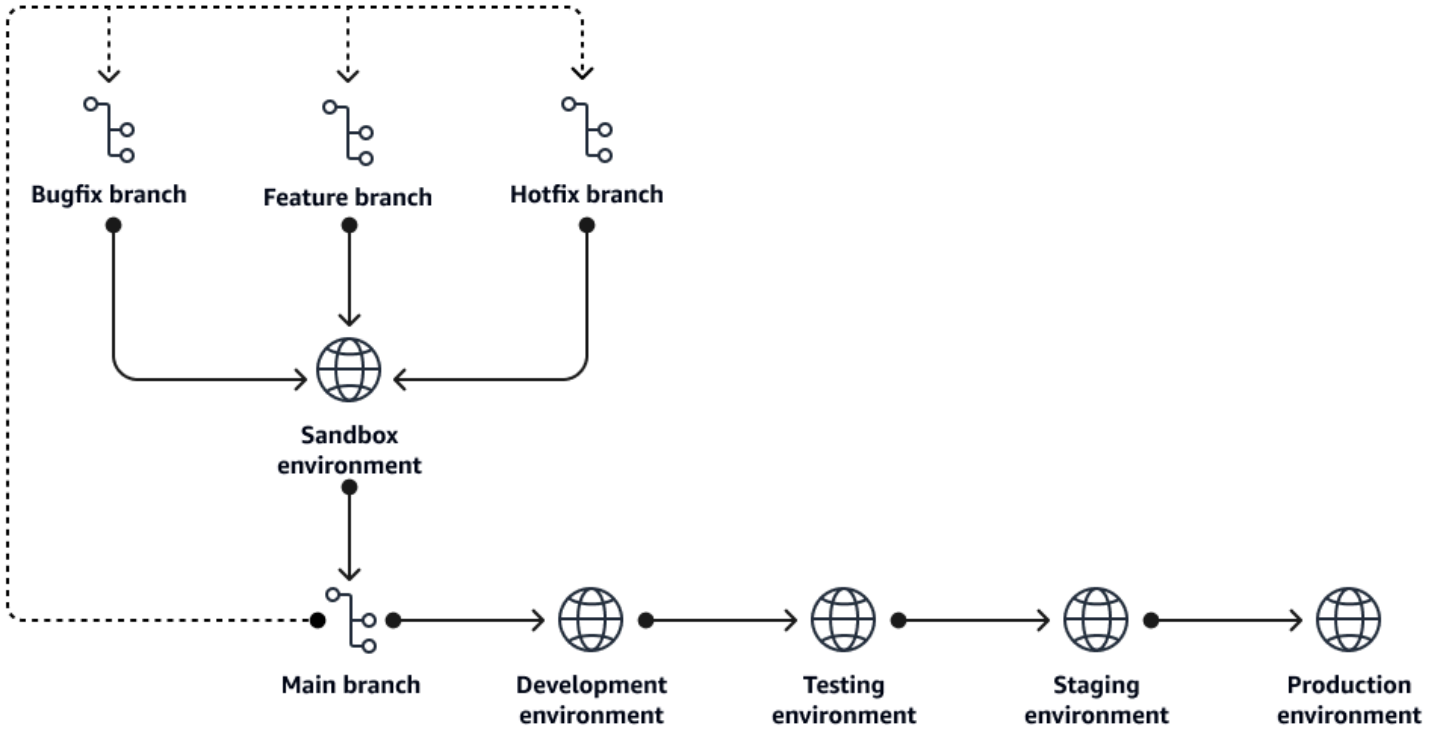
GitHub 흐름 전략의 시각적 개요

다음 다이어그램을 Punnett 사각형처럼 사용하여 GitHub Flow 분기 전략을 이해할 수 있습니다. 세로 축의 브랜치를 가로 축의 AWS 환경과 정렬하여 각 시나리오에서 수행할 작업을 결정합니다. 원으로 표시된 숫자는 다이어그램에 표시된 작업 순서를 안내합니다. 이 다이어그램은 샌드박스 환경의 기능 브랜치에서 기본 브랜치의 프로덕션 릴리스에 이르기까지 GitHub Flow 브랜치 전략의 개발 워크플로를 보여줍니다. 각 환경에서 발생하는 활동에 대한 자세한 내용은이 설명서의 DevOps 환경을 참조하세요.



GitHub 흐름 전략의 브랜치

GitHub Flow 분기 전략에는 일반적으로 다음과 같은 분기가 있습니다.



기능 브랜치

브feature랜치에서 기능을 개발합니다. feature 브랜치를 생성하려면 브랜치에서 브main랜치합니다. 개발자는 feature브랜치에서 코드를 반복, 커밋 및 테스트합니다. 기능이 완료되면 개발자는에 대한 병합 요청을 생성하여 기능을 승격합니다main.

명명 규칙:	`feature/<story number>_<developer initials>_<descriptor>`
명명 규칙 예제:	`feature/123456_MS_Implement_Feature_A`

버그 수정 브랜치

bugfix 브랜치는 문제를 해결하는 데 사용됩니다. 이러한 브랜치는 브main랜치에서 분기됩니다. 샌드박스 또는 하위 환경에서 버그 수정을 테스트한 후 병합 요청을 main 통해에 병합하여 상위 환경으로 승격할 수 있습니다. 이는 조직 및 추적에 권장되는 이름 지정 규칙이며,이 프로세스는 기능 브랜치를 사용하여 관리할 수도 있습니다.

명명 규칙:	`bugfix/<ticket number>_<developer initials>_<descriptor>`
명명 규칙 예제:	`bugfix/123456_MS_Fix_Problem_A`

핫픽스 브랜치

hotfix 브랜치는 개발 직원과 프로덕션에 배포된 코드 간의 지연을 최소화하면서 영향력이 큰 중요한 문제를 해결하는 데 사용됩니다. 이러한 브랜치는 브main랜치에서 분기됩니다. 핫픽스를 샌드박스 또는 하위 환경에서 테스트한 후 병합 요청을 main 통해에 병합하여 상위 환경으로 승격할 수 있습니다. 이는 조직 및 추적에 권장되는 이름 지정 규칙이며,이 프로세스는 기능 브랜치를 사용하여 관리할 수도 있습니다.

명명 규칙:	`hotfix/<ticket number>_<developer initials>_<descriptor>`
명명 규칙 예제:	`hotfix/123456_MS_Fix_Problem_A`

기본 브랜치

main 브랜치는 항상 프로덕션에서 실행 중인 코드를 나타냅니다. 코드는 병합 요청을 사용하여 main브랜치에서 feature브랜치로 병합됩니다. 삭제를 방지하고 개발자가 코드를에 직접 푸시하지 못하도록 하려면 main브랜치에 대한 브main랜치 보호를 활성화합니다.

명명 규칙:	`main`

GitHub 흐름 전략의 장단점

Github Flow 분기 전략은 강력한 커뮤니케이션 기술을 갖춘 소규모의 성숙한 개발 팀에 적합합니다. 이 전략은 지속적인 제공을 구현하려는 팀에 적합하며 일반적인 CI/CD 엔진에서 잘 지원됩니다. GitHub Flow는 가볍고 규칙이 너무 많지 않으며 빠르게 움직이는 팀을 지원할 수 있습니다. 팀의 규정 준수 또는 릴리스 프로세스가 엄격한 경우에는 적합하지 않습니다. 병합 충돌은이 모델에서 일반적이며 자주 발생할 수 있습니다. 병합 충돌 해결은 핵심 기술이므로 그에 따라 모든 팀원을 교육해야 합니다.

장점

GitHub Flow는 개발 프로세스를 개선하고, 협업을 간소화하고, 소프트웨어의 전반적인 품질을 향상시킬 수 있는 몇 가지 이점을 제공합니다. 다음은 몇 가지 주요 이점입니다.

- 유연하고 가벼운 - GitHub Flow는 개발자가 소프트웨어 개발 프로젝트에서 협업하는 데 도움이 되는 가볍고 유연한 워크플로입니다. 이를 통해 복잡성을 최소화하면서 빠르게 반복하고 실험할 수 있습니다.
- 간소화된 공동 작업 - GitHub Flow는 기능 개발을 관리하기 위한 명확하고 간소화된 프로세스를 제공합니다. 이를 통해 빠르게 검토하고 병합할 수 있는 작고 집중적인 변경 사항을 장려하여 효율성을 개선할 수 있습니다.
- 버전 관리 지우기 - GitHub Flow를 사용하면 모든 변경이 별도의 브랜치에서 이루어집니다. 이렇게 하면 명확하고 추적 가능한 버전 관리 기록이 설정됩니다. 이를 통해 개발자는 변경 사항을 추적 및 이해하고, 필요한 경우 되돌리고, 신뢰할 수 있는 코드 기반을 유지할 수 있습니다.
- 원활한 지속적 통합 - GitHub Flow는 지속적 통합 도구와 통합됩니다. 풀 요청을 생성하면 자동화된 테스트 및 배포 프로세스를 시작할 수 있습니다. CI 도구를 사용하면 변경 사항이 `main`브랜치에 병합되기 전에 철저하게 테스트하여 코드 베이스에 버그가 발생할 위험을 줄일 수 있습니다.
- 빠른 피드백 및 지속적인 개선 - GitHub Flow는 풀 요청을 통해 빈번한 코드 검토 및 논의를 촉진하여 빠른 피드백 루프를 장려합니다. 이를 통해 문제를 조기에 감지하고, 팀원 간의 지식 공유를 촉진하며, 궁극적으로 개발 팀 내에서 코드 품질과 협업을 개선할 수 있습니다.
- 간소화된 롤백 및 되돌리기 - 코드 변경으로 인해 예기치 않은 버그 또는 문제가 발생하는 경우 GitHub Flow는 변경 사항을 롤백하거나 되돌리는 프로세스를 간소화합니다. 커밋 및 브랜치의 명확한 기록을 확보하면 문제가 되는 변경 사항을 더 쉽게 식별하고 되돌릴 수 있으므로 안정적이고 기능적인 코드 기반을 유지하는 데 도움이 됩니다.
- 경량 학습 곡선 - GitHub Flow는 Gitflow보다 더 쉽게 학습하고 채택할 수 있으며, 특히 Git 및 버전 제어 개념에 이미 익숙한 팀의 경우 더욱 그렇습니다. 단순하고 직관적인 분기 모델을 사용하면 다양한 경험 수준의 개발자가 액세스할 수 있으므로 새로운 개발 워크플로 채택과 관련된 학습 곡선을 줄일 수 있습니다.
- 지속적인 개발 - GitHub Flow는 팀이 `main`브랜치에 병합되는 즉시 모든 변경 사항을 즉시 배포할 수 있도록 하여 지속적인 배포 접근 방식을 수용할 수 있도록 지원합니다. 이 간소화된 프로세스는 불필요한 지연을 없애고 사용자가 최신 업데이트 및 개선 사항을 신속하게 사용할 수 있도록 합니다. 따라서 개발 주기가 더 민첩하고 응답성이 뛰어납니다.

단점

GitHub Flow는 몇 가지 이점을 제공하지만 잠재적 단점도 고려하는 것이 중요합니다.

- 대규모 프로젝트에 대한 제한적 적합성 - GitHub Flow는 복잡한 코드 베이스와 여러 장기 기능 브랜치가 있는 대규모 프로젝트에는 적합하지 않을 수 있습니다. 이러한 경우 Gitflow와 같은 보다 구조화된 워크플로는 동시 개발 및 릴리스 관리를 더 잘 제어할 수 있습니다.
- 공식 릴리스 구조 부족 - GitHub Flow는 버전 관리, 핫픽스 또는 유지 관리 브랜치와 같은 릴리스 프로세스 또는 지원 기능을 명시적으로 정의하지 않습니다. 이는 엄격한 릴리스 관리가 필요하거나 장기 지원 및 유지 관리가 필요한 프로젝트의 제한 사항일 수 있습니다.
- 장기 릴리스 계획에 대한 제한된 지원 - GitHub Flow는 엄격한 로드맵 또는 광범위한 기능 종속성이 있는 프로젝트와 같이 장기 릴리스 계획이 필요한 프로젝트와 일치하지 않을 수 있는 단기 기능 브랜치에 중점을 둡니다. 복잡한 릴리스 일정을 관리하는 것은 GitHub Flow의 제약 내에서 어려울 수 있습니다.
- 빈번한 병합 충돌 가능성 - GitHub Flow는 빈번한 분기 및 병합을 장려하므로 특히 개발 활동이 많은 프로젝트에서 병합 충돌이 발생할 가능성이 있습니다. 이러한 충돌을 해결하는 데는 시간이 많이 걸릴 수 있으며 개발 팀의 추가 노력이 필요할 수 있습니다.
- 공식화된 워크플로 단계 부족 - GitHub Flow는 알파, 베타 또는 릴리스 후보 단계와 같은 개발을 위한 명시적 단계를 정의하지 않습니다. 이렇게 하면 프로젝트의 현재 상태나 다양한 브랜치 또는 릴리스의 안정성 수준을 전달하기가 더 어려울 수 있습니다.
- 주요 변경 사항의 영향 - GitHub Flow는 변경 사항을 `main`브랜치에 자주 병합하도록 권장하므로 코드 베이스의 안정성에 영향을 미치는 주요 변경 사항을 도입할 위험이 더 높습니다. 이러한 위험을 효과적으로 완화하려면 엄격한 코드 검토 및 테스트 관행이 중요합니다.

Gitflow 분기 전략

Gitflow는 여러 브랜치를 사용하여 코드를 개발에서 프로덕션으로 이동하는 분기 모델입니다. Gitflow는 릴리스 주기가 예약되어 있고 기능 모음을 릴리스로 정의해야 하는 팀에 적합합니다. 개발은 승인을 받아 통합에 사용되는 개발 브랜치로 병합되는 개별 기능 브랜치에서 완료됩니다. 이 브랜치의 기능은 프로덕션 준비가 완료된 것으로 간주됩니다. 계획된 모든 기능이 개발 브랜치에 누적되면 상위 환경에 배포하기 위한 릴리스 브랜치가 생성됩니다. 이렇게 분리하면 정의된 일정에 따라 이름이 지정된 환경으로 이동하는 변경 사항을 제어할 수 있습니다. 필요한 경우이 프로세스를 더 빠른 배포 모델로 가속화할 수 있습니다.

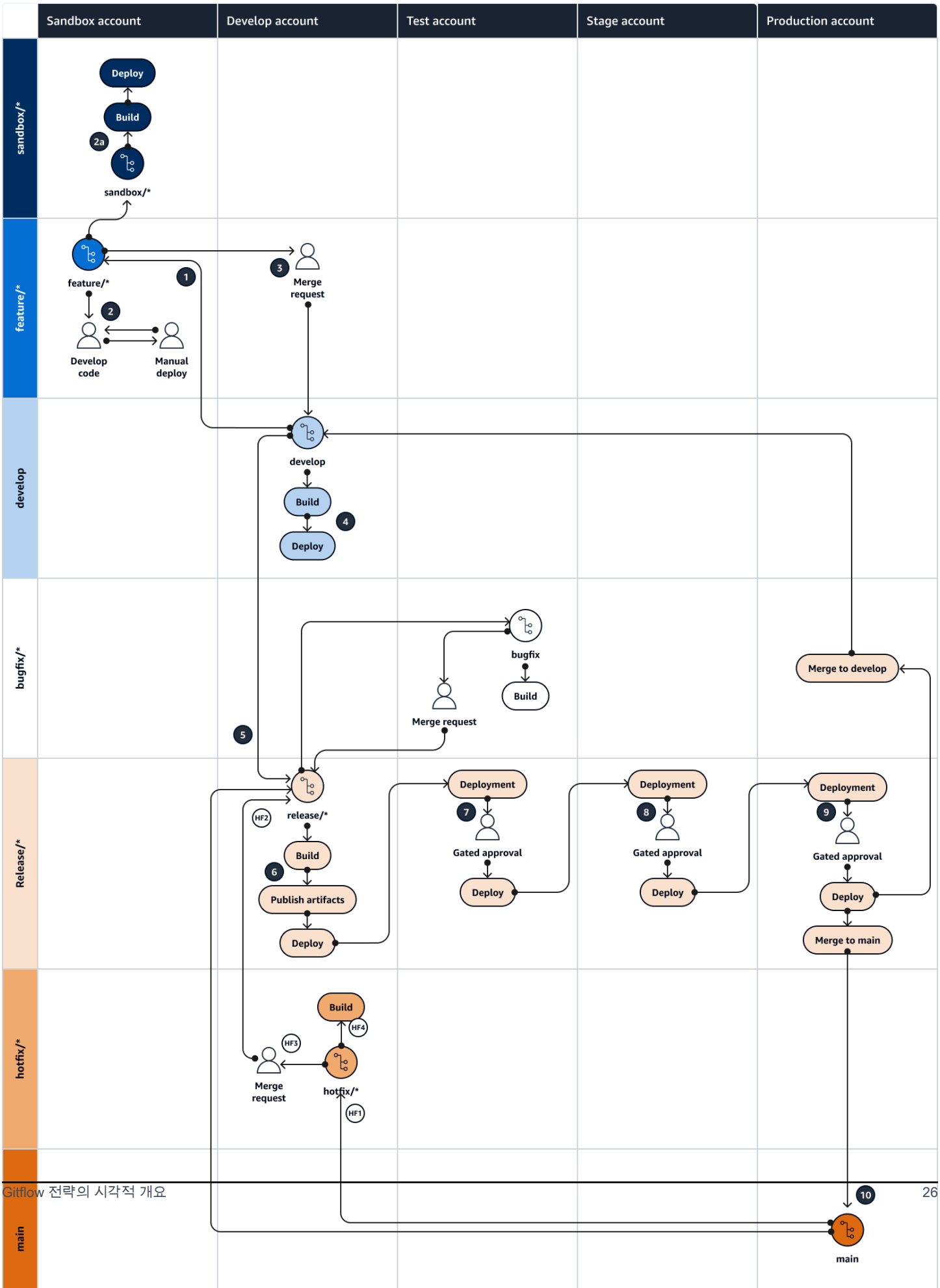
Gitflow 분기 전략에 대한 자세한 내용은 다음 리소스를 참조하세요.

- 다중 계정 DevOps 환경을 위한 Gitflow 분기 전략 구현(AWS 권고 가이드)
- 원본 Gitflow 블로그(Vincent Driessen 블로그 게시물)
- Gitflow 워크플로(Atlassian)

이 섹션의 주제:

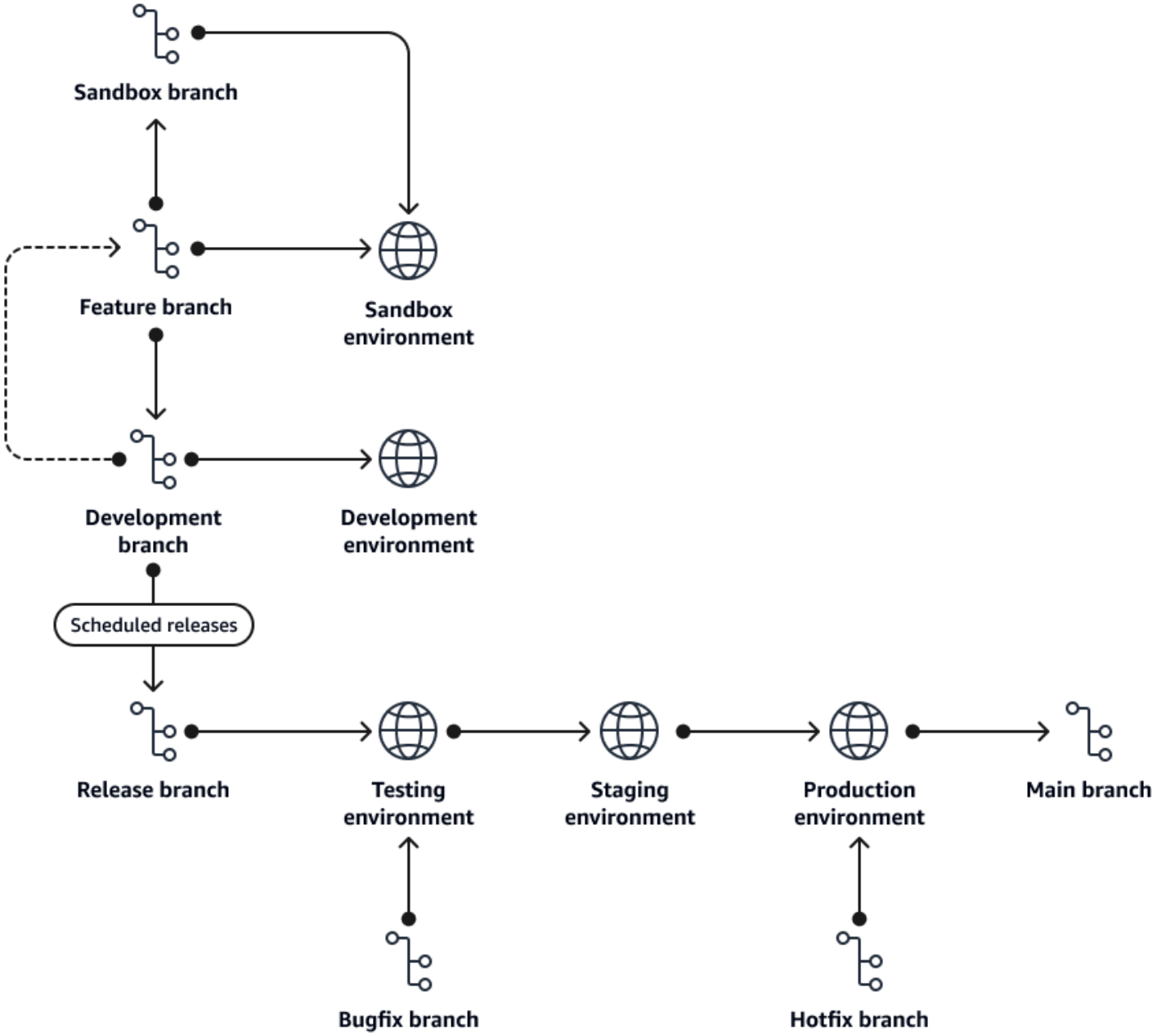
Gitflow 전략의 시각적 개요

다음 다이어그램은 Punnett 사각형처럼 Gitflow 분기 전략을 이해하는 데 사용할 수 있습니다. 세로 축의 브랜치를 가로 축의 AWS 환경과 정렬하여 각 시나리오에서 수행할 작업을 결정합니다. 원으로 표시된 숫자는 다이어그램에 표시된 작업 순서를 안내합니다. 각 환경에서 발생하는 활동에 대한 자세한 내용은이 가이드의 DevOps 환경을 참조하세요.


Sandbox account
Develop account
Test account
Stage account
Production account
sandbox/*
feature/*
develop
bugfix/*
Release/*
hotfix/*
main
Deploy
Build
2a
sandbox/*
feature/*
1
2
Develop code
Manual deploy
3
Merge request
develop
Build
4
Deploy
bugfix
Build
Merge request
5
Merge to develop
release/*
HF2
Build
6
Publish artifacts
Deploy
Deployment
7
Gated approval
Deploy
Deployment
8
Gated approval
Deploy
Deployment
9
Gated approval
Deploy
Merge to main
Build
HF4
HF3
Merge request
hotfix/*
HF1
10
main

Gitflow 전략의 브랜치

Gitflow 분기 전략에는 일반적으로 다음과 같은 분기가 있습니다.



기능 브랜치

`Feature` 브랜치는 기능을 개발하는 단기 브랜치입니다. `feature` 브랜치는 브`develop`랜치에서 분기하여 생성됩니다. 개발자는 `feature`브랜치에서 코드를 반복, 커밋 및 테스트합니다. 기능이 완료되면 개발자는 해당 기능을 승격합니다. 특성 브랜치에서 전달되는 경로는 두 개뿐입니다.

- sandbox 브랜치에 병합
- develop 브랜치에 병합 요청 생성

명명 규칙:	`feature/<story number>_<developer initials>_<descriptor>`
명명 규칙 예제:	`feature/123456_MS_Implement_Feature_A`

샌드박스 브랜치

sandbox 브랜치는 Gitflow의 비표준 단기 브랜치입니다. 그러나 CI/CD 파이프라인 개발에 유용합니다. sandbox 브랜치는 주로 다음과 같은 목적으로 사용됩니다.

- 수동 배포 대신 CI/CD 파이프라인을 사용하여 샌드박스 환경에 대한 전체 배포를 수행합니다.
- 개발 또는 테스트와 같은 낮은 환경에서 전체 테스트를 위한 병합 요청을 제출하기 전에 파이프라인을 개발하고 테스트합니다.

Sandbox 브랜치는 본질적으로 일시적이며 수명이 길지 않습니다. 특정 테스트가 완료된 후 삭제해야 합니다.

명명 규칙:	`sandbox/<story number>_<developer initials>_<descriptor>`
명명 규칙 예제:	`sandbox/123456_MS_Test_Pipeline_Deploy`

브랜치 개발

develop 브랜치는 기능이 통합, 구축, 검증 및 개발 환경에 배포되는 수명이 긴 브랜치입니다. 모든 feature브랜치는 브develop랜치에 병합됩니다. develop 브랜치로의 병합은 성공적인 빌드와 2개의 개발자 승인이 필요한 병합 요청을 통해 완료됩니다. 삭제를 방지하려면 브랜치에서 develop브랜치 보호를 활성화합니다.

명명 규칙:	`develop`

릴리스 브랜치

Gitflow에서 브`release`랜치는 단기 브랜치입니다. 이러한 브랜치는 build-once, deploy-many 방법론을 수용하여 여러 환경에 배포할 수 있으므로 특별합니다. `Release`브랜치는 테스트, 스테이징 또는 프로덕션 환경을 대상으로 할 수 있습니다. 개발 팀이 기능을 더 높은 환경으로 승격하기로 결정한 후 새 `release`브랜치를 생성하고 이전 릴리스의 버전 번호 증분을 사용합니다. 각 환경의 게이트에서 배포를 진행하려면 수동 승인이 필요합니다. `Release`브랜치는 병합 요청을 변경해야 합니다.

`release` 브랜치를 프로덕션에 배포한 후에는 `develop` 및 `main`브랜치에 다시 병합하여 버그 수정 또는 핫픽스가 향후 개발 작업에 다시 병합되도록 해야 합니다.

명명 규칙:	`release/v{major}.{minor}`
명명 규칙 예제:	`release/v1.0`

기본 브랜치

`main` 브랜치는 프로덕션에서 실행 중인 코드를 항상 나타내는 수명이 긴 브랜치입니다. 코드는 릴리스 파이프라인에서 성공적으로 배포된 후 릴리스 `main`브랜치에서 브랜치에 자동으로 병합됩니다. 삭제를 방지하려면 브랜치에서 `main`브랜치 보호를 활성화합니다.

명명 규칙:	`main`

bugfix 브랜치

브`bugfix`랜치는 프로덕션에 릴리스되지 않은 릴리스 브랜치의 문제를 해결하는 데 사용되는 단기 브랜치입니다. `bugfix` 브랜치는 `release`브랜치의 수정 사항을 테스트, 스테이징 또는 프로덕션 환경으로 승격하는 데만 사용해야 합니다. `bugfix` 브랜치는 항상 브`release`랜치에서 분기됩니다.

버그 수정을 테스트한 후 병합 요청을 통해 `release`브랜치로 승격할 수 있습니다. 그런 다음 표준 릴리스 프로세스에 따라 `release`브랜치를 앞으로 푸시할 수 있습니다.

명명 규칙:	`bugfix/<ticket>_<developer initials>_<descriptor>`
명명 규칙 예제:	`bugfix/123456_MS_Fix_Problem_A`

핫픽스 브랜치

`hotfix` 브랜치는 프로덕션 문제를 해결하는 데 사용되는 단기 브랜치입니다. 프로덕션 환경에 도달하기 위해 신속하게 처리해야 하는 수정 사항을 승격하는 데만 사용됩니다. `hotfix` 브랜치는 항상에서 분기됩니다`main`.

핫픽스를 테스트한 후에서 생성된 `release`브랜치에 대한 병합 요청을 통해 핫픽스를 프로덕션으로 승격할 수 있습니다`main`. 그런 다음 테스트를 위해 표준 릴리스 프로세스에 따라 `release`브랜치를 앞으로 푸시할 수 있습니다.

명명 규칙:	`hotfix/<ticket>_<developer initials>_<descriptor>`
명명 규칙 예제:	`hotfix/123456_MS_Fix_Problem_A`

Gitflow 전략의 장단점

Gitflow 분기 전략은 엄격한 릴리스 및 규정 준수 요구 사항이 있는 더 크고 분산된 팀에 적합합니다. Gitflow는 조직의 예측 가능한 릴리스 주기에 기여하며, 이는 대규모 조직에서 선호되는 경우가 많습니다. Gitflow는 가드레일이 소프트웨어 개발 수명 주기를 올바르게 완료해야 하는 팀에도 적합합니다. 이는 전략에 포함된 검토 및 품질 보증 기회가 여러 개 있기 때문입니다. Gitflow는 여러 버전의 프로덕션 릴리스를 동시에 유지해야 하는 팀에도 적합합니다. GItflow의 몇 가지 단점은 다른 분기 모델보다 더 복잡하며 성공적으로 완료하려면 패턴을 엄격하게 준수해야 한다는 것입니다. Gitflow는 릴리스 브랜치 관리의 엄격한 특성으로 인해 지속적 전달을 위해 노력하는 조직에 적합하지 않습니다. Gitflow 릴리스 브랜치는 적시에 제대로 처리되지 않으면 기술 부채가 누적될 수 있는 수명이 긴 브랜치일 수 있습니다.

장점

Gitflow 기반 개발은 개발 프로세스를 개선하고, 협업을 간소화하고, 소프트웨어의 전반적인 품질을 향상시킬 수 있는 몇 가지 이점을 제공합니다. 다음은 몇 가지 주요 이점입니다.

- 예측 가능한 릴리스 프로세스 - Gitflow는 정기적이고 예측 가능한 릴리스 프로세스를 따릅니다. 정기적인 개발 및 릴리스 주기가 있는 팀에 적합합니다.
- 공동 작업 개선 - Gitflow는 `feature` 및 `release`브랜치 사용을 장려합니다. 이 두 브랜치는 팀이 서로에 대한 종속성을 최소화하면서 병렬로 작업하는 데 도움이 됩니다.
- 여러 환경에 매우 적합 - Gitflow는 수명이 더 긴 `release`브랜치일 수 있는 브랜치를 사용합니다. 이러한 브랜치를 통해 팀은 장기간에 걸쳐 개별 릴리스를 대상으로 지정할 수 있습니다.
- 프로덕션 환경에서 여러 버전 - 팀이 프로덕션 환경에서 여러 버전의 소프트웨어를 지원하는 경우 Gitflow `release`브랜치는이 요구 사항을 지원합니다.
- 기본 제공 코드 품질 검토 - Gitflow는 코드를 다른 환경으로 승격하기 전에 코드 검토 및 승인을 사용하도록 요구하고 장려합니다. 이 프로세스는 모든 코드 프로모션에 대해이 단계를 요구하여 개발자 간의 마찰을 제거합니다.
- 조직 이점 - Gitflow는 조직 수준에서도 이점이 있습니다. Gitflow는 조직이 릴리스 일정을 이해하고 예상하는 데 도움이 되는 표준 릴리스 주기의 사용을 권장합니다. 이제 비즈니스는 새로운 기능을 제공할 수 있는 시기를 이해하므로 전달 날짜가 설정되어 있기 때문에 타임라인에 대한 마찰이 줄어듭니다.

단점

Gitflow 기반 개발에는 개발 프로세스와 팀 역학에 영향을 미칠 수 있는 몇 가지 단점이 있습니다. 다음은 몇 가지 주목할 만한 단점입니다.

- 복잡성 - Gitflow는 새로운 팀이 학습할 수 있는 복잡한 패턴이며 이를 성공적으로 사용하려면 Gitflow 규칙을 준수해야 합니다.
- 지속적 배포 - Gitflow는 많은 배포가 프로덕션에 신속하게 릴리스되는 모델에 적합하지 않습니다. 이는 Gitflow에서 여러 브랜치와 브랜치를 관리하는 엄격한 워크플로를 사용해야 하기 때문입니다`release`.
- 브랜치 관리 - Gitflow는 유지 관리가 어려울 수 있는 여러 브랜치를 사용합니다. 브랜치를 서로 올바르게 정렬하려면 다양한 브랜치를 추적하고 릴리스된 코드를 병합하는 것이 어려울 수 있습니다.
- 기술 부채 - Gitflow 릴리스는 일반적으로 다른 분기 모델보다 느리기 때문에 릴리스에 더 많은 기능이 누적되어 기술 부채가 누적될 수 있습니다.

팀은 Gitflow 기반 개발이 프로젝트에 적합한 접근 방식인지 결정할 때 이러한 단점을 신중하게 고려해야 합니다.

다음 단계

이 가이드에서는 세 가지 일반적인 Git 브랜칭 전략인 플로우, GitHub 깃플로우, 트렁크 간의 차이점을 설명합니다. 워크플로를 자세히 설명하고 각 워크플로의 장단점도 제공합니다. 다음 단계는 조직에 맞는 이러한 표준 워크플로 중 하나를 선택하는 것입니다. 이러한 분기 전략 중 하나를 구현하려면 다음을 참조하십시오.

- 다중 계정 환경을 위한 트렁크 브랜칭 전략 구현 DevOps
- 다중 계정 환경을 위한 GitHub Flow 분기 전략 구현 DevOps
- 다중 계정 환경을 위한 Gitflow 분기 전략 구현 DevOps

DevOps Git과 프로세스를 사용하기 위한 팀의 여정을 어디서부터 시작해야 할지 잘 모르겠다면 표준 솔루션을 선택하여 테스트해 보는 것이 좋습니다. 표준 분기 규칙을 사용하면 팀이 기존 문서를 기반으로 작성하고 자신에게 가장 적합한 것이 무엇인지 알 수 있습니다.

조직 또는 개발 팀에 맞지 않더라도 전략을 변경하는 것을 두려워하지 마세요. 개발 팀의 요구 사항과 요구 사항은 시간이 지남에 따라 변할 수 있으며, 완벽한 단일 솔루션은 없습니다.

리소스

이 가이드에는 Git에 대한 교육이 포함되어 있지 않지만, 이 교육이 필요한 경우 인터넷에서 많은 고품질 리소스를 이용할 수 있습니다. Git 설명서 사이트에서 시작하는 것이 좋습니다.

다음 리소스는 에서의 Git 브랜칭 여정에 도움이 될 수 있습니다. AWS 클라우드

AWS 규범적 지침

- 다중 계정 환경을 위한 트렁크 브랜칭 전략 구현 DevOps
- 다중 계정 환경을 위한 GitHub Flow 분기 전략 구현 DevOps
- 다중 계정 환경을 위한 Gitflow 분기 전략 구현 DevOps

기타 지침 AWS

- AWS DevOps 지침
- AWS 배포 파이프라인 참조 아키텍처
- DevOps란 무엇입니까?
- DevOps리소스

기타 리소스

- 트웰-팩터 앱 방법론 (12factor.net)
- 기프트 시크릿 () GitHub
- 기트플로우
  - 오리지널 Gitflow 블로그 (빈센트 드리센 블로그 게시물)
  - 깃플로우 워크플로 (아틀라시안)
  - Gitflow on GitHub: GitHub 기반 리포지토리에서 Git Flow 워크플로를 사용하는 방법 (비디오) YouTube
  - Git 플로우 초기화 예제 (비디오) YouTube
  - Gitflow 릴리즈 브랜치를 처음부터 끝까지 (비디오) YouTube
- GitHub 플로우

  - GitHub 플로우 퀵스타트 (GitHub 설명서)
  - GitHub 플로우를 선택해야 하는 이유 (GitHub 플로우 웹사이트)
- 트렁크
  - 트렁크 기반 개발 소개 (트렁크 기반 개발 웹 사이트)

기여자

작성

- 마이크 스티븐스, 선임 클라우드 애플리케이션 아키텍트, AWS
- 레이잔 윌슨, 선임 클라우드 애플리케이션 아키텍트, AWS
- 압힐라쉬 비노드, 팀장, 선임 클라우드 애플리케이션 아키텍트 AWS

리뷰

- 스티븐 DiCato, 선임 보안 컨설턴트, AWS
- 가우라브 사무드라, 클라우드 애플리케이션 아키텍트, AWS
- 스티븐 구겐하이머, 팀장, 선임 클라우드 애플리케이션 아키텍트, AWS

테크니컬 라이팅

- 릴리 AbouHarb, 선임 테크니컬 라이터, AWS

문서 기록

아래 표에 이 가이드의 주요 변경 사항이 설명되어 있습니다. 향후 업데이트에 대한 알림을 받으려면 RSS 피드를 구독하십시오.

변경 사항	설명	날짜
업데이트	트렁크 전략의 브랜치, GitHub 흐름 전략의 브랜치, Gitflow 전략 섹션의 브랜치의 이미지를 대체했습니다.	2026년 6월 5일
최초 게시	—	2024년 2월 15일

기계 번역으로 제공되는 번역입니다. 제공된 번역과 원본 영어의 내용이 상충하는 경우에는 영어 버전이 우선합니다.