Amazon EKS

Eksctl User Guide

Eksctl User Guide: Amazon EKS

Table of Contents

What is Eksctl?

eksctl is a command-line utility tool that automates and simplifies the process of creating, managing, and operating Amazon Elastic Kubernetes Service (Amazon EKS) clusters. Written in Go, eksctl provides a declarative syntax through YAML configurations and CLI commands to handle complex EKS cluster operations that would otherwise require multiple manual steps across different AWS services.

eksctl is particularly valuable for DevOps engineers, platform teams, and Kubernetes administrators who need to consistently deploy and manage EKS clusters at scale. It's especially useful for organizations transitioning from self-managed Kubernetes to EKS, or those implementing infrastructure as code (IaC) practices, as it can be integrated into existing CI/CD pipelines and automation workflows. The tool abstracts away many of the complex interactions between AWS services required for EKS cluster setup, such as VPC configuration, IAM role creation, and security group management.

Key features of eksctl include the ability to create fully functional EKS clusters with a single command, support for custom networking configurations, automated node group management, and GitOps workflow integration. The tool manages cluster upgrades, scales node groups, and handles add-on management through a declarative approach. eksctl also provides advanced capabilities such as Fargate profile configuration, managed node group customization, and spot instance integration, while maintaining compatibility with other AWS tools and services through native AWS SDK integration.

Features

The features that are currently implemented are:

- Create, get, list and delete clusters
- Create, drain and delete nodegroups
- Scale a nodegroup
- Update a cluster
- Use custom AMIs
- Configure VPC Networking
- Configure access to API endpoints
- Support for GPU nodegroups

- Spot instances and mixed instances
- IAM Management and Add-on Policies
- List cluster Cloudformation stacks
- Install coredns
- Write kubeconfig file for a cluster

Eksctl FAQ

General

Can I use `eksctl` to manage clusters which weren't created by `eksctl`?

Yes! From version `0.40.0` you can run `eksctl` against any cluster, whether it was created by `eksctl` or not. For more information, see the section called "Non eksctl-created clusters".

Nodegroups

How can I change the instance type of my nodegroup?

From the point of view of `eksctl`, nodegroups are immutable. This means that once created the only thing `eksctl` can do is scale the nodegroup up or down.

To change the instance type, create a new nodegroup with the desired instance type, then drain it so that the workloads move to the new one. After that step is complete you can delete the old nodegroup.

How can I see the generated userdata for a nodegroup?

First you'll need the name of the Cloudformation stack that manages the nodegroup:

```
eksctl utils describe-stacks --region=us-west-2 --cluster NAME
```

You'll see a name similar to `eksctl-CLUSTER_NAME-nodegroup-NODEGROUP_NAME`.

You can execute the following to get the userdata. Note the final line which decodes from base64 and decompresses the gzipped data.

```
NG_STACK=eksctl-scrumptious-monster-1595247364-nodegroup-ng-29b8862f # your stack here
LAUNCH_TEMPLATE_ID=$(aws cloudformation describe-stack-resources --stack-name $NG_STACK
 \
```

```
| jq -r '.StackResources | map(select(.LogicalResourceId == "NodeGroupLaunchTemplate")
 \
| .PhysicalResourceId)[0]')
aws ec2 describe-launch-template-versions --launch-template-id $LAUNCH_TEMPLATE_ID \
| jq -r '.LaunchTemplateVersions[0].LaunchTemplateData.UserData' \
| base64 -d | gunzip
```

Ingress

How do I set up ingress with `eksctl`?

We recommend using the AWS Load Balancer Controller. Documentation on how to deploy the controller to your cluster, as well as how to migrate from the old ALB Ingress Controller, can be found here.

For the Nginx Ingress Controller, setup would be the same as any on other Kubernetes cluster.

Kubectl

I'm using an HTTPS proxy and cluster certificate validation fails, how can I use the system CAs?

Set the environment variable `KUBECONFIG_USE_SYSTEM_CA` to make `kubeconfig` respect the system certificate authorities.

Dry Run

The dry-run feature allows you to inspect and change the instances matched by the instance selector before proceeding to creating a nodegroup.

When `eksctl create cluster` is called with the instance selector options and `--dry-run`, eksctl will output a ClusterConfig file containing a nodegroup representing the CLI options and the instance types set to the instances matched by the instance selector resource criteria.

```
eksctl create cluster --name development --dry-run


apiVersion: eksctl.io/v1alpha5
cloudWatch:
  clusterLogging: {}
iam:
  vpcResourceControllerPolicy: true
  withOIDC: false
```

```
kind: ClusterConfig
managedNodeGroups:
- amiFamily: AmazonLinux2
  desiredCapacity: 2
  disableIMDSv1: true
  disablePodIMDS: false
  iam:
    withAddonPolicies:
      albIngress: false
      appMesh: false
      appMeshPreview: false
      autoScaler: false
      certManager: false
      cloudWatch: false
      ebs: false
      efs: false
      externalDNS: false
      fsx: false
      imageBuilder: false
      xRay: false
  instanceSelector: {}
  instanceType: m5.large
  labels:
    alpha.eksctl.io/cluster-name: development
    alpha.eksctl.io/nodegroup-name: ng-4aba8a47
  maxSize: 2
  minSize: 2
  name: ng-4aba8a47
  privateNetworking: false
  securityGroups:
    withLocal: null
    withShared: null
  ssh:
    allow: false
    enableSsm: false
    publicKeyPath: ""
  tags:
    alpha.eksctl.io/nodegroup-name: ng-4aba8a47
    alpha.eksctl.io/nodegroup-type: managed
  volumeIOPS: 3000
  volumeSize: 80
  volumeThroughput: 125
  volumeType: gp3
metadata:
```

```
  name: development
  region: us-west-2
  version: "1.24"
privateCluster:
  enabled: false
vpc:
  autoAllocateIPv6: false
  cidr: 192.168.0.0/16
  clusterEndpoints:
    privateAccess: false
    publicAccess: true
  manageSharedNodeSecurityGroupRules: true
  nat:
    gateway: Single
```

The generated ClusterConfig can then be passed to `eksctl create cluster`:

```
eksctl create cluster -f generated-cluster.yaml
```

When a ClusterConfig file is passed with `--dry-run`, eksctl will output a ClusterConfig file containing the values set in the file.

One-off Options in eksctl

There are certain one-off options that cannot be represented in the `ClusterConfig` file, e.g., `--install-vpc-controllers`.

It is expected that:

```
eksctl create cluster --<options...> --dry-run > config.yaml
```

followed by:

```
eksctl create cluster -f config.yaml
```

would be equivalent to running the first command without `--dry-run`.

eksctl therefore disallows passing options that cannot be represented in the config file when `--dry-run` is passed.

Important

If you need to pass an AWS profile, set the `AWS_PROFILE` environment variable, instead of passing the `--profile` CLI option.

Tutorial

This topic walks you through installing and configuring eksctl, then using it to create an Amazon EKS cluster.

Step 1: Install eksctl

Complete the following steps to download and install the latest version of eksctl on your Linux or macOS device:

To install eksctl with Homebrew

1. (Prerequisite) Install Homebrew.
2. Add the AWS tap:

```
brew tap aws/tap
```

3. Install eksctl

```
brew install aws/tap/eksctl
```

Before using eksctl, complete these configuration steps:

1. Install prerequisites:
   - Install AWS CLI version 2.x or later.
   - Install kubectl using Homebrew:

```
brew install kubernetes-cli
```

2. Configure AWS credentials in your environment:

```
aws configure
```

3. Verify AWS CLI configuration:

```
aws sts get-caller-identity
```

Step 2: Create cluster config file

Create a cluster configuration file using these steps:

1. Create a new file named `cluster.yaml`:

```
touch cluster.yaml
```

2. Add the following basic cluster configuration:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
   name: basic-cluster
   region: us-west-2

nodeGroups:
   - name: ng-1
     instanceType: m5.large
     desiredCapacity: 2
     minSize: 1
     maxSize: 3
     ssh:
        allow: false
```

3. Customize the configuration:
   - Update the `region` to match your desired AWS region.
   - Modify the `instanceType` based on your workload requirements.
   - Adjust the `desiredCapacity`, `minSize`, and `maxSize` according to your needs.
4. Validate the configuration file:

```
eksctl create cluster -f cluster.yaml --dry-run
```

Step 3: Create cluster

Follow these steps to create your EKS cluster:

1. Create the cluster using the configuration file:

```
eksctl create cluster -f cluster.yaml
```

2. Wait for cluster creation (this typically takes 15-20 minutes).
3. Verify cluster creation:

```
eksctl get cluster
```

4. Configure kubectl to use your new cluster:

```
aws eks update-kubeconfig --name basic-cluster --region us-west-2
```

5. Verify cluster connectivity:

```
kubectl get nodes
```

Your cluster is now ready to use.

Optional: Delete Cluster

Remember to delete the cluster when you're done to avoid unnecessary charges:

```
eksctl delete cluster -f cluster.yaml
```

> **Note**
>
> Cluster creation can incur AWS charges. Make sure to review the Amazon EKS pricing before creating a cluster.

Next Steps

- Configure Kubectl to connect to the cluster
- Deploy a sample app

Installation options for Eksctl

`eksctl` is available to install from official releases as described below. We recommend that you install `eksctl` from only the official GitHub releases. You may opt to use a third-party installer, but please be advised that AWS does not maintain nor support these methods of installation. Use them at your own discretion.

Prerequisite

You will need to have AWS API credentials configured. What works for AWS CLI or any other tools (kops, Terraform, etc.) should be sufficient. You can use `~/.aws/credentials` file or environment variables. For more information, see the AWS CLI Reference.

You will also need AWS IAM Authenticator for Kubernetes command (either `aws-iam-authenticator` or `aws eks get-token` (available in version 1.16.156 or greater of AWS CLI) in your PATH.

The IAM account used for EKS cluster creation should have these minimal access levels.

AWS Service	Access Level
CloudFormation	Full Access
EC2	**Full:** Tagging **Limited:** List, Read, Write
EC2 Auto Scaling	**Limited:** List, Write
EKS	Full Access
IAM	**Limited:** List, Read, Write, Permissions Management
Systems Manager	**Limited:** List, Read

For Unix

To download the latest release, run:

```
# for ARM systems, set ARCH to: `arm64`, `armv6` or `armv7`
ARCH=amd64
PLATFORM=$(uname -s)_$ARCH

curl -sLO "https://github.com/eksctl-io/eksctl/releases/latest/download/eksctl_
$PLATFORM.tar.gz"

# (Optional) Verify checksum
curl -sL "https://github.com/eksctl-io/eksctl/releases/latest/download/
eksctl_checksums.txt" | grep $PLATFORM | sha256sum --check

tar -xzf eksctl_$PLATFORM.tar.gz -C /tmp && rm eksctl_$PLATFORM.tar.gz

sudo install -m 0755 /tmp/eksctl /usr/local/bin && rm /tmp/eksctl
```

For Windows

Direct download (latest release):

- AMD64/x86_64
- ARMv6
- ARMv7
- ARM64

Make sure to unzip the archive to a folder in the PATH variable.

Optionally, verify the checksum:

1. Download the checksum file: latest
2. Use Command Prompt to manually compare `CertUtil`'s output to the checksum file downloaded.

   ```
   REM Replace amd64 with armv6, armv7 or arm64
   CertUtil -hashfile eksctl_Windows_amd64.zip SHA256
   ```

3. Using PowerShell to automate the verification using the `-eq` operator to get a `True` or `False` result:

   ```
   # Replace amd64 with armv6, armv7 or arm64
   ```

```
 (Get-FileHash -Algorithm SHA256 .\eksctl_Windows_amd64.zip).Hash -eq ((Get-Content .
\eksctl_checksums.txt) -match 'eksctl_Windows_amd64.zip' -split ' ')[0]
```

Using Git Bash:

```
# for ARM systems, set ARCH to: `arm64`, `armv6` or `armv7`
ARCH=amd64
PLATFORM=windows_$ARCH

curl -sLO "https://github.com/eksctl-io/eksctl/releases/latest/download/eksctl_
$PLATFORM.zip"

# (Optional) Verify checksum
curl -sL "https://github.com/eksctl-io/eksctl/releases/latest/download/
eksctl_checksums.txt" | grep $PLATFORM | sha256sum --check

unzip eksctl_$PLATFORM.zip -d $HOME/bin

rm eksctl_$PLATFORM.zip
```

The `eksctl` executable is placed in `$HOME/bin`, which is in `$PATH` from Git Bash.

Homebrew

You can use Homebrew to install software on MacOS and Linux.

AWS maintains a Homebrew tap including eksctl.

For more information about the Homebrew tap, see the project on Github and the Homebrew formula for eksctl.

To install eksctl with Homebrew

1. (Prerequisite) Install Homebrew
2. Add the AWS tap

```
brew tap aws/tap
```

3. Install eksctl

```
brew install aws/tap/eksctl
```

Docker

For every release and RC a container image is pushed to ECR repository `public.ecr.aws/eksctl/eksctl`. Learn more about the usage on ECR Public Gallery - eksctl. For example,

```
docker run --rm -it public.ecr.aws/eksctl/eksctl version
```

Shell Completion

Bash

To enable bash completion, run the following, or put it in `~/.bashrc` or `~/.profile`:

```
. <(eksctl completion bash)
```

Zsh

For zsh completion, please run:

```
mkdir -p ~/.zsh/completion/
eksctl completion zsh > ~/.zsh/completion/_eksctl
```

and put the following in `~/.zshrc`:

```
fpath=($fpath ~/.zsh/completion)
```

Note if you're not running a distribution like oh-my-zsh you may first have to enable autocompletion (and put in `~/.zshrc` to make it persistent):

```
autoload -U compinit
compinit
```

Fish

The below commands can be used for fish auto completion:

```
mkdir -p ~/.config/fish/completions
eksctl completion fish > ~/.config/fish/completions/eksctl.fish
```

Powershell

The below command can be referred for setting it up. Please note that the path might be different depending on your system settings.

```
eksctl completion powershell > C:\Users\Documents\WindowsPowerShell\Scripts\eksctl.ps1
```

Updates

> **Important**
>
> If you install eksctl by directly downloading it (not using a package manager) you need to manually update it.

Clusters

This chapter covers creating and configuring EKS clusters using eksctl. It also includes add-ons and EKS Auto Mode.

Topics:

- Start quickly with basic EKS clusters using default managed nodegroups
- Create customized clusters using config files with specific configurations
- Deploy clusters in existing VPCs with private networking and custom IAM policies

- the section called “Configure kubelet”
  - Prevent node resource starvation by configuring kubelet and system daemon reservations
  - Customize eviction thresholds for memory and filesystem availability
  - Enable or disable specific kubelet feature gates across node groups
- the section called “EKS Connector”
  - Centralize management of hybrid Kubernetes deployments through EKS Console
  - Configure IAM roles and permissions for external cluster access
  - Remove external clusters and cleanup associated AWS resources
- the section called “EKS Fully-Private Cluster”
  - Meet security requirements with fully-private EKS clusters having no outbound internet access
  - Configure private access to AWS services through VPC endpoints
  - Create and manage private nodegroups with explicit networking settings
- the section called “Karpenter Support”
  - Automate node provisioning
  - Create custom Karpenter provisioner configurations
  - Set up Karpenter with spot instance interruption handling
- the section called “Amazon EMR”
  - Create IAM identity mapping between EMR and EKS cluster
- the section called “EKS Fargate Support”
  - Define custom Fargate profiles for pod scheduling
  - Manage Fargate profiles through creation and configuration updates
- the section called “Non eksctl-created clusters”
  - Standardize management of clusters created outside eksctl
  - Use eksctl commands on existing non-eksctl clusters
- the section called “Enable Zonal Shift”
  - Improve application availability by enabling rapid zone failover capabilities
  - Configure zonal shift on new EKS cluster deployments

- Enable zonal shift features on existing EKS clusters

Creating and managing clusters

This topic covers how to create and delete EKS clusters using Eksctl. You can create clusters with a CLI command, or by creating a cluster configuration YAML file.

Creating a simple cluster

Create a simple cluster with the following command:

```
eksctl create cluster
```

That will create an EKS cluster in your default region (as specified by your AWS CLI configuration) with one managed nodegroup containing two m5.large nodes.

eksctl now creates a managed nodegroup by default when a config file isn't used. To create a self-managed nodegroup, pass `--managed=false` to `eksctl create cluster` or `eksctl create nodegroup`.

Considerations

- When creating clusters in `us-east-1`, you might encounter an `UnsupportedAvailabilityZoneException`. If this happens, copy the suggested zones and pass the `--zones` flag, for example: `eksctl create cluster --region=us-east-1 --zones=us-east-1a,us-east-1b,us-east-1d`. This issue may occur in other regions but is less common. In most cases, you won't need to use the `--zone` flag.

Create cluster using config file

You can create a cluster using a config file instead of flags.

First, create `cluster.yaml` file:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
```

```
metadata:
  name: basic-cluster
  region: eu-north-1

nodeGroups:
  - name: ng-1
    instanceType: m5.large
    desiredCapacity: 10
    volumeSize: 80
    ssh:
      allow: true # will use ~/.ssh/id_rsa.pub as the default ssh key
  - name: ng-2
    instanceType: m5.xlarge
    desiredCapacity: 2
    volumeSize: 100
    ssh:
      publicKeyPath: ~/.ssh/ec2_id_rsa.pub
```

Next, run this command:

```
eksctl create cluster -f cluster.yaml
```

This will create a cluster as described.

If you needed to use an existing VPC, you can use a config file like this:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: cluster-in-existing-vpc
  region: eu-north-1

vpc:
  subnets:
    private:
      eu-north-1a: { id: subnet-0ff156e0c4a6d300c }
      eu-north-1b: { id: subnet-0549cdab573695c03 }
      eu-north-1c: { id: subnet-0426fb4a607393184 }

nodeGroups:
  - name: ng-1-workers
    labels: { role: workers }
```

```
    instanceType: m5.xlarge
    desiredCapacity: 10
    privateNetworking: true
  - name: ng-2-builders
    labels: { role: builders }
    instanceType: m5.2xlarge
    desiredCapacity: 2
    privateNetworking: true
    iam:
      withAddonPolicies:
        imageBuilder: true
```

The cluster name or nodegroup name must contain only alphanumeric characters (case-sensitive) and hyphens. It must start with an alphabetic character and can't exceed 128 characters, or you will receive a validation error. For more information, see Create a stack from the CloudFormation console in the AWS CloudFormation user guide.

Update kubeconfig for new cluster

After the cluster has been created, the appropriate kubernetes configuration will be added to your kubeconfig file. This is, the file that you have configured in the environment variable `KUBECONFIG` or `~/.kube/config` by default. The path to the kubeconfig file can be overridden using the `--kubeconfig` flag.

Other flags that can change how the kubeconfig file is written:

flag	type	use	default value
--kubeconfig	string	path to write kubeconfig (incompatible with --auto-kubeconfig)	$KUBECONFIG or ~/.kube/config
--set-kubeconfig-context	bool	if true then current-context will be set in kubeconfig; if a context is already set then it will be overwritten	true

flag	type	use	default value
--auto-kubeconfig	bool	save kubeconfig file by cluster name	true
--write-kubeconfig	bool	toggle writing of kubeconfig	true

Delete cluster

To delete this cluster, run:

```
eksctl delete cluster -f cluster.yaml
```

> **Warning**
>
> Use the `--wait` flag with delete operations to ensure deletion errors are properly reported.

Without the `--wait` flag, eksctl will only issue a delete operation to the cluster's CloudFormation stack and won't wait for its deletion. In some cases, AWS resources using the cluster or its VPC may cause cluster deletion to fail. If your delete fails or you forget the wait flag, you may have to go to the CloudFormation GUI and delete the eks stacks from there.

> **Warning**
>
> PDB policies may block node removal during cluster deletion.

When deleting a cluster with nodegroups, Pod Disruption Budget (PDB) policies can prevent nodes from being removed successfully. For example, clusters with `aws-ebs-csi-driver` installed typically have two pods with a PDB policy allowing only one pod to be unavailable at a time, making the other pod unevictable during deletion. To successfully delete the cluster in these scenarios, use the `disable-nodegroup-eviction` flag to bypass PDB policy checks:

```
eksctl delete cluster -f cluster.yaml --disable-nodegroup-eviction
```

See the examples/ directory in the eksctl GitHub repo for more sample config files.

Dry Run

The dry-run feature enables generating a ClusterConfig file that skips cluster creation and outputs a ClusterConfig file that represents the supplied CLI options and contains the default values set by eksctl.

More info can be found on the Dry Run page.

EKS Auto Mode

eksctl supports EKS Auto Mode, a feature that extends AWS management of Kubernetes clusters beyond the cluster itself, to allow AWS to also set up and manage the infrastructure that enables the smooth operation of your workloads. This allows you to delegate key infrastructure decisions and leverage the expertise of AWS for day-to-day operations. Cluster infrastructure managed by AWS includes many Kubernetes capabilities as core components, as opposed to add-ons, such as compute autoscaling, pod and service networking, application load balancing, cluster DNS, block storage, and GPU support.

Creating an EKS cluster with Auto Mode enabled

`eksctl` has added a new `autoModeConfig` field to enable and configure Auto Mode. The shape of the `autoModeConfig` field is

```
autoModeConfig:
    # defaults to false
    enabled: boolean
    # optional, defaults to [general-purpose, system].
    # To disable creation of nodePools, set it to the empty array ([]).
    nodePools: []string
    # optional, eksctl creates a new role if this is not supplied
    # and nodePools are present.
    nodeRoleARN: string
```

If `autoModeConfig.enabled` is true, eksctl creates an EKS cluster by passing `computeConfig.enabled: true`, `kubernetesNetworkConfig.elasticLoadBalancing.enabled: true`, and

`storageConfig.blockStorage.enabled: true` to the EKS API, enabling management of data plane components like compute, storage and networking.

To create an EKS cluster with Auto Mode enabled, set `autoModeConfig.enabled: true`, as in

```
# auto-mode-cluster.yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
    name: auto-mode-cluster
    region: us-west-2

autoModeConfig:
    enabled: true
```

```
eksctl create cluster -f auto-mode-cluster.yaml
```

eksctl creates a node role to use for nodes launched by Auto Mode. eksctl also creates the `general-purpose` and `system` node pools. To disable creation of the default node pools, e.g., to configure your own node pools that use a different set of subnets, set `nodePools: []`, as in

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
    name: auto-mode-cluster
    region: us-west-2

autoModeConfig:
    enabled: true
    nodePools: [] # disables creation of default node pools.
```

Updating an EKS cluster to use Auto Mode

To update an existing EKS cluster to use Auto Mode, run

```
# cluster.yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
    name: cluster
    region: us-west-2
```

```
autoModeConfig:
    enabled: true
```

```
eksctl update auto-mode-config -f cluster.yaml
```

> **Note**
>
> If the cluster was created by eksctl, and it uses public subnets as cluster subnets, Auto Mode will launch nodes in public subnets. To use private subnets for worker nodes launched by Auto Mode, update the cluster to use private subnets.

Disabling Auto Mode

To disable Auto Mode, set `autoModeConfig.enabled: false` and run

```
# cluster.yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
    name: auto-mode-cluster
    region: us-west-2

autoModeConfig:
    enabled: false
```

```
eksctl update auto-mode-config -f cluster.yaml
```

Further information

- EKS Auto Mode

EKS Access Entries

You can use eksctl to manage EKS Access Entries. Use access entries to grant Kubernetes permissions to AWS IAM Identities. For example, you might grant a developer role permission to read Kubernetes resources in a cluster.

This topic covers how to use eksctl to manage access entries. For general information about access entries, see Grant IAM users access to Kubernetes with EKS access entries.

You can attach Kubernetes access policies defined by AWS, or associate an IAM Identity with a Kubernetes group.

For more information about the available pre-defined policies, see Associate access policies with access entries.

If you need to define custom Kubernetes policies, associate the IAM Identity with a Kubernetes group, and grant permissions to that group.

Cluster authentication mode

You can only use access entries if the authentication mode of the cluster permits it.

For more information, see Set Cluster Authentication Mode

Set authentication mode with a YAML file

`eksctl` has added a new `accessConfig.authenticationMode` field under ClusterConfig, which can be set to one of the following three values:

- `CONFIG_MAP` - default in EKS API - only `aws-auth` ConfigMap will be used
- `API` - only access entries API will be used
- `API_AND_CONFIG_MAP` - default in `eksctl` - both `aws-auth` ConfigMap and access entries API can be used

Set authentication mode in ClusterConfig YAML:

```
accessConfig:
  authenticationMode: <>
```

Update authentication mode with a command

If you want to use access entries on an already existing, non-eksctl created, cluster, where `CONFIG_MAP` option is used, the user will need to first set `authenticationMode` to `API_AND_CONFIG_MAP`. For that, `eksctl` has introduced a new command for updating the cluster authentication mode, which works both with CLI flags e.g.

```
eksctl utils update-authentication-mode --cluster my-cluster --authentication-mode
 API_AND_CONFIG_MAP
```

Access Entry Resources

Access entries have a type, such as `STANDARD` or `EC2_LINUX`. The type depends on how you are using the access entry.

- The `standard` type is for granting Kubernetes permissions to IAM Users and IAM Roles.
  - For example, you can view Kubernetes resources in the AWS console by attaching an access policy to the Role or User you use to access the console.
- The `EC2_LINUX` and `EC2_WINDOWS` types are for granting Kubernetes permissions to EC2 instances. Instances use these permissions to join the cluster.

For more information about the *types* of access entries, see Create access entries

IAM Entities

You can use access entries to grant Kubernetes permissions to IAM Identities such as IAM Users and IAM Roles.

Use the `accessConfig.accessEntries` field to associate the ARN of an IAM resource with a Access Entries EKS API. For example:

```
accessConfig:
  authenticationMode: API_AND_CONFIG_MAP
  accessEntries:
    - principalARN: arn:aws:iam::111122223333:user/my-user-name
      type: STANDARD
      kubernetesGroups: # optional Kubernetes groups
        - group1 # groups can used to give permissions via RBAC
        - group2

    - principalARN: arn:aws:iam::111122223333:role/role-name-1
      accessPolicies: # optional access polices
        - policyARN: arn:aws:eks::aws:cluster-access-policy/AmazonEKSViewPolicy
          accessScope:
            type: namespace
            namespaces:
              - default
```

```
            - my-namespace
            - dev-*

  - principalARN: arn:aws:iam::111122223333:role/admin-role
    accessPolicies: # optional access polices
      - policyARN: arn:aws:eks::aws:cluster-access-policy/AmazonEKSClusterAdminPolicy
        accessScope:
          type: cluster

  - principalARN: arn:aws:iam::111122223333:role/role-name-2
    type: EC2_LINUX
```

In addition to associating EKS policies, one can also specify the Kubernetes groups to which an IAM entity belongs, thus granting permissions via RBAC.

Managed nodegroups and Fargate

The integration with access entries for these resources will be achieved behind the scenes, by the EKS API. Newly created managed node groups and Fargate pods will create API access entries, rather than using pre-loaded RBAC resources. Existing node groups and Fargate pods will not be changed, and continue to rely on the entries in the aws-auth config map.

Self-managed nodegroups

Each access entry has a type. For authorizing self-managed nodegroups, `eksctl` will create a unique access entry for each nodegroup with the principal ARN set to the node role ARN and type set to either `EC2_LINUX` or `EC2_WINDOWS` depending on nodegroup amiFamily.

When creating your own access entries, you can also specify `EC2_LINUX` (for an IAM role used with Linux or Bottlerocket self-managed nodes), `EC2_WINDOWS` (for IAM roles used with Windows self-managed nodes), `FARGATE_LINUX` (for IAM roles used with AWS Fargate (Fargate)), or `STANDARD` as a type. If you don't specify a type, the default type is set to `STANDARD`.

> **Note**
>
> When deleting a nodegroup created with a pre-existing `instanceRoleARN`, it is the user's responsibility to delete the corresponding access entry when no more nodegroups are associated with it. This is because eksctl does not attempt to find out if an access entry is still in use by non-eksctl created self-managed nodegroups as it is a complicated process.

Create access entry

This can be done in two different ways, either during cluster creation, specifying the desired access entries as part of the config file and running:

```
eksctl create cluster -f config.yaml
```

OR post cluster creation, by running:

```
eksctl create accessentry -f config.yaml
```

For an example config file for creating access entries, see 40-access-entries.yaml in the eksctl GitHub repo.

Get access entry

The user can retrieve all access entries associated with a certain cluster by running one of the following:

```
eksctl get accessentry -f config.yaml
```

OR

```
eksctl get accessentry --cluster my-cluster
```

Alternatively, to retrieve only the access entry corresponding to a certain IAM entity one shall use the `--principal-arn` flag. e.g.

```
eksctl get accessentry --cluster my-cluster --principal-arn
 arn:aws:iam::111122223333:user/admin
```

Delete access entry

To delete a single access entry at a time use:

```
eksctl delete accessentry --cluster my-cluster --principal-arn
 arn:aws:iam::111122223333:user/admin
```

To delete multiple access entries, use the `--config-file` flag and specify all the `principalARN's` corresponding with the access entries, under the top-level `accessEntry` field, e.g.

```
...
accessEntry:
  - principalARN: arn:aws:iam::111122223333:user/my-user-name
  - principalARN: arn:aws:iam::111122223333:role/role-name-1
  - principalARN: arn:aws:iam::111122223333:role/admin-role
```

```
eksctl delete accessentry -f config.yaml
```

Migrate from aws-auth ConfigMap

The user can migrate their existing IAM identities from `aws-auth` configmap to access entries by running the following:

```
eksctl utils migrate-to-access-entry --cluster my-cluster --target-authentication-mode
 <API or API_AND_CONFIG_MAP>
```

When `--target-authentication-mode` flag is set to `API`, authentication mode is switched to `API` mode (skipped if already in `API` mode), IAM identity mappings will be migrated to access entries, and `aws-auth` configmap is deleted from the cluster.

When `--target-authentication-mode` flag is set to `API_AND_CONFIG_MAP`, authentication mode is switched to `API_AND_CONFIG_MAP` mode (skipped if already in `API_AND_CONFIG_MAP` mode), IAM identity mappings will be migrated to access entries, but `aws-auth` configmap is preserved.

> **Note**
>
> When `--target-authentication-mode` flag is set to `API`, this command will not update authentication mode to `API` mode if `aws-auth` configmap has one of the below constraints.

- There is an Account level identity mapping.

- One or more Roles/Users are mapped to the kubernetes group(s) which begin with prefix `system:` (except for EKS specific groups i.e. `system:masters`, `system:bootstrappers`, `system:nodes` etc).
- One or more IAM identity mapping(s) are for a [Service Linked Role](link:IAM/latest/UserGuide/using-service-linked-roles.html).

Disable cluster creator admin permissions

`eksctl` has added a new field `accessConfig.bootstrapClusterCreatorAdminPermissions: boolean` that, when set to false, disables granting cluster-admin permissions to the IAM identity creating the cluster. i.e.

add the option to the config file:

```
accessConfig:
  bootstrapClusterCreatorAdminPermissions: false
```

and run:

```
eksctl create cluster -f config.yaml
```

Non eksctl-created clusters

You can run `eksctl` commands against clusters which were not created by `eksctl`.

> **Note**
>
> Eksctl can only support unowned clusters with names which are compatible with AWS CloudFormation. Any cluster names which do not match this will fail CloudFormation API validation check.

Supported commands

The following commands can be used against clusters created by any means other than `eksctl`. The commands, flags and config file options can be used in exactly the same way.

If we have missed some functionality, please let us know.

- ✓ Create:
  - ✓ `eksctl create nodegroup` (see note below)
  - ✓ `eksctl create fargateprofile`
  - ✓ `eksctl create iamserviceaccount`
  - ✓ `eksctl create iamidentitymapping`
- ✓ Get:
  - ✓ `eksctl get clusters/cluster`
  - ✓ `eksctl get fargateprofile`
  - ✓ `eksctl get nodegroup`
  - ✓ `eksctl get labels`
- ✓ Delete:
  - ✓ `eksctl delete cluster`
  - ✓ `eksctl delete nodegroup`
  - ✓ `eksctl delete fargateprofile`
  - ✓ `eksctl delete iamserviceaccount`
  - ✓ `eksctl delete iamidentitymapping`
- ✓ Upgrade:
  - ✓ `eksctl upgrade cluster`
  - ✓ `eksctl upgrade nodegroup`
- ✓ Set/Unset:
  - ✓ `eksctl set labels`
  - ✓ `eksctl unset labels`
- ✓ Scale:
  - ✓ `eksctl scale nodegroup`
- ✓ Drain:
  - ✓ `eksctl drain nodegroup`
- ✓ Enable:
  - ✓ `eksctl enable profile`
  - ✓ `eksctl enable repo`

✓ Utils:

- ✓ `eksctl utils associate-iam-oidc-provider`
- ✓ `eksctl utils describe-stacks`
- ✓ `eksctl utils install-vpc-controllers`
- ✓ `eksctl utils nodegroup-health`
- ✓ `eksctl utils set-public-access-cidrs`
- ✓ `eksctl utils update-cluster-endpoints`
- ✓ `eksctl utils update-cluster-logging`
- ✓ `eksctl utils write-kubeconfig`
- ✓ `eksctl utils update-coredns`
- ✓ `eksctl utils update-aws-node`
- ✓ `eksctl utils update-kube-proxy`

Creating nodegroups

`eksctl create nodegroup` is the only command which requires specific input from the user.

Since users can create their clusters with any networking configuration they like, for the time-being, `eksctl` will not attempt to retrieve or guess these values. This may change in the future as we learn more about how people are using this command on non eksctl-created clusters.

This means that in order to create nodegroups or managed nodegroups on a cluster which was not created by `eksctl`, a config file containing VPC details must be provided. At a minimum:

```
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: non-eksctl-created-cluster
  region: us-west-2

vpc:
  id: "vpc-12345"
  securityGroup: "sg-12345"    # this is the ControlPlaneSecurityGroup
  subnets:
    private:
```

```
      private1:
          id: "subnet-12345"
      private2:
          id: "subnet-67890"
    public:
      public1:
          id: "subnet-12345"
      public2:
          id: "subnet-67890"

...
```

For more information on VPC configuration options, see Networking.

Registering non-EKS clusters with EKS Connector

You can use the EKS Connector to view clusters outside of AWS in the EKS Console. This process requires registering the cluster with EKS and running the EKS Connector agent on the external Kubernetes cluster.

`eksctl` simplifies registering non-EKS clusters by creating the required AWS resources and generating Kubernetes manifests for EKS Connector to apply to the external cluster.

Register Cluster

To register or connect a non-EKS Kubernetes cluster, run

```
eksctl register cluster --name <name> --provider <provider>
2021-08-19 13:47:26 [#]  creating IAM role "eksctl-20210819194112186040"
2021-08-19 13:47:26 [#]  registered cluster "<name>" successfully
2021-08-19 13:47:26 [#]  wrote file eks-connector.yaml to <current directory>
2021-08-19 13:47:26 [#]  wrote file eks-connector-clusterrole.yaml to <current
 directory>
2021-08-19 13:47:26 [#]  wrote file eks-connector-console-dashboard-full-access-
group.yaml to <current directory>
2021-08-19 13:47:26 [!]  note: "eks-connector-clusterrole.yaml" and "eks-connector-
console-dashboard-full-access-group.yaml" give full EKS Console access to IAM identity
 "<aws-arn>", edit if required; read https://eksctl.io/usage/eks-connector for more
 info
2021-08-19 13:47:26 [#]  run `kubectl apply -f eks-connector.yaml,eks-connector-
clusterrole.yaml,eks-connector-console-dashboard-full-access-group.yaml` before
 <expiry> to connect the cluster
```

This command will register the cluster and write three files that contain the Kubernetes manifests for EKS Connector that must be applied to the external cluster before the registration expires.

> **Note**
>
> `eks-connector-clusterrole.yaml` and `eks-connector-console-dashboard-full-access-clusterrole.yaml` give `get` and `list` permissions for Kubernetes resources in all namespaces to the calling IAM identity and must be edited accordingly if required before applying them to the cluster. To configure more restricted access, see Granting access to a user to view a cluster.

To provide an existing IAM role to use for EKS Connector, pass it via `--role-arn` as in:

```
eksctl register cluster --name <name> --provider <provider> --role-arn=<role-arn>
```

If the cluster already exists, eksctl will return an error.

Deregister cluster

To deregister or disconnect a registered cluster, run

```
eksctl deregister cluster --name <name>
2021-08-19 16:04:09 [#]  unregistered cluster "<name>" successfully
2021-08-19 16:04:09 [#]  run `kubectl delete namespace eks-connector` and `kubectl
 delete -f eks-connector-binding.yaml` on your cluster to remove EKS Connector
 resources
```

This command will deregister the external cluster and remove its associated AWS resources, but you are required to remove the EKS connector Kubernetes resources from the cluster.

Further information

- EKS Connector

Customizing kubelet configuration

System resources can be reserved through the configuration of the kubelet. This is recommended, because in the case of resource starvation the kubelet might not be able to evict pods

and eventually make the node become `NotReady`. To do this, config files can include the `kubeletExtraConfig` field which accepts a free form yaml that will be embedded into the `kubelet.yaml`.

Some fields in the `kubelet.yaml` are set by eksctl and therefore are not overwritable, such as the `address`, `clusterDomain`, `authentication`, `authorization`, or `serverTLSBootstrap`.

The following example config file creates a nodegroup that reserves `300m` vCPU, `300Mi` of memory and `1Gi` of ephemeral-storage for the kubelet; `300m` vCPU, `300Mi` of memory and `1Gi`of ephemeral storage for OS system daemons; and kicks in eviction of pods when there is less than `200Mi` of memory available or less than 10% of the root filesystem.

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: dev-cluster-1
  region: eu-north-1

nodeGroups:
  - name: ng-1
    instanceType: m5a.xlarge
    desiredCapacity: 1
    kubeletExtraConfig:
        kubeReserved:
            cpu: "300m"
            memory: "300Mi"
            ephemeral-storage: "1Gi"
        kubeReservedCgroup: "/kube-reserved"
        systemReserved:
            cpu: "300m"
            memory: "300Mi"
            ephemeral-storage: "1Gi"
        evictionHard:
            memory.available:  "200Mi"
            nodefs.available: "10%"
        featureGates:
            RotateKubeletServerCertificate: true # has to be enabled, otherwise it will
 be disabled
```

In this example, given instances of type `m5a.xlarge` which have 4 vCPUs and 16GiB of memory, the `Allocatable` amount of CPUs would be 3.4 and 15.4 GiB of memory. It is important to know

that the values specified in the config file for the fields in `kubeletExtraConfig` will completely overwrite the default values specified by eksctl. However, omitting one or more `kubeReserved` parameters will cause the missing parameters to be defaulted to sane values based on the aws instance type being used.

kubeReserved calculation

While it is generally recommended to configure a mixed instance NodeGroup to use instances with the same CPU and RAM configuration; that's not a strict requirement. Therefore the `kubeReserved` calculation uses the *smallest instance* in the `InstanceDistribution.InstanceTypes` field. This way NodeGroups with disparate instance types will not reserve too many resources on the smallest instance. However, this could lead to a reservation that is too small for the largest instance type.

> **Warning**
>
> By default `eksctl` sets `featureGates.RotateKubeletServerCertificate=true`, but when custom `featureGates` are provided, it will be unset. You should always include `featureGates.RotateKubeletServerCertificate=true`, unless you have to disable it.

CloudWatch logging

This topic explains how to configure Amazon CloudWatch logging for your EKS cluster's control plane components. CloudWatch logging provides visibility into your cluster's control plane operations, which is essential for troubleshooting issues, auditing cluster activities, and monitoring the health of your Kubernetes components.

Enabling CloudWatch logging

CloudWatch logging for EKS control plane is not enabled by default due to data ingestion and storage costs.

To enable control plane logging when cluster is created, you will need to define **`cloudWatch.clusterLogging.enableTypes`** setting in your `ClusterConfig` (see below for examples).

So if you have a config file with correct **`cloudWatch.clusterLogging.enableTypes`** setting, you can create a cluster with `eksctl create cluster --config-file=<path>`.

If you have created a cluster already, you can use `eksctl utils update-cluster-logging`.

> **Note**
>
> this command runs in plan mode by default, you will need to specify `--approve` flag to apply the changes to your cluster.

If you are using a config file, run:

```
eksctl utils update-cluster-logging --config-file=<path>
```

Alternatively, you can use CLI flags.

To enable all types of logs, run:

```
eksctl utils update-cluster-logging --enable-types all
```

To enable `audit` logs, run:

```
eksctl utils update-cluster-logging --enable-types audit
```

To enable all but `controllerManager` logs, run:

```
eksctl utils update-cluster-logging --enable-types=all --disable-
types=controllerManager
```

If the `api` and `scheduler` log types were already enabled, to disable `scheduler` and enable `controllerManager` at the same time, run:

```
eksctl utils update-cluster-logging --enable-types=controllerManager --disable-
types=scheduler
```

This will leave `api` and `controllerManager` as the only log types enabled.

To disable all types of logs, run:

```
eksctl utils update-cluster-logging --disable-types all
```

`ClusterConfig` Examples

In an EKS cluster, the `enableTypes` field under `clusterLogging` can take a list of possible values to enable the different types of logs for the control plane components.

The following are the possible values:

- `api`: Enables the Kubernetes API server logs.
- `audit`: Enables the Kubernetes audit logs.
- `authenticator`: Enables the authenticator logs.
- `controllerManager`: Enables the Kubernetes controller manager logs.
- `scheduler`: Enables the Kubernetes scheduler logs.

To learn more, see EKS documentation.

Disable all logs

To disable all types, use `[]` or remove the `cloudWatch` section completely.

Enable all logs

You can enable all types with `"*"` or `"all"`. For example:

```
cloudWatch:
  clusterLogging:
    enableTypes: ["*"]
```

Enable one or more logs

You can enable a subset of types by listing the types you want to enable. For example:

```
cloudWatch:
  clusterLogging:
```

```
    enableTypes:
      - "audit"
      - "authenticator"
```

Log retention period

By default, logs are stored in CloudWatch Logs, indefinitely. You can specify the number of days for which the control plane logs should be retained in CloudWatch Logs. The following example retains logs for 7 days:

```
cloudWatch:
  clusterLogging:
    logRetentionInDays: 7
```

Complete example

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: cluster-11
  region: eu-west-2

nodeGroups:
  - name: ng-1
    instanceType: m5.large
    desiredCapacity: 1

cloudWatch:
  clusterLogging:
    enableTypes: ["audit", "authenticator"]
    logRetentionInDays: 7
```

EKS Fully-Private Cluster

eksctl supports creation of fully-private clusters that have no outbound internet access and have only private subnets. VPC endpoints are used to enable private access to AWS services.

This guide describes how to create a private cluster without outbound internet access.

Creating a fully-private cluster

The only required field to create a fully-private cluster is `privateCluster.enabled`:

```
privateCluster:
  enabled: true
```

Post cluster creation, eksctl commands that need access to the Kubernetes API server will have to be run from within the cluster's VPC, a peered VPC or using some other means like AWS Direct Connect. eksctl commands that need access to the EKS APIs will not work if they're being run from within the cluster's VPC. To fix this, create an interface endpoint for Amazon EKS to privately access the Amazon Elastic Kubernetes Service (Amazon EKS) management APIs from your Amazon Virtual Private Cloud (VPC). In a future release, eksctl will add support to create this endpoint so it does not need to be manually created. Commands that need access to the OpenID Connect provider URL will need to be run from outside of your cluster's VPC once you've enabled AWS PrivateLink for Amazon EKS.

Creating managed nodegroups will continue to work, and creating self-managed nodegroups will work as it needs access to the API server via the EKS interface endpoint if the command is run from within the cluster's VPC, a peered VPC or using some other means like AWS Direct Connect.

> **Note**
>
> VPC endpoints are charged by the hour and based on usage. More details about pricing can be found at AWS PrivateLink pricing

> **Warning**
>
> Fully-private clusters are not supported in `eu-south-1`.

Configuring private access to additional AWS services

To enable worker nodes to access AWS services privately, eksctl creates VPC endpoints for the following services:

- Interface endpoints for ECR (both `ecr.api` and `ecr.dkr`) to pull container images (AWS CNI plugin etc)

- A gateway endpoint for S3 to pull the actual image layers
- An interface endpoint for EC2 required by the `aws-cloud-provider` integration
- An interface endpoint for STS to support Fargate and IAM Roles for Services Accounts (IRSA)
- An interface endpoint for CloudWatch logging (`logs`) if CloudWatch logging is enabled

These VPC endpoints are essential for a functional private cluster, and as such, eksctl does not support configuring or disabling them. However, a cluster might need private access to other AWS services (e.g., Autoscaling required by the Cluster Autoscaler). These services can be specified in `privateCluster.additionalEndpointServices`, which instructs eksctl to create a VPC endpoint for each of them.

For example, to allow private access to Autoscaling and CloudWatch logging:

```
privateCluster:
  enabled: true
  additionalEndpointServices:
  # For Cluster Autoscaler
  - "autoscaling"
  # CloudWatch logging
  - "logs"
```

The endpoints supported in `additionalEndpointServices` are `autoscaling`, `cloudformation` and `logs`.

Skipping endpoint creations

If a VPC has already been created with the necessary AWS endpoints set up and linked to the subnets described in the EKS documentation, `eksctl` can skip creating them by providing the option `skipEndpointCreation` like this:

```
privateCluster:
  enabled: true
  skipEndpointCreation: true
```

This setting cannot be used together with `additionalEndpointServices`. It will skip all endpoint creation. Also, this setting is only recommended if the endpoint <# subnet topology is correctly set up. If subnet ids are correct, `vpce` routing is set up with prefix addresses, all the necessary EKS endpoints are created and linked to the provided VPC. `eksctl` will not alter any of these resources.

Nodegroups

Only private nodegroups (both managed and self-managed) are supported in a fully-private cluster because the cluster's VPC is created without any public subnets. The `privateNetworking` field (`nodeGroup[].privateNetworking` and `managedNodeGroup[`) must be explicitly set. It is an error to leave `privateNetworking` unset in a fully-private cluster.

```
nodeGroups:
- name: ng1
  instanceType: m5.large
  desiredCapacity: 2
  # privateNetworking must be explicitly set for a fully-private cluster
  # Rather than defaulting this field to `true`,
  # we require users to explicitly set it to make the behaviour
  # explicit and avoid confusion.
  privateNetworking: true

managedNodeGroups:
- name: m1
  instanceType: m5.large
  desiredCapacity: 2
  privateNetworking: true
```

Cluster Endpoint Access

A fully-private cluster does not support modifying `clusterEndpointAccess` during cluster creation. It is an error to set either `clusterEndpoints.publicAccess` or `clusterEndpoints.privateAccess`, as a fully-private cluster can have private access only, and allowing modification of these fields can break the cluster.

User-supplied VPC and subnets

eksctl supports creation of fully-private clusters using a pre-existing VPC and subnets. Only private subnets can be specified and it's an error to specify subnets under `vpc.subnets.public`.

eksctl creates VPC endpoints in the supplied VPC and modifies route tables for the supplied subnets. Each subnet should have an explicit route table associated with it because eksctl does not modify the main route table.

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
```

```
metadata:
  name: private-cluster
  region: us-west-2

privateCluster:
  enabled: true
  additionalEndpointServices:
  - "autoscaling"

vpc:
  subnets:
    private:
      us-west-2b:
        id: subnet-0818beec303f8419b
      us-west-2c:
        id: subnet-0d42ef09490805e2a
      us-west-2d:
        id: subnet-0da7418077077c5f9


nodeGroups:
- name: ng1
  instanceType: m5.large
  desiredCapacity: 2
  # privateNetworking must be explicitly set for a fully-private cluster
  # Rather than defaulting this field to true for a fully-private cluster, we require
 users to explicitly set it
  # to make the behaviour explicit and avoid confusion.
  privateNetworking: true

managedNodeGroups:
- name: m1
  instanceType: m5.large
  desiredCapacity: 2
  privateNetworking: true
```

Managing a fully-private cluster

For all commands to work post cluster creation, eksctl will need private access to the EKS API server endpoint, and outbound internet access (for `EKS:DescribeCluster`). Commands that do not need access to the API server will be supported if eksctl has outbound internet access.

Force-delete a fully-private cluster

Errors are likely to occur when deleting a fully-private cluster through eksctl since eksctl does not automatically have access to all of the cluster's resources. `--force` exists to solve this: it will force delete the cluster and continue when errors occur.

Limitations

A limitation of the current implementation is that eksctl initially creates the cluster with both public and private endpoint access enabled, and disables public endpoint access after all operations have completed. This is required because eksctl needs access to the Kubernetes API server to allow self-managed nodes to join the cluster and to support GitOps and Fargate. After these operations have completed, eksctl switches the cluster endpoint access to private-only. This additional update does mean that creation of a fully-private cluster will take longer than for a standard cluster. In the future, eksctl may switch to a VPC-enabled Lambda function to perform these API operations.

Outbound access via HTTP proxy servers

eksctl is able to talk to the AWS APIs via a configured HTTP(S) proxy server, however you will need to ensure you set your proxy exclusion list correctly.

Generally, you will need to ensure that requests for the VPC endpoint for your cluster are not routed via your proxies by setting an appropriate `no_proxy` environment variable including the value `.eks.amazonaws.com`.

If your proxy server performs "SSL interception" and you are using IAM Roles for Service Accounts (IRSA), you will need to ensure that you explicitly bypass SSL Man-in-the-Middle for the domain `oidc.<region>.amazonaws.com`. Failure to do so will result in eksctl obtaining the incorrect root certificate thumbprint for the OIDC provider, and the AWS VPC CNI plugin will fail to start due to being unable to obtain IAM credentials, rendering your cluster inoperative.

Further information

- EKS Private Clusters

Addons

This topic describes how to manage Amazon EKS Add-Ons for your Amazon EKS clusters using eksctl. EKS Add-Ons is a feature that lets you enable and manage Kubernetes operational software through the EKS API, simplifying the process of installing, configuring, and updating cluster add-ons.

> **Warning**
>
> eksctl now installs default addons (vpc-cni, coredns, kube-proxy) as EKS addons instead of self-managed addons. This means you should use `eksctl update addon` instead of `eksctl utils update-*` commands for clusters created with eksctl v0.184.0 and above.

You can create clusters without any default networking addons when you want to use alternative CNI plugins like Cilium and Calico.

EKS Add-ons now support receiving IAM permissions via EKS Pod Identity Associations, allowing them to connect with AWS services outside of the cluster

Creating addons

Eksctl provides more flexibility for managing cluster addons:

In your config file, you can specify the addons you want and (if required) the role or policies to attach to them:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: example-cluster
  region: us-west-2

iam:
  withOIDC: true

addons:
- name: vpc-cni
  # all below properties are optional
  version: 1.7.5
  tags:
```

```
    team: eks
  # you can specify at most one of:
  attachPolicyARNs:
  - arn:aws:iam::account:policy/AmazonEKS_CNI_Policy
  # or
  serviceAccountRoleARN: arn:aws:iam::account:role/AmazonEKSCNIAccess
  # or
  attachPolicy:
    Statement:
    - Effect: Allow
      Action:
      - ec2:AssignPrivateIpAddresses
      - ec2:AttachNetworkInterface
      - ec2:CreateNetworkInterface
      - ec2:DeleteNetworkInterface
      - ec2:DescribeInstances
      - ec2:DescribeTags
      - ec2:DescribeNetworkInterfaces
      - ec2:DescribeInstanceTypes
      - ec2:DetachNetworkInterface
      - ec2:ModifyNetworkInterfaceAttribute
      - ec2:UnassignPrivateIpAddresses
      Resource: '*'
```

You can specify at most one of `attachPolicy`, `attachPolicyARNs` and `serviceAccountRoleARN`.

If none of these are specified, the addon will be created with a role that has all recommended policies attached.

> **Note**
>
> In order to attach policies to addons your cluster must have `OIDC` enabled. If it's not enabled we ignore any policies attached.

You can then either have these addons created during the cluster creation process:

```
eksctl create cluster -f config.yaml
```

Or create the addons explicitly after cluster creation using the config file or CLI flags:

```
eksctl create addon -f config.yaml
```

```
eksctl create addon --name vpc-cni --version 1.7.5 --service-account-role-arn <role-arn>
```

```
eksctl create addon --name aws-ebs-csi-driver --namespace-config 'namespace=custom-namespace'
```

> **Tip**
>
> Use the `--namespace-config` flag to deploy addons to a custom namespace instead of the default namespace.

During addon creation, if a self-managed version of the addon already exists on the cluster, you can choose how potential `configMap` conflicts shall be resolved by setting `resolveConflicts` option via the config file, e.g.

```
addons:
- name: vpc-cni
  attachPolicyARNs:
    - arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy
  resolveConflicts: overwrite
```

For addon create, the `resolveConflicts` field supports three distinct values:

- none - EKS doesn't change the value. The create might fail.
- `overwrite` - EKS overwrites any config changes back to EKS default values.
- `preserve` - EKS doesn't change the value. The create might fail. (Similarly to none, but different from `preserve` in updating addons).

Listing enabled addons

You can see what addons are enabled in your cluster by running:

```
eksctl get addons --cluster <cluster-name>
```

or

```
eksctl get addons -f config.yaml
```

Setting the addon's version

Setting the version of the addon is optional. If the `version` field is left empty `eksctl` will resolve the default version for the addon. More information about which version is the default version for specific addons can be found in the AWS documentation about EKS. Note that the default version might not necessarily be the latest version available.

The addon version can be set to `latest`. Alternatively, the version can be set with the EKS build tag specified, such as `v1.7.5-eksbuild.1` or `v1.7.5-eksbuild.2`. It can also be set to the release version of the addon, such as `v1.7.5` or `1.7.5`, and the `eksbuild` suffix tag will be discovered and set for you.

See the section below on how to discover available addons and their versions.

Discovering addons

You can discover what addons are available to install on your cluster by running:

```
eksctl utils describe-addon-versions --cluster <cluster-name>
```

This will discover your cluster's kubernetes version and filter on that. Alternatively if you want to see what addons are available for a particular kubernetes version you can run:

```
eksctl utils describe-addon-versions --kubernetes-version <version>
```

You can also discover addons by filtering on their `type`, `owner` and/or `publisher`. For e.g., to see addons for a particular owner and type you can run:

```
eksctl utils describe-addon-versions --kubernetes-version 1.22 --types "infra-
management, policy-management" --owners "aws-marketplace"
```

The `types`, `owners` and `publishers` flags are optional and can be specified together or individually to filter the results.

Discovering the configuration schema for addons

After discovering the addon and version, you can view the customization options by fetching its JSON configuration schema.

```
eksctl utils describe-addon-configuration --name vpc-cni --version v1.12.0-eksbuild.1
```

This returns a JSON schema of the various options available for this addon.

Working with configuration values

`ConfigurationValues` can be provided in the configuration file during the creation or update of addons. Only JSON and YAML formats are supported.

For eg.,

```
addons:
- name: coredns
  configurationValues: |-
    replicaCount: 2
```

```
addons:
- name: coredns
  version: latest
  configurationValues: "{\"replicaCount\":3}"
  resolveConflicts: overwrite
```

> **Note**
>
> Bear in mind that when addon configuration values are being modified, configuration conflicts will arise.

```
Thus, we need to specify how to deal with those by setting the `resolveConflicts` field
 accordingly.
As in this scenario we want to modify these values, we'd set `resolveConflicts:
 overwrite`.
```

Additionally, the get command will now also retrieve `ConfigurationValues` for the addon. e.g.

```
eksctl get addon --cluster my-cluster --output yaml
```

```
- ConfigurationValues: '{"replicaCount":3}'
  IAMRole: ""
  Issues: null
  Name: coredns
  NewerVersion: ""
  Status: ACTIVE
  Version: v1.8.7-eksbuild.3
```

Using custom namespace

A custom namespace can be provided in the configuration file during the creation of addons. A namespace can't be updated once an addon is created.

Using config file

```
addons:
  - name: aws-ebs-csi-driver
    version: latest
    namespaceConfig:
      namespace: custom-namespace
```

Using CLI flag

Alternatively, you can specify a custom namespace using the `--namespace-config` flag:

```
eksctl create addon --cluster my-cluster --name aws-ebs-csi-driver --namespace-config
 'namespace=custom-namespace'
```

The get command will also retrieve the namespace value for the addon

```
- ConfigurationValues: ""
  IAMRole: ""
  Issues: null
  Name: aws-ebs-csi-driver
  NamespaceConfig:
    namespace: custom-namespace
  NewerVersion: ""
```

```
  PodIdentityAssociations: null
  Status: ACTIVE
  Version: v1.47.0-eksbuild.1
```

Updating addons

You can update your addons to newer versions and change what policies are attached by running:

```
eksctl update addon -f config.yaml
```

```
eksctl update addon --name vpc-cni --version 1.8.0 --service-account-role-arn <new-
role>
```

> **Note**
>
> The namespace configuration cannot be updated once an addon is created. The `--namespace-config` flag is only available during addon creation.

Similarly to addon creation, When updating an addon, you have full control over the config changes that you may have previously applied on that add-on's `configMap`. Specifically, you can preserve, or overwrite them. This optional functionality is available via the same config file field `resolveConflicts`. e.g.,

```
addons:
- name: vpc-cni
  attachPolicyARNs:
    - arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy
  resolveConflicts: preserve
```

For addon update, the `resolveConflicts` field accepts three distinct values:

- none - EKS doesn't change the value. The update might fail.
- `overwrite` - EKS overwrites any config changes back to EKS default values.
- `preserve` - EKS preserves the value. If you choose this option, we recommend that you test any field and value changes on a non-production cluster before updating the add-on on your production cluster.

Deleting addons

You can delete an addon by running:

```
eksctl delete addon --cluster <cluster-name> --name <addon-name>
```

This will delete the addon and any IAM roles associated to it.

When you delete your cluster all IAM roles associated to addons are also deleted.

Cluster creation flexibility for default networking addons

When a cluster is created, EKS automatically installs VPC CNI, CoreDNS and kube-proxy as self-managed addons. To disable this behavior in order to use other CNI plugins like Cilium and Calico, eksctl now supports creating a cluster without any default networking addons. To create such a cluster, set `addonsConfig.disableDefaultAddons`, as in:

```
addonsConfig:
  disableDefaultAddons: true
```

```
eksctl create cluster -f cluster.yaml
```

To create a cluster with only CoreDNS and kube-proxy and not VPC CNI, specify the addons explicitly in `addons` and set `addonsConfig.disableDefaultAddons`, as in:

```
addonsConfig:
  disableDefaultAddons: true
addons:
  - name: kube-proxy
  - name: coredns
```

```
eksctl create cluster -f cluster.yaml
```

As part of this change, eksctl now installs default addons as EKS addons instead of self-managed addons during cluster creation if `addonsConfig.disableDefaultAddons` is not explicitly set to true. As such, `eksctl utils update-*` commands can no longer be used for updating addons for clusters created with eksctl v0.184.0 and above:

- `eksctl utils update-aws-node`

- `eksctl utils update-coredns`
- `eksctl utils update-kube-proxy`

Instead, `eksctl update addon` should be used now.

To learn more, see Amazon EKS introduces cluster creation flexibility for networking add-ons.

Enabling Access for Amazon EMR

In order to allow EMR to perform operations on the Kubernetes API, its SLR needs to be granted the required RBAC permissions. eksctl provides a command that creates the required RBAC resources for EMR, and updates the `aws-auth` ConfigMap to bind the role with the SLR for EMR.

```
eksctl create iamidentitymapping --cluster dev --service-name emr-containers --
namespace default
```

EKS Fargate Support

AWS Fargate is a managed compute engine for Amazon ECS that can run containers. In Fargate you don't need to manage servers or clusters.

Amazon EKS can now launch pods onto AWS Fargate. This removes the need to worry about how you provision or manage infrastructure for pods and makes it easier to build and run performant, highly-available Kubernetes applications on AWS.

Creating a cluster with Fargate support

You can add a cluster with Fargate support with:

```
eksctl create cluster --fargate
[#]  eksctl version 0.11.0
[#]  using region ap-northeast-1
[#]  setting availability zones to [ap-northeast-1a ap-northeast-1d ap-northeast-1c]
[#]  subnets for ap-northeast-1a - public:192.168.0.0/19 private:192.168.96.0/19
[#]  subnets for ap-northeast-1d - public:192.168.32.0/19 private:192.168.128.0/19
[#]  subnets for ap-northeast-1c - public:192.168.64.0/19 private:192.168.160.0/19
[#]  nodegroup "ng-dba9d731" will use "ami-02e124a380df41614" [AmazonLinux2/1.14]
[#]  using Kubernetes version 1.14
```

```
[#]  creating EKS cluster "ridiculous-painting-1574859263" in "ap-northeast-1" region
[#]  will create 2 separate CloudFormation stacks for cluster itself and the initial
 nodegroup
[#]  if you encounter any issues, check CloudFormation console or try 'eksctl utils
 describe-stacks --region=ap-northeast-1 --cluster=ridiculous-painting-1574859263'
[#]  CloudWatch logging will not be enabled for cluster "ridiculous-
painting-1574859263" in "ap-northeast-1"
[#]  you can enable it with 'eksctl utils update-cluster-logging --enable-
types={SPECIFY-YOUR-LOG-TYPES-HERE (e.g. all)} --region=ap-northeast-1 --
cluster=ridiculous-painting-1574859263'
[#]  Kubernetes API endpoint access will use default of {publicAccess=true,
 privateAccess=false} for cluster "ridiculous-painting-1574859263" in "ap-northeast-1"
[#]  2 sequential tasks: { create cluster control plane "ridiculous-
painting-1574859263", create nodegroup "ng-dba9d731" }
[#]  building cluster stack "eksctl-ridiculous-painting-1574859263-cluster"
[#]  deploying stack "eksctl-ridiculous-painting-1574859263-cluster"
[#]  building nodegroup stack "eksctl-ridiculous-painting-1574859263-nodegroup-ng-
dba9d731"
[#]  --nodes-min=2 was set automatically for nodegroup ng-dba9d731
[#]  --nodes-max=2 was set automatically for nodegroup ng-dba9d731
[#]  deploying stack "eksctl-ridiculous-painting-1574859263-nodegroup-ng-dba9d731"
[#]  all EKS cluster resources for "ridiculous-painting-1574859263" have been created
[#]  saved kubeconfig as "/Users/marc/.kube/config"
[#]  adding identity "arn:aws:iam::123456789012:role/eksctl-ridiculous-painting-157485-
NodeInstanceRole-104DXUJOFDPO5" to auth ConfigMap
[#]  nodegroup "ng-dba9d731" has 0 node(s)
[#]  waiting for at least 2 node(s) to become ready in "ng-dba9d731"
[#]  nodegroup "ng-dba9d731" has 2 node(s)
[#]  node "ip-192-168-27-156.ap-northeast-1.compute.internal" is ready
[#]  node "ip-192-168-95-177.ap-northeast-1.compute.internal" is ready
[#]  creating Fargate profile "default" on EKS cluster "ridiculous-painting-1574859263"
[#]  created Fargate profile "default" on EKS cluster "ridiculous-painting-1574859263"
[#]  kubectl command should work with "/Users/marc/.kube/config", try 'kubectl get
 nodes'
[#]  EKS cluster "ridiculous-painting-1574859263" in "ap-northeast-1" region is ready
```

This command will have created a cluster and a Fargate profile. This profile contains certain information needed by AWS to instantiate pods in Fargate. These are:

- pod execution role to define the permissions required to run the pod and the networking location (subnet) to run the pod. This allows the same networking and security permissions to be applied to multiple Fargate pods and makes it easier to migrate existing pods on a cluster to Fargate.

- Selector to define which pods should run on Fargate. This is composed by a `namespace` and `labels`.

When the profile is not specified but support for Fargate is enabled with `--fargate` a default Fargate profile is created. This profile targets the `default` and the `kube-system` namespaces so pods in those namespaces will run on Fargate.

The Fargate profile that was created can be checked with the following command:

```
eksctl get fargateprofile --cluster ridiculous-painting-1574859263 -o yaml
- name: fp-default
  podExecutionRoleARN: arn:aws:iam::123456789012:role/eksctl-ridiculous-
painting-1574859263-ServiceRole-EIFQOH0S1GE7
  selectors:
  - namespace: default
  - namespace: kube-system
  subnets:
  - subnet-0b3a5522f3b48a742
  - subnet-0c35f1497067363f3
  - subnet-0a29aa00b25082021
```

To learn more about selectors see Designing Fargate profiles.

Creating a cluster with Fargate support using a config file

The following config file declares an EKS cluster with both a nodegroup composed of one EC2 `m5.large` instance and two Fargate profiles. All pods defined in the `default` and `kube-system` namespaces will run on Fargate. All pods in the `dev` namespace that also have the label `dev=passed` will also run on Fargate. Any other pods will be scheduled on the node in `ng-1`.

```
# An example of ClusterConfig with a normal nodegroup and a Fargate profile.
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: fargate-cluster
  region: ap-northeast-1

nodeGroups:
  - name: ng-1
```

```
    instanceType: m5.large
    desiredCapacity: 1

fargateProfiles:
  - name: fp-default
    selectors:
      # All workloads in the "default" Kubernetes namespace will be
      # scheduled onto Fargate:
      - namespace: default
      # All workloads in the "kube-system" Kubernetes namespace will be
      # scheduled onto Fargate:
      - namespace: kube-system
  - name: fp-dev
    selectors:
      # All workloads in the "dev" Kubernetes namespace matching the following
      # label selectors will be scheduled onto Fargate:
      - namespace: dev
        labels:
          env: dev
          checks: passed
```

```
eksctl create cluster -f cluster-fargate.yaml
[#]  eksctl version 0.11.0
[#]  using region ap-northeast-1
[#]  setting availability zones to [ap-northeast-1c ap-northeast-1a ap-northeast-1d]
[#]  subnets for ap-northeast-1c - public:192.168.0.0/19 private:192.168.96.0/19
[#]  subnets for ap-northeast-1a - public:192.168.32.0/19 private:192.168.128.0/19
[#]  subnets for ap-northeast-1d - public:192.168.64.0/19 private:192.168.160.0/19
[#]  nodegroup "ng-1" will use "ami-02e124a380df41614" [AmazonLinux2/1.14]
[#]  using Kubernetes version 1.14
[#]  creating EKS cluster "fargate-cluster" in "ap-northeast-1" region with Fargate
 profile and un-managed nodes
[#]  1 nodegroup (ng-1) was included (based on the include/exclude rules)
[#]  will create a CloudFormation stack for cluster itself and 1 nodegroup stack(s)
[#]  will create a CloudFormation stack for cluster itself and 0 managed nodegroup
 stack(s)
[#]  if you encounter any issues, check CloudFormation console or try 'eksctl utils
 describe-stacks --region=ap-northeast-1 --cluster=fargate-cluster'
[#]  CloudWatch logging will not be enabled for cluster "fargate-cluster" in "ap-
northeast-1"
[#]  you can enable it with 'eksctl utils update-cluster-logging --enable-
types={SPECIFY-YOUR-LOG-TYPES-HERE (e.g. all)} --region=ap-northeast-1 --
cluster=fargate-cluster'
```

```
[#]  Kubernetes API endpoint access will use default of {publicAccess=true,
 privateAccess=false} for cluster "fargate-cluster" in "ap-northeast-1"
[#]  2 sequential tasks: { create cluster control plane "fargate-cluster", create
 nodegroup "ng-1" }
[#]  building cluster stack "eksctl-fargate-cluster-cluster"
[#]  deploying stack "eksctl-fargate-cluster-cluster"
[#]  building nodegroup stack "eksctl-fargate-cluster-nodegroup-ng-1"
[#]  --nodes-min=1 was set automatically for nodegroup ng-1
[#]  --nodes-max=1 was set automatically for nodegroup ng-1
[#]  deploying stack "eksctl-fargate-cluster-nodegroup-ng-1"
[#]  all EKS cluster resources for "fargate-cluster" have been created
[#]  saved kubeconfig as "/home/user1/.kube/config"
[#]  adding identity "arn:aws:iam::123456789012:role/eksctl-fargate-cluster-nod-
NodeInstanceRole-42Q80B2Z147I" to auth ConfigMap
[#]  nodegroup "ng-1" has 0 node(s)
[#]  waiting for at least 1 node(s) to become ready in "ng-1"
[#]  nodegroup "ng-1" has 1 node(s)
[#]  node "ip-192-168-71-83.ap-northeast-1.compute.internal" is ready
[#]  creating Fargate profile "fp-default" on EKS cluster "fargate-cluster"
[#]  created Fargate profile "fp-default" on EKS cluster "fargate-cluster"
[#]  creating Fargate profile "fp-dev" on EKS cluster "fargate-cluster"
[#]  created Fargate profile "fp-dev" on EKS cluster "fargate-cluster"
[#]  "coredns" is now schedulable onto Fargate
[#]  "coredns" is now scheduled onto Fargate
[#]  "coredns" is now scheduled onto Fargate
[#]  "coredns" pods are now scheduled onto Fargate
[#]  kubectl command should work with "/home/user1/.kube/config", try 'kubectl get
 nodes'
[#]  EKS cluster "fargate-cluster" in "ap-northeast-1" region is ready
```

Designing Fargate profiles

Each selector entry has up to two components, namespace and a list of key-value pairs. Only the namespace component is required to create a selector entry. All rules (namespaces, key value pairs) must apply to a pod to match a selector entry. A pod only needs to match one selector entry to run on the profile. Any pod that matches all the conditions in a selector field would be scheduled to be run on Fargate. Any pods not matching either the whitelisted Namespaces but where the user manually set the scheduler: fargate-scheduler field would be stuck in a Pending state, as they were not authorized to run on Fargate.

Profiles must meet the following requirements:

- One selector is mandatory per profile
- Each selector must include a namespace; labels are optional

Example: scheduling workload in Fargate

To schedule pods on Fargate for the example mentioned above, one could, for example, create a namespace called `dev` and deploy the workload there:

```
kubectl create namespace dev
namespace/dev created

kubectl run nginx --image=nginx --restart=Never --namespace dev
pod/nginx created

kubectl get pods --all-namespaces --output wide
NAMESPACE     NAME                       READY   STATUS    AGE   IP                NODE
dev           nginx                      1/1     Running   75s   192.168.183.140
 fargate-ip-192-168-183-140.ap-northeast-1.compute.internal
kube-system   aws-node-44qst             1/1     Running   21m   192.168.70.246
 ip-192-168-70-246.ap-northeast-1.compute.internal
kube-system   aws-node-4vr66             1/1     Running   21m   192.168.23.122
 ip-192-168-23-122.ap-northeast-1.compute.internal
kube-system   coredns-699bb99bf8-84x74   1/1     Running   26m   192.168.2.95
 ip-192-168-23-122.ap-northeast-1.compute.internal
kube-system   coredns-699bb99bf8-f6x6n   1/1     Running   26m   192.168.90.73
 ip-192-168-70-246.ap-northeast-1.compute.internal
kube-system   kube-proxy-brxhg           1/1     Running   21m   192.168.23.122
 ip-192-168-23-122.ap-northeast-1.compute.internal
kube-system   kube-proxy-zd7s8           1/1     Running   21m   192.168.70.246
 ip-192-168-70-246.ap-northeast-1.compute.internal
```

From the output of the last `kubectl get pods` command we can see that the `nginx` pod is deployed in a node called `fargate-ip-192-168-183-140.ap-northeast-1.compute.internal`.

Managing Fargate profiles

To deploy Kubernetes workloads on Fargate, EKS needs a Fargate profile. When creating a cluster like in the examples above, `eksctl` takes care of this by creating a default profile. Given an already existing cluster, it's also possible to create a Fargate profile with the `eksctl create fargateprofile` command:

Note

This operation is only supported on clusters that run on the EKS platform version `eks.5` or higher.

Note

If the existing cluster was created with a version of `eksctl` prior to 0.11.0, you will need to run `eksctl upgrade cluster` before creating the Fargate profile.

```
eksctl create fargateprofile --namespace dev --cluster fargate-example-cluster
[#]  creating Fargate profile "fp-9bfc77ad" on EKS cluster "fargate-example-cluster"
[#]  created Fargate profile "fp-9bfc77ad" on EKS cluster "fargate-example-cluster"
```

You can also specify the name of the Fargate profile to be created. This name must not start with the prefix `eks-`.

```
eksctl create fargateprofile --namespace dev --cluster fargate-example-cluster --name
 fp-development
[#]  created Fargate profile "fp-development" on EKS cluster "fargate-example-cluster"
```

Using this command with CLI flags eksctl can only create a single Fargate profile with a simple selector. For more complex selectors, for example with more namespaces, eksctl supports using a config file:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: fargate-example-cluster
  region: ap-northeast-1

fargateProfiles:
  - name: fp-default
    selectors:
      # All workloads in the "default" Kubernetes namespace will be
      # scheduled onto Fargate:
```

```
      - namespace: default
      # All workloads in the "kube-system" Kubernetes namespace will be
      # scheduled onto Fargate:
      - namespace: kube-system
  - name: fp-dev
    selectors:
      # All workloads in the "dev" Kubernetes namespace matching the following
      # label selectors will be scheduled onto Fargate:
      - namespace: dev
        labels:
          env: dev
          checks: passed
```

```
eksctl create fargateprofile -f fargate-example-cluster.yaml
[#]  creating Fargate profile "fp-default" on EKS cluster "fargate-example-cluster"
[#]  created Fargate profile "fp-default" on EKS cluster "fargate-example-cluster"
[#]  creating Fargate profile "fp-dev" on EKS cluster "fargate-example-cluster"
[#]  created Fargate profile "fp-dev" on EKS cluster "fargate-example-cluster"
[#]  "coredns" is now scheduled onto Fargate
[#]  "coredns" pods are now scheduled onto Fargate
```

To see existing Fargate profiles in a cluster:

```
eksctl get fargateprofile --cluster fargate-example-cluster
NAME         SELECTOR_NAMESPACE  SELECTOR_LABELS  POD_EXECUTION_ROLE_ARN
                                                    SUBNETS
fp-9bfc77ad  dev                 <none>           arn:aws:iam::123456789012:role/
eksctl-fargate-example-cluster-ServiceRole-1T5F78E5FSH79
 subnet-00adf1d8c99f83381,subnet-04affb163ffab17d4,subnet-035b34379d5ef5473
```

And to see them in `yaml` format:

```
eksctl get fargateprofile --cluster fargate-example-cluster -o yaml
- name: fp-9bfc77ad
  podExecutionRoleARN: arn:aws:iam::123456789012:role/eksctl-fargate-example-cluster-
ServiceRole-1T5F78E5FSH79
  selectors:
  - namespace: dev
  subnets:
  - subnet-00adf1d8c99f83381
  - subnet-04affb163ffab17d4
  - subnet-035b34379d5ef5473
```

Or in `json` format:

```
eksctl get fargateprofile --cluster fargate-example-cluster -o json
[
    {
        "name": "fp-9bfc77ad",
        "podExecutionRoleARN": "arn:aws:iam::123456789012:role/eksctl-fargate-example-
cluster-ServiceRole-1T5F78E5FSH79",
        "selectors": [
            {
                "namespace": "dev"
            }
        ],
        "subnets": [
            "subnet-00adf1d8c99f83381",
            "subnet-04affb163ffab17d4",
            "subnet-035b34379d5ef5473"
        ]
    }
]
```

Fargate profiles are immutable by design. To change something, create a new Fargate profile with the desired changes and delete the old one with the `eksctl delete fargateprofile` command like in the following example:

```
eksctl delete fargateprofile --cluster fargate-example-cluster --name fp-9bfc77ad --
wait
2019-11-27T19:04:26+09:00 [#]  deleting Fargate profile "fp-9bfc77ad"
  ClusterName: "fargate-example-cluster",
  FargateProfileName: "fp-9bfc77ad"
}
```

Note that the profile deletion is a process that can take up to a few minutes. When the `--wait` flag is not specified, `eksctl` optimistically expects the profile to be deleted and returns as soon as the AWS API request has been sent. To make `eksctl` wait until the profile has been successfully deleted, use `--wait` like in the example above.

Further reading

- AWS Fargate
- Amazon EKS can now launch pods onto AWS Fargate

Cluster upgrades

An *`eksctl`-managed* cluster can be upgraded in 3 easy steps:

1. upgrade control plane version with `eksctl upgrade cluster`
2. upgrade nodegroups
3. update the default networking add-ons (For more information, see the section called "Default add-on updates"):

Carefully review cluster upgrade related resources:

- Update existing cluster to new Kubernetes version in the Amazon EKS User Guide
- Best Practices for Cluster Upgrades in the EKS Best Practices Guide

> **Note**
>
> The old `eksctl update cluster` will be deprecated. Use `eksctl upgrade cluster` instead.

Updating control plane version

Control plane version upgrades must be done for one minor version at a time.

To upgrade control plane to the next available version run:

```
eksctl upgrade cluster --name=<clusterName>
```

This command will not apply any changes right away, you will need to re-run it with `--approve` to apply the changes.

The target version for the cluster upgrade can be specified both with the CLI flag:

```
eksctl upgrade cluster --name=<clusterName> --version=1.16
```

or with the config file

```
cat cluster1.yaml
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: cluster-1
  region: eu-north-1
  version: "1.16"

eksctl upgrade cluster --config-file cluster1.yaml
```

Warning

The only values allowed for the `--version` and `metadata.version` arguments are the current version of the cluster or one version higher. Upgrades of more than one Kubernetes version are not supported.

Default add-on updates

This topic explains how to update the default pre-installed add-ons that are included on EKS clusters.

Warning

eksctl now installs default addons as EKS addons instead of self-managed addons. Read more about its implications in Cluster creation flexibility for default networking addons. For updating addons, `eksctl utils update-<addon>` cannot be used for clusters created with eksctl v0.184.0 and above. This guide is only valid for clusters created before this change.

There are 3 default add-ons that get included in each EKS cluster:

- `kube-proxy`
- `aws-node`
- `coredns`

Update pre-installed add-on

For official EKS addons that are created manually through `eksctl create addons` or upon cluster creation, the way to manage them is through `eksctl create/get/update/delete addon`. In such cases, please refer to the docs about EKS Add-Ons.

The process for updating each of them is different, hence there are 3 distinct commands that you will need to run. All of the following commands accept `--config-file`. By default each of these commands runs in plan mode, if you are happy with the proposed changes, re-run with `--approve`.

To update `kube-proxy`, run:

```
eksctl utils update-kube-proxy --cluster=<clusterName>
```

To update `aws-node`, run:

```
eksctl utils update-aws-node --cluster=<clusterName>
```

To update `coredns`, run:

```
eksctl utils update-coredns --cluster=<clusterName>
```

Once upgraded, be sure to run `kubectl get pods -n kube-system` and check if all addon pods are in ready state, you should see something like this:

```
NAME                       READY   STATUS    RESTARTS   AGE
aws-node-g5ghn             1/1     Running   0          2m
aws-node-zfc9s             1/1     Running   0          2m
coredns-7bcbfc4774-g6gg8   1/1     Running   0          1m
coredns-7bcbfc4774-hftng   1/1     Running   0          1m
kube-proxy-djkp7           1/1     Running   0          3m
kube-proxy-mpdsp           1/1     Running   0          3m
```

Support for Zonal Shift in EKS clusters

EKS now supports Amazon Application Recovery Controller (ARC) zonal shift and zonal autoshift that enhances the resiliency of multi-AZ cluster environments. With AWS Zonal Shift, customers

can shift in-cluster traffic away from an impaired availability zone, ensuring new Kubernetes pods and nodes are launched in healthy availability zones only.

Creating a cluster with zonal shift enabled

```
# zonal-shift-cluster.yaml
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: highly-available-cluster
  region: us-west-2


zonalShiftConfig:
  enabled: true
```

```
eksctl create cluster -f zonal-shift-cluster.yaml
```

Enabling zonal shift on an existing cluster

To enable or disable zonal shift on an existing cluster, run

```
eksctl utils update-zonal-shift-config -f zonal-shift-cluster.yaml
```

or without a config file:

```
eksctl utils update-zonal-shift-config --cluster=zonal-shift-cluster --enabled
```

Further information

- EKS Zonal Shift

Karpenter Support

`eksctl` provides support for adding Karpenter to a newly created cluster. It will create all the necessary prerequisites outlined in Karpenter's Getting Started section including installing

Karpenter itself using Helm. We currently support installing versions `0.28.0+`. See the Karpenter compatibility section for further details.

The following cluster configuration outlines a typical Karpenter installation:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: cluster-with-karpenter
  region: us-west-2
  version: '1.32' # requires a version of Kubernetes compatible with Karpenter
  tags:
    karpenter.sh/discovery: cluster-with-karpenter # here, it is set to the cluster
 name
iam:
  withOIDC: true # required

karpenter:
  version: '1.2.1' # Exact version should be specified according to the Karpenter
 compatibility matrix

managedNodeGroups:
  - name: managed-ng-1
    minSize: 1
    maxSize: 2
    desiredCapacity: 1
```

The version is Karpenter's version as it can be found in their Helm Repository. The following options are also available to be set:

```
karpenter:
  version: '1.2.1'
  createServiceAccount: true # default is false
  defaultInstanceProfile: 'KarpenterNodeInstanceProfile' # default is to use the IAM
 instance profile created by eksctl
  withSpotInterruptionQueue: true # adds all required policies and rules for supporting
 Spot Interruption Queue, default is false
```

OIDC must be defined in order to install Karpenter.

Once Karpenter is successfully installed, add NodePool(s) and NodeClass(es) to allow Karpenter to start adding nodes to the cluster.

The NodePool's `nodeClassRef` section must match the name of an `EC2NodeClass`. For example:

```
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: example
  annotations:
    kubernetes.io/description: "Example NodePool"
spec:
  template:
    spec:
      requirements:
        - key: kubernetes.io/arch
          operator: In
          values: ["amd64"]
        - key: kubernetes.io/os
          operator: In
          values: ["linux"]
        - key: karpenter.sh/capacity-type
          operator: In
          values: ["on-demand"]
        - key: karpenter.k8s.aws/instance-category
          operator: In
          values: ["c", "m", "r"]
        - key: karpenter.k8s.aws/instance-generation
          operator: Gt
          values: ["2"]
      nodeClassRef:
        group: karpenter.k8s.aws
        kind: EC2NodeClass
        name: example # must match the name of an EC2NodeClass
```

```
apiVersion: karpenter.k8s.aws/v1
kind: EC2NodeClass
metadata:
  name: example
  annotations:
    kubernetes.io/description: "Example EC2NodeClass"
spec:
  role: "eksctl-KarpenterNodeRole-${CLUSTER_NAME}" # replace with your cluster name
```

```
  subnetSelectorTerms:
    - tags:
        karpenter.sh/discovery: "${CLUSTER_NAME}" # replace with your cluster name
  securityGroupSelectorTerms:
    - tags:
        karpenter.sh/discovery: "${CLUSTER_NAME}" # replace with your cluster name
  amiSelectorTerms:
    - alias: al2023@latest # Amazon Linux 2023
```

Note that you must specify one of `role` or `instanceProfile` for launch nodes. If you choose to use `instanceProfile` the name of the profile created by `eksctl` follows the pattern: `eksctl-KarpenterNodeInstanceProfile-<cluster-name>`.

Automatic Security Group Tagging

`eksctl` automatically tags the cluster's shared node security group with `karpenter.sh/discovery` when both Karpenter is enabled (`karpenter.version` specified) and the `karpenter.sh/discovery` tag exists in `metadata.tags`. This enables AWS Load Balancer Controller compatibility.

Note with karpenter 0.32.0+, Provisioners have been deprecated and replaced by NodePool.

Cluster Config Schema

> **Note**
>
> The location of the schema is currently being migrated.

You can use a yaml file to create a cluster. View the schema reference.

For example:

```
eksctl create cluster -f cluster.yaml
```

The schema reference for this file is available on GitHub.

For more information about using the file, see the section called “Creating and managing clusters”.

Nodegroups

This chapter includes information about how you create and configure Nodegroups with Eksctl. Nodegroups are groups of EC2 instances attached to an EKS cluster.

Topics:

- the section called “Spot instances”
  - Create and manage EKS clusters with Spot instances using managed node groups
  - Configure Spot instances for unmanaged node groups using the MixedInstancesPolicy
  - Distinguish Spot and On-Demand instances using the `node-lifecycle` Kubernetes label
- the section called “Auto Scaling”
  - Enable automatic scaling of Kubernetes cluster nodes by creating a cluster or nodegroup with IAM role that allows the use of the cluster autoscaler
  - Configure nodegroup definitions to include necessary tags and annotations for the cluster autoscaler to scale the nodegroup
  - Create separate nodegroups for each availability zone if workloads have zone-specific requirements, such as zone-specific storage or affinity rules
- the section called “EKS managed nodegroups”
  - Provision and manage EC2 instances (nodes) for Amazon EKS Kubernetes clusters
  - Easily apply bug fixes, security patches, and update nodes to the latest Kubernetes versions
- the section called “EKS Hybrid Nodes”
  - Enable running on-premises and edge applications on customer-managed infrastructure with the same Amazon EKS clusters, features, and tools used in AWS
  - Configure networking to connect on-premises networks to a VPC, using options like AWS Site-to-Site VPN or AWS Direct Connect
  - Set up credentials for remote nodes to authenticate with the EKS cluster, using either AWS Systems Manager (SSM) or AWS IAM Roles Anywhere
- the section called “Node Repair Config”
  - Enabling Node Repair for EKS Managed Nodegroups to automatically monitor and replace or reboot unhealthy worker nodes
- the section called “ARM Support”

- Create an EKS cluster with ARM-based Graviton instances for improved performance and cost-efficiency

- the section called “Taints”
  - Apply taints to specific node groups in a Kubernetes cluster
  - Control scheduling and eviction of pods based on taint keys, values, and effects
- the section called “Launch template support”
  - Launching managed node groups using a provided EC2 Launch Template
  - Upgrading a managed node group to use a different version of a Launch Template
  - Understanding limitations and considerations when using custom AMIs and Launch Templates with managed node groups
- the section called “Work with node groups”
  - Enable SSH access to EC2 instances in the node group
  - Scale the number of nodes in a node group up or down
- the section called “Custom subnets”
  - Extend an existing VPC with a new subnet and add a Nodegroup to that subnet
- the section called “Node bootstrapping”
  - Understand the new node initialization process (nodeadm) introduced in AmazonLinux2023
  - Learn about the default NodeConfig settings applied by eksctl for self-managed and EKS-managed nodes
  - Customize the node bootstrapping process by providing an overrideBootstrapCommand with a custom NodeConfig
- the section called “Unmanaged nodegroups”
  - Create or update unmanaged node groups in an EKS cluster
  - Update default Kubernetes add-ons like kube-proxy, aws-node, and CoreDNS
- the section called “GPU Support”
  - Eksctl supports selecting GPU instance types for nodegroups, enabling the use of GPU-accelerated workloads on EKS clusters.
  - Eksctl automatically installs the NVIDIA Kubernetes device plugin when a GPU-enabled instance type is selected, facilitating the use of GPU resources in the cluster.
  - Users can disable automatic plugin installation and manually install a specific version of the NVIDIA Kubernetes device plugin using the provided commands.

- the section called “Instance Selector”
  - Automatically generate a list of suitable EC2 instance types based on resource criteria like vCPUs, memory, GPUs, and CPU architecture
  - Create clusters and node groups with the instance types matched by the specified instance selector criteria
  - Perform a dry run to inspect and modify the instance types matched by the instance selector before creating a node group
- the section called “Additional Volume Mappings”
  - Configure additional volume mappings for a managed node group in an EKS cluster
  - Customize volume properties like size, type, encryption, IOPS, and throughput for the additional volumes
  - Attach existing EBS snapshots as additional volumes to the node group
- the section called “Windows Worker Nodes”
  - Add Windows node groups to an existing Linux Kubernetes cluster to enable running Windows workloads
  - Schedule workloads on the appropriate operating system (Windows or Linux) using node selectors based on the `kubernetes.io/os` and `kubernetes.io/arch` labels
- the section called “Custom AMI support”
  - Use the `--node-ami` flag to specify a custom AMI for node groups, query AWS for the latest EKS-optimized AMI, or use AWS Systems Manager Parameter Store to find the AMI.
  - Set the `--node-ami-family` flag to specify the operating system family for the node group AMI, such as AmazonLinux2, Ubuntu2204, or WindowsServer2022CoreContainer.
  - For Windows node groups, specify a custom AMI and provide a PowerShell bootstrap script via the `overrideBootstrapCommand`.
- the section called “Custom DNS”
  - Overwrite the DNS server IP address used for internal and external DNS lookups

Work with node groups

Creating nodegroups

You can add one or more nodegroups in addition to the initial nodegroup created along with the cluster.

To create an additional nodegroup, use:

```
eksctl create nodegroup --cluster=<clusterName> [--name=<nodegroupName>]
```

> **Note**
>
> `--version` flag is not supported for managed nodegroups. It always inherits the version from control plane.

By default, new unmanaged nodegroups inherit the version from the control plane (`--version=auto`), but you can specify a different version, you can also use `--version=latest` to force use of whichever is the latest version.

Additionally, you can use the same config file used for `eksctl create cluster`:

```
eksctl create nodegroup --config-file=<path>
```

Creating a nodegroup from a config file

Nodegroups can also be created through a cluster definition or config file. Given the following example config file and an existing cluster called `dev-cluster`:

```
# dev-cluster.yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: dev-cluster
  region: eu-north-1

managedNodeGroups:
  - name: ng-1-workers
    labels: { role: workers }
    instanceType: m5.xlarge
    desiredCapacity: 10
    volumeSize: 80
    privateNetworking: true
  - name: ng-2-builders
    labels: { role: builders }
```

```
    instanceType: m5.2xlarge
    desiredCapacity: 2
    volumeSize: 100
    privateNetworking: true
```

The nodegroups `ng-1-workers` and `ng-2-builders` can be created with this command:

```
eksctl create nodegroup --config-file=dev-cluster.yaml
```

Load Balancing

If you have already prepared for attaching existing classic load balancers or/and target groups to the nodegroups, you can specify these in the config file. The classic load balancers or/and target groups are automatically associated with the ASG when creating nodegroups. This is only supported for self-managed nodegroups defined via the `nodeGroups` field.

```
# dev-cluster-with-lb.yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: dev-cluster
  region: eu-north-1

nodeGroups:
  - name: ng-1-web
    labels: { role: web }
    instanceType: m5.xlarge
    desiredCapacity: 10
    privateNetworking: true
    classicLoadBalancerNames:
      - dev-clb-1
      - dev-clb-2
    asgMetricsCollection:
      - granularity: 1Minute
        metrics:
          - GroupMinSize
          - GroupMaxSize
          - GroupDesiredCapacity
          - GroupInServiceInstances
          - GroupPendingInstances
          - GroupStandbyInstances
```

```
          - GroupTerminatingInstances
          - GroupTotalInstances
  - name: ng-2-api
    labels: { role: api }
    instanceType: m5.2xlarge
    desiredCapacity: 2
    privateNetworking: true
    targetGroupARNs:
      - arn:aws:elasticloadbalancing:eu-north-1:01234567890:targetgroup/dev-target-
group-1/abcdef0123456789
```

Nodegroup selection in config files

To perform a `create` or `delete` operation on only a subset of the nodegroups specified in a config file, there are two CLI flags that accept a list of globs, `0` and `1`, e.g.:

```
eksctl create nodegroup --config-file=<path> --include='ng-prod-*-??' --exclude='ng-
test-1-ml-a,ng-test-2-?'
```

Using the example config file above, one can create all the workers nodegroup except the workers one with the following command:

```
eksctl create nodegroup --config-file=dev-cluster.yaml --exclude=ng-1-workers
```

Or one could delete the builders nodegroup with:

```
eksctl delete nodegroup --config-file=dev-cluster.yaml --include=ng-2-builders --
approve
```

In this case, we also need to supply the `--approve` command to actually delete the nodegroup.

Include and exclude rules

- if no `--include` or `--exclude` is specified everything is included
- if only `--include` is specified, only nodegroups that match those globs will be included
- if only `--exclude` is specified, all nodegroups that do not match those globs are included
- if both are specified then `--exclude` rules take precedence over `--include` (i.e. nodegroups that match rules in both groups will be excluded)

Listing nodegroups

To list the details about a nodegroup or all of the nodegroups, use:

```
eksctl get nodegroup --cluster=<clusterName> [--name=<nodegroupName>]
```

To list one or more nodegroups in YAML or JSON format, which outputs more info than the default log table, use:

```
# YAML format
eksctl get nodegroup --cluster=<clusterName> [--name=<nodegroupName>] --output=yaml

# JSON format
eksctl get nodegroup --cluster=<clusterName> [--name=<nodegroupName>] --output=json
```

Nodegroup immutability

By design, nodegroups are immutable. This means that if you need to change something (other than scaling) like the AMI or the instance type of a nodegroup, you would need to create a new nodegroup with the desired changes, move the load and delete the old one. See the Deleting and draining nodegroups section.

Scaling nodegroups

Nodegroup scaling is a process that can take up to a few minutes. When the `--wait` flag is not specified, `eksctl` optimistically expects the nodegroup to be scaled and returns as soon as the AWS API request has been sent. To make `eksctl` wait until the nodes are available, add a `--wait` flag like the example below.

> **Note**
>
> Scaling a nodegroup down/in (i.e. reducing the number of nodes) may result in errors as we rely purely on changes to the ASG. This means that the node(s) being removed/terminated aren't explicitly drained. This may be an area for improvement in the future.

Scaling a managed nodegroup is achieved by directly calling the EKS API that updates a managed node group configuration.

Scaling a single nodegroup

A nodegroup can be scaled by using the `eksctl scale nodegroup` command:

```
eksctl scale nodegroup --cluster=<clusterName> --nodes=<desiredCount> --
name=<nodegroupName> [ --nodes-min=<minSize> ] [ --nodes-max=<maxSize> ] --wait
```

For example, to scale nodegroup `ng-a345f4e1` in `cluster-1` to 5 nodes, run:

```
eksctl scale nodegroup --cluster=cluster-1 --nodes=5 ng-a345f4e1
```

A nodegroup can also be scaled by using a config file passed to `--config-file` and specifying the name of the nodegroup that should be scaled with `--name`. Eksctl will search the config file and discover that nodegroup as well as its scaling configuration values.

If the desired number of nodes is `NOT` within the range of current minimum and current maximum number nodes, one specific error will be shown. These values can also be passed with flags `--nodes-min` and `--nodes-max` respectively.

Scaling multiple nodegroups

Eksctl can discover and scale all the nodegroups found in a config file that is passed with `--config-file`.

Similarly to scaling a single nodegroup, the same set of validations apply to each nodegroup. For example, the desired number of nodes must be within the range of the minimum and maximum number of nodes.

Deleting and draining nodegroups

To delete a nodegroup, run:

```
eksctl delete nodegroup --cluster=<clusterName> --name=<nodegroupName>
```

Include and exclude rules can also be used with this command.

> **Note**
>
> This will drain all pods from that nodegroup before the instances are deleted.

To skip eviction rules during the drain process, run:

```
eksctl delete nodegroup --cluster=<clusterName> --name=<nodegroupName> --disable-
eviction
```

All nodes are cordoned and all pods are evicted from a nodegroup on deletion, but if you need to drain a nodegroup without deleting it, run:

```
eksctl drain nodegroup --cluster=<clusterName> --name=<nodegroupName>
```

To uncordon a nodegroup, run:

```
eksctl drain nodegroup --cluster=<clusterName> --name=<nodegroupName> --undo
```

To ignore eviction rules such as PodDisruptionBudget settings, run:

```
eksctl drain nodegroup --cluster=<clusterName> --name=<nodegroupName> --disable-
eviction
```

To speed up the drain process you can specify `--parallel <value>` for the number of nodes to drain in parallel.

Other features

You can also enable SSH, ASG access and other features for a nodegroup, e.g.:

```
eksctl create nodegroup --cluster=cluster-1 --node-
labels="autoscaling=enabled,purpose=ci-worker" --asg-access --full-ecr-access --ssh-
access
```

Update labels

There are no specific commands in `eksctl` to update the labels of a nodegroup, but it can easily be achieved using `kubectl`, e.g.:

```
kubectl label nodes -l alpha.eksctl.io/nodegroup-name=ng-1 new-label=foo
```

SSH Access

You can enable SSH access for nodegroups by configuring one of `publicKey`, `publicKeyName` and `publicKeyPath` in your nodegroup configuration. Alternatively you can use AWS Systems Manager (SSM) to SSH onto nodes, by configuring the nodegroup with `enableSsm`:

```
managedNodeGroups:
  - name: ng-1
    instanceType: m5.large
    desiredCapacity: 1
    ssh: # import public key from file
      publicKeyPath: ~/.ssh/id_rsa_tests.pub
  - name: ng-2
    instanceType: m5.large
    desiredCapacity: 1
    ssh: # use existing EC2 key
      publicKeyName: ec2_dev_key
  - name: ng-3
    instanceType: m5.large
    desiredCapacity: 1
    ssh: # import inline public key
      publicKey: "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDqZEdzvHnK/GVP8nLngRHu/
GDi/3PeES7+Bx6l3koXn/Oi/UmM9/jcW5XGziZ/
oe1cPJ777eZV7muEvXg5ZMQBrYxUtYCdvd8Rt6DIoSqDLsIPqbuuNlQoBHq/PU2IjpWnp/
wrJQXMk94IIrGjY8QHfCnpuMENCucVaifgAhwyeyuO5KiqUmD8E0RmcsotHKBV9X8H5eqLXd8zMQaPl
+Ub7j5PG+9KftQu0F/QhdFvpSLsHaxvBzA5nhIltjkaFcwGQnD1rpCM3+UnQE7Izoa5Yt1xoUWRwnF
+L2TKovW7+bYQ1kxsuuiX149jXTCJDVjkYCqi7HkrXYqcC1sbsror someuser@hostname"
  - name: ng-4
    instanceType: m5.large
    desiredCapacity: 1
    ssh: # enable SSH using SSM
      enableSsm: true
```

Unmanaged nodegroups

In `eksctl`, setting `--managed=false` or using the `nodeGroups` field creates an unmanaged nodegroup. Bear in mind that unmanaged nodegroups do not appear in the EKS console, which as a general rule only knows about EKS-managed nodegroups.

You should be upgrading nodegroups only after you ran `eksctl upgrade cluster`. (See Upgrading clusters.)

If you have a simple cluster with just an initial nodegroup (i.e. created with `eksctl create cluster`), the process is very simple:

1. Get the name of old nodegroup:

```
eksctl get nodegroups --cluster=<clusterName> --region=<region>
```

> **Note**
>
> ```
> You should see only one nodegroup here, if you see more - read the next
>  section.
> ```

2. Create a new nodegroup:

```
 eksctl create nodegroup --cluster=<clusterName> --region=<region> --
name=<newNodeGroupName> --managed=false
```

3. Delete the old nodegroup:

```
 eksctl delete nodegroup --cluster=<clusterName> --region=<region> --
name=<oldNodeGroupName>
```

> **Note**
>
> ```
> This will drain all pods from that nodegroup before the instances are deleted.
>  In some scenarios, Pod Disruption Budget (PDB) policies can prevent pods to
>  be evicted. To delete the nodegroup regardless of PDB, one should use the `--
> disable-eviction` flag, will bypass checking PDB policies.
> ```

Updating multiple nodegroups

If you have multiple nodegroups, it's your responsibility to track how each one was configured. You can do this by using config files, but if you haven't used it already, you will need to inspect your cluster to find out how each nodegroup was configured.

In general terms, you are looking to:

- review which nodegroups you have and which ones can be deleted or must be replaced for the new version
- note down configuration of each nodegroup, consider using config file to ease upgrades next time

Updating with config file

If you are using config file, you will need to do the following.

Edit config file to add new nodegroups, and remove old nodegroups. If you just want to upgrade nodegroups and keep the same configuration, you can just change nodegroup names, e.g. append `-v2` to the name.

To create all of new nodegroups defined in the config file, run:

```
eksctl create nodegroup --config-file=<path>
```

Once you have new nodegroups in place, you can delete old ones:

```
eksctl delete nodegroup --config-file=<path> --only-missing
```

> **Note**
>
> First run is in plan mode, if you are happy with the proposed changes, re-run with `--approve`.

Updating default add-ons

You may need to update the networking add-ons installed on your cluster. For more information, see the section called “Default add-on updates”.

EKS managed nodegroups

Amazon EKS managed nodegroups is a feature that automates the provisioning and lifecycle management of nodes (EC2 instances) for Amazon EKS Kubernetes clusters. Customers can provision optimized groups of nodes for their clusters and EKS will keep their nodes up to date with the latest Kubernetes and host OS versions.

An EKS managed node group is an autoscaling group and associated EC2 instances that are managed by AWS for an Amazon EKS cluster. Each node group uses the Amazon EKS-optimized Amazon Linux 2 AMI. Amazon EKS makes it easy to apply bug fixes and security patches to nodes, as well as update them to the latest Kubernetes versions. Each node group launches an autoscaling group for your cluster, which can span multiple AWS VPC availability zones and subnets for high-availability.

NEW Launch Template support for managed nodegroups

> **Note**
>
> The term "unmanaged nodegroups" has been used to refer to nodegroups that eksctl has supported since the beginning (represented via the `nodeGroups` field). The `ClusterConfig` file continues to use the `nodeGroups` field for defining unmanaged nodegroups, and managed nodegroups are defined with the `managedNodeGroups` field.

Creating managed nodegroups

```
$ eksctl create nodegroup
```

New clusters

To create a new cluster with a managed nodegroup, run

```
eksctl create cluster
```

To create multiple managed nodegroups and have more control over the configuration, a config file can be used.

> **Note**
>
> Managed nodegroups do not have complete feature parity with unmanaged nodegroups.

```
# cluster.yaml
# A cluster with two managed nodegroups
---
```

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: managed-cluster
  region: us-west-2

managedNodeGroups:
  - name: managed-ng-1
    minSize: 2
    maxSize: 4
    desiredCapacity: 3
    volumeSize: 20
    ssh:
      allow: true
      publicKeyPath: ~/.ssh/ec2_id_rsa.pub
      # new feature for restricting SSH access to certain AWS security group IDs
      sourceSecurityGroupIds: ["sg-00241fbb12c607007"]
    labels: {role: worker}
    tags:
      nodegroup-role: worker
    iam:
      withAddonPolicies:
        externalDNS: true
        certManager: true

  - name: managed-ng-2
    instanceType: t2.large
    minSize: 2
    maxSize: 3
```

Another example of a config file for creating a managed nodegroup can be found here.

It's possible to have a cluster with both managed and unmanaged nodegroups. Unmanaged nodegroups do not show up in the Amazon EKS console but `eksctl get nodegroup` will list both types of nodegroups.

```
# cluster.yaml
# A cluster with an unmanaged nodegroup and two managed nodegroups.
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
```

```
metadata:
  name: managed-cluster
  region: us-west-2

nodeGroups:
  - name: ng-1
    minSize: 2

managedNodeGroups:
  - name: managed-ng-1
    minSize: 2
    maxSize: 4
    desiredCapacity: 3
    volumeSize: 20
    ssh:
      allow: true
      publicKeyPath: ~/.ssh/ec2_id_rsa.pub
      # new feature for restricting SSH access to certain AWS security group IDs
      sourceSecurityGroupIds: ["sg-00241fbb12c607007"]
    labels: {role: worker}
    tags:
      nodegroup-role: worker
    iam:
      withAddonPolicies:
        externalDNS: true
        certManager: true

  - name: managed-ng-2
    instanceType: t2.large
    privateNetworking: true
    minSize: 2
    maxSize: 3
```

NEW Support for custom AMI, security groups, `instancePrefix`, `instanceName`, `ebsOptimized`, `volumeType`, `volumeName`, `volumeEncrypted`, `volumeKmsKeyID`, `volumeIOPS`, `maxPodsPerNode`, `preBootstrapCommands`, `overrideBootstrapCommand`, and `disableIMDSv1`

```
# cluster.yaml
# A cluster with a managed nodegroup with customization.
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
```

```
metadata:
  name: managed-cluster
  region: us-west-2

managedNodeGroups:
  - name: custom-ng
    ami: ami-0e124de4755b2734d
    securityGroups:
      attachIDs: ["sg-1234"]
    maxPodsPerNode: 80
    ssh:
      allow: true
    volumeSize: 100
    volumeName: /dev/xvda
    volumeEncrypted: true
    # defaults to true, which enforces the use of IMDSv2 tokens
    disableIMDSv1: false
    overrideBootstrapCommand: |
      #!/bin/bash
      /etc/eks/bootstrap.sh managed-cluster --kubelet-extra-args '--node-
labels=eks.amazonaws.com/nodegroup=custom-ng,eks.amazonaws.com/nodegroup-
image=ami-0e124de4755b2734d'
```

If you are requesting an instance type that is only available in one zone (and the eksctl config requires specification of two) make sure to add the availability zone to your node group request:

```
# cluster.yaml
# A cluster with a managed nodegroup with "availabilityZones"
---

apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: flux-cluster
  region: us-east-2
  version: "1.23"

availabilityZones: ["us-east-2b", "us-east-2c"]
managedNodeGroups:
  - name: workers
    instanceType: hpc6a.48xlarge
```

```
    minSize: 64
    maxSize: 64
    labels: { "fluxoperator": "true" }
    availabilityZones: ["us-east-2b"]
    efaEnabled: true
    placement:
      groupName: eks-efa-testing
```

This can be true for instance types like the Hpc6 family that are only available in one zone.

Existing clusters

```
eksctl create nodegroup --managed
```

Tip : if you are using a `ClusterConfig` file to describe your whole cluster, describe your new managed node group in the `managedNodeGroups` field and run:

```
eksctl create nodegroup --config-file=YOUR_CLUSTER.yaml
```

Upgrading managed nodegroups

You can update a nodegroup to the latest EKS-optimized AMI release version for the AMI type you are using at any time.

If your nodegroup is the same Kubernetes version as the cluster, you can update to the latest AMI release version for that Kubernetes version of the AMI type you are using. If your nodegroup is the previous Kubernetes version from the cluster's Kubernetes version, you can update the nodegroup to the latest AMI release version that matches the nodegroup's Kubernetes version, or update to the latest AMI release version that matches the clusters Kubernetes version. You cannot roll back a nodegroup to an earlier Kubernetes version.

To upgrade a managed nodegroup to the latest AMI release version:

```
eksctl upgrade nodegroup --name=managed-ng-1 --cluster=managed-cluster
```

The nodegroup can be upgraded to the latest AMI release for a specified Kubernetes version using:

```
eksctl upgrade nodegroup --name=managed-ng-1 --cluster=managed-cluster --kubernetes-
version=<kubernetes-version>
```

To upgrade to a specific AMI release version instead of the latest version, pass `--release-version`:

```
eksctl upgrade nodegroup --name=managed-ng-1 --cluster=managed-cluster --release-
version=1.19.6-20210310
```

> **Note**
>
> If the managed nodes are deployed using custom AMIs, the following workflow must be followed in order to deploy a new version of the custom AMI.

- initial deployment of the nodegroup must be done using a launch template. e.g.

```
managedNodeGroups:
  - name: launch-template-ng
    launchTemplate:
      id: lt-1234
      version: "2" #optional (uses the default version of the launch template if
 unspecified)
```

- create a new version of the custom AMI (using Amazon EKS console).
- create a new launch template version with the new AMI ID (using Amazon EKS console).
- upgrade the nodes to the new version of the launch template. e.g.

```
eksctl upgrade nodegroup --name nodegroup-name --cluster cluster-name --launch-
template-version new-template-version
```

Handling parallel upgrades for nodes

Multiple managed nodes can be upgraded simultaneously. To configure parallel upgrades, define the `updateConfig` of a nodegroup when creating the nodegroup. An example `updateConfig` can be found here.

To avoid any downtime to your workloads due to upgrading multiple nodes at once, you can limit the number of nodes that can become unavailable during an upgrade by specifying this in the `maxUnavailable` field of an `updateConfig`. Alternatively, use `maxUnavailablePercentage`,

which defines the maximum number of unavailable nodes as a percentage of the total number of nodes.

Note that `maxUnavailable` cannot be higher than `maxSize`. Also, `maxUnavailable` and `maxUnavailablePercentage` cannot be used simultaneously.

This feature is only available for managed nodes.

Updating managed nodegroups

`eksctl` allows updating the UpdateConfig section of a managed nodegroup. This section defines two fields. `MaxUnavailable` and `MaxUnavailablePercentage`. Your nodegroups are unaffected during the update, thus downtime shouldn't be expected.

The command `update nodegroup` should be used with a config file using the `--config-file` flag. The nodegroup should contain an `nodeGroup.updateConfig` section. More information can be found here.

Nodegroup Health issues

EKS Managed Nodegroups automatically checks the configuration of your nodegroup and nodes for health issues and reports them through the EKS API and console. To view health issues for a nodegroup:

```
eksctl utils nodegroup-health --name=managed-ng-1 --cluster=managed-cluster
```

Managing Labels

EKS Managed Nodegroups supports attaching labels that are applied to the Kubernetes nodes in the nodegroup. This is specified via the `labels` field in eksctl during cluster or nodegroup creation.

To set new labels or updating existing labels on a nodegroup:

```
eksctl set labels --cluster managed-cluster --nodegroup managed-ng-1 --labels
 kubernetes.io/managed-by=eks,kubernetes.io/role=worker
```

To unset or remove labels from a nodegroup:

```
eksctl unset labels --cluster managed-cluster --nodegroup managed-ng-1 --labels
 kubernetes.io/managed-by,kubernetes.io/role
```

To view all labels set on a nodegroup:

```
eksctl get labels --cluster managed-cluster --nodegroup managed-ng-1
```

Scaling Managed Nodegroups

`eksctl scale nodegroup` also supports managed nodegroups. The syntax for scaling a managed or unmanaged nodegroup is the same.

```
eksctl scale nodegroup --name=managed-ng-1 --cluster=managed-cluster --nodes=4 --nodes-min=3 --nodes-max=5
```

Further information

- EKS Managed Nodegroups

Node bootstrapping

AmazonLinux2023

AL2023 introduced a new node initialization process nodeadm that uses a YAML configuration schema, dropping the use of `/etc/eks/bootstrap.sh` script.

> **Note**
>
> With Kubernetes versions 1.30 and above, Amazon Linux 2023 is the default OS.

Default settings for AL2

For self-managed nodes and EKS-managed nodes based on custom AMIs, `eksctl` creates a default, minimal, `NodeConfig` and automatically injects it into the nodegroups's launch template userdata. i.e.

```
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=//
```

```
--//
Content-Type: application/node.eks.aws

apiVersion: node.eks.aws/v1alpha1
kind: NodeConfig
spec:
  cluster:
    apiServerEndpoint: https://XXXX.us-west-2.eks.amazonaws.com
    certificateAuthority: XXXX
    cidr: 10.100.0.0/16
    name: my-cluster
  kubelet:
    config:
      clusterDNS:
      - 10.100.0.10
    flags:
    - --node-labels=alpha.eksctl.io/cluster-name=my-cluster,alpha.eksctl.io/nodegroup-
name=my-nodegroup
    - --register-with-taints=special=true:NoSchedule

--//--
```

For EKS-managed nodes based on native AMIs, the default `NodeConfig` is being added by EKS MNG under the hood, appended directly to the EC2's userdata. Thus, in this scenario, `eksctl` does not need to include it within the launch template.

Configuring the bootstrapping process

To set advanced properties of `NodeConfig`, or simply override the default values, eksctl allows you to specify a custom `NodeConfig` via `nodeGroup.overrideBootstrapCommand` or `managedNodeGroup.overrideBootstrapCommand` e.g.

```
managedNodeGroups:
  - name: mng-1
    amiFamily: AmazonLinux2023
    ami: ami-0253856dd7ab7dbc8
    overrideBootstrapCommand: |
      apiVersion: node.eks.aws/v1alpha1
      kind: NodeConfig
      spec:
        instance:
          localStorage:
```

```
            strategy: RAID0
```

This custom config will be prepended to the userdata by eksctl, and merged by `nodeadm` with the default config. Read more about `nodeadm`'s capability of merging multiple configuration objects here.

Launch Template support for Managed Nodegroups

eksctl supports launching managed nodegroups using a provided EC2 Launch Template. This enables multiple customization options for nodegroups including providing custom AMIs and security groups, and passing user data for node bootstrapping.

Creating managed nodegroups using a provided launch template

```
# managed-cluster.yaml
# A cluster with two managed nodegroups
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: managed-cluster
  region: us-west-2

managedNodeGroups:
  - name: managed-ng-1
    launchTemplate:
      id: lt-12345
      version: "2" # optional (uses the default launch template version if unspecified)

  - name: managed-ng-2
    minSize: 2
    desiredCapacity: 2
    maxSize: 4
    labels:
      role: worker
    tags:
      nodegroup-name: managed-ng-2
    privateNetworking: true
    launchTemplate:
      id: lt-12345
```

Upgrading a managed nodegroup to use a different launch template version

```
eksctl upgrade nodegroup --name=managed-ng-1 --cluster=managed-cluster --launch-
template-version=3
```

> **Note**
>
> If a launch template is using a custom AMI, then the new version should also use a custom AMI or the upgrade operation will fail

If a launch template is not using a custom AMI, the Kubernetes version to upgrade to can also be specified:

```
eksctl upgrade nodegroup --name=managed-ng-1 --cluster=managed-cluster --launch-
template-version=3 --kubernetes-version=1.17
```

Notes on custom AMI and launch template support

- When a launch template is provided, the following fields are not supported: `instanceType`, `ami`, `ssh.allow`, `ssh.sourceSecurityGroupIds`, `securityGroups`, `instancePrefix`, `instanceName`, `ebsOptimized`, `volumeEncrypted`, `volumeKmsKeyID`, `volumeIOPS`, `maxPodsPerNode`, `preBootstrapCommands`, `overrideBootstrapCommand` and `disableIMDSv1`.
- When using a custom AMI (`ami`), `overrideBootstrapCommand` must also be set to perform the bootstrapping.
- `overrideBootstrapCommand` can only be set when using a custom AMI.
- When a launch template is provided, tags specified in the nodegroup config apply to the EKS Nodegroup resource only and are not propagated to EC2 instances.

Custom subnets

It's possible to extend an existing VPC with a new subnet and add a Nodegroup to that subnet.

Why

Should the cluster run out of pre-configured IPs, it's possible to resize the existing VPC with a new CIDR to add a new subnet to it. To see how to do that, read this guide on AWS Extending VPCs.

TL;DR

Go to the VPC's configuration and add click on Actions->Edit CIDRs and add a new range. For example:

```
192.168.0.0/19 -> existing CIDR
+ 192.169.0.0/19 -> new CIDR
```

Now you need to add a new Subnet. Depending on if it's a new Private or a Public subnet, you will have to copy the routing information from a private or a public subnet respectively.

Once the subnet is created, add routing, and copy either the NAT gateway ID or the Internet Gateway from another subnet in the VPC. Take care that if it's a public subnet Enable Automatic IP Assignment. Actions->Modify auto-assign IP settings->Enable auto-assign public IPv4 address.

Don't forget to also copy the TAGS of the existing subnets depending on Public or Private subnet configuration. This is important, otherwise the subnet will not be part of the cluster and instances in the subnet will be unable to join.

When finished, copy the new subnet's ID. Repeat as often as necessary.

How

To create a nodegroup in the created subnet(s) run the following command:

```
eksctl create nodegroup --cluster <cluster-name> --name my-new-subnet --subnet-ids
 subnet-0edeb3a04bec27141,subnet-0edeb3a04bec27142,subnet-0edeb3a04bec27143
# or for a single subnet id
eksctl create nodegroup --cluster <cluster-name> --name my-new-subnet --subnet-ids
 subnet-0edeb3a04bec27141
```

Or, use the configuration as such:

```
eksctl create nodegroup -f cluster-managed.yaml
```

With a configuration like this:

```
# A simple example of ClusterConfig object with two nodegroups:
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: cluster-3
  region: eu-north-1

nodeGroups:
  - name: new-subnet-nodegroup
    instanceType: m5.large
    desiredCapacity: 1
    subnets:
      - subnet-id1
      - subnet-id2
```

Wait for the nodegroup to be created and the new instances should have the new IP ranges of the subnet(s).

Deleting the cluster

Since the new addition modified the existing VPC by adding a dependency outside of the CloudFormation stack, CloudFormation can no longer remove the cluster.

Before deleting the cluster, remove all created extra subnets by hand, then proceed by calling `eksctl`:

```
eksctl delete cluster -n <cluster-name> --wait
```

Custom DNS

There are two ways of overwriting the DNS server IP address used for all the internal and external DNS lookups. This is the equivalent of the `--cluster-dns` flag for the `kubelet`.

The first, is through the `clusterDNS` field. Config files accepts a `string` field called `clusterDNS` with the IP address of the DNS server to use. This will be passed to the `kubelet` that in turn will

pass it to the pods through the `/etc/resolv.conf` file. For more information, see the schema of the config file.

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: cluster-1
  region: eu-north-1

nodeGroups:
- name: ng-1
  clusterDNS: 169.254.20.10
```

Note that this configuration only accepts one IP address. To specify more than one address, use the `kubeletExtraConfig` parameter:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: cluster-1
  region: eu-north-1

nodeGroups:
  - name: ng-1
    kubeletExtraConfig:
      clusterDNS: ["169.254.20.10","172.20.0.10"]
```

Taints

To apply taints to a specific nodegroup use the `taints` config section like this:

```
    taints:
      - key: your.domain.com/db
        value: "true"
        effect: NoSchedule
      - key: your.domain.com/production
        value: "true"
        effect: NoExecute
```

A full example can be found here.

Instance Selector

eksctl supports specifying multiple instance types for managed and self-managed nodegroups, but with over 270 EC2 instance types, users have to spend time figuring out which instance types would be well suited for their nodegroup. It's even harder when using Spot instances because you need to choose a set of instances that works together well with the Cluster Autoscaler.

eksctl now integrates with the EC2 instance selector, which addresses this problem by generating a list of instance types based on resource criteria: vCPUs, memory, # of GPUs and CPU architecture. When the instance selector criteria is passed, eksctl creates a nodegroup with the instance types set to the instance types matching the supplied criteria.

Create cluster and nodegroups

To create a cluster with a single nodegroup that uses instance types matched by the instance selector resource criteria passed to eksctl, run

```
eksctl create cluster --instance-selector-vcpus=2 --instance-selector-memory=4
```

This will create a cluster and a managed nodegroup with the `instanceTypes` field set to `[c5.large, c5a.large, c5ad.large, c5d.large, t2.medium, t3.medium, t3a.medium]` (the set of instance types returned may change).

For unmanaged nodegroups, the `instancesDistribution.instanceTypes` field will be set:

```
eksctl create cluster --managed=false --instance-selector-vcpus=2 --instance-selector-
memory=4
```

The instance selector criteria can also be specified in ClusterConfig:

```
# instance-selector-cluster.yaml
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: cluster
  region: us-west-2
```

```
nodeGroups:
- name: ng
  instanceSelector:
    vCPUs: 2
    memory: "4" # 4 GiB, unit defaults to GiB

managedNodeGroups:
- name: mng
  instanceSelector:
    vCPUs: 2
    memory: 2GiB #
    cpuArchitecture: x86_64 # default value
```

```
eksctl create cluster -f instance-selector-cluster.yaml
```

The following instance selector CLI options are supported by `eksctl create cluster` and `eksctl create nodegroup`:

`--instance-selector-vcpus`, `--instance-selector-memory`, `--instance-selector-gpus` and `instance-selector-cpu-architecture`

An example file can be found here.

Dry Run

The dry-run feature allows you to inspect and change the instances matched by the instance selector before proceeding to creating a nodegroup.

```
eksctl create cluster --name development --instance-selector-vcpus=2 --instance-
selector-memory=4 --dry-run

apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
# ...
managedNodeGroups:
- amiFamily: AmazonLinux2
  instanceSelector:
    memory: "4"
    vCPUs: 2
  instanceTypes:
  - c5.large
```

```
  - c5a.large
  - c5ad.large
  - c5d.large
  - t2.medium
  - t3.medium
  - t3a.medium
...
# other config
```

The generated ClusterConfig can then be passed to `eksctl create cluster`:

```
eksctl create cluster -f generated-cluster.yaml
```

The `instanceSelector` field representing the CLI options will also be added to the ClusterConfig file for visibility and documentation purposes. When `--dry-run` is omitted, this field will be ignored and the `instanceTypes` field will be used, otherwise any changes to `instanceTypes` would get overridden by eksctl.

When a ClusterConfig file is passed with `--dry-run`, eksctl will output a ClusterConfig file containing the same set of nodegroups after expanding each nodegroup's instance selector resource criteria.

```
# instance-selector-cluster.yaml
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: cluster
  region: us-west-2

nodeGroups:
- name: ng
  instanceSelector:
    vCPUs: 2
    memory: 4 # 4 GiB, unit defaults to GiB

managedNodeGroups:
- name: mng
  instanceSelector:
    vCPUs: 2
    memory: 2GiB #
```

```
    cpuArchitecture: x86_64 # default value
```

```
eksctl create cluster -f instance-selector-cluster.yaml --dry-run

apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
# ...
managedNodeGroups:
- amiFamily: AmazonLinux2
  # ...
  instanceSelector:
    cpuArchitecture: x86_64
    memory: 2GiB
    vCPUs: 2
  instanceTypes:
  - t3.small
  - t3a.small
nodeGroups:
- amiFamily: AmazonLinux2
  # ...
  instanceSelector:
    memory: "4"
    vCPUs: 2
  instanceType: mixed
  instancesDistribution:
    capacityRebalance: false
    instanceTypes:
    - c5.large
    - c5a.large
    - c5ad.large
    - c5d.large
    - t2.medium
    - t3.medium
    - t3a.medium
# ...
```

Spot instances

Managed Nodegroups

`eksctl` supports Spot worker nodes using EKS Managed Nodegroups, a feature that allows EKS customers with fault-tolerant applications to easily provision and manage EC2 Spot Instances for

their EKS clusters. EKS Managed Nodegroup will configure and launch an EC2 Autoscaling group of Spot Instances following Spot best practices and draining Spot worker nodes automatically before the instances are interrupted by AWS. There is no incremental charge to use this feature and customers pay only for using the AWS resources, such as EC2 Spot Instances and EBS volumes.

To create a cluster with a managed nodegroup using Spot instances, pass the `--spot` flag and an optional list of instance types:

```
eksctl create cluster --spot --instance-types=c3.large,c4.large,c5.large
```

To create a managed nodegroup using Spot instances on an existing cluster:

```
eksctl create nodegroup --cluster=<clusterName> --spot --instance-
types=c3.large,c4.large,c5.large
```

To create Spot instances using managed nodegroups via a config file:

```
# spot-cluster.yaml

apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: spot-cluster
  region: us-west-2

managedNodeGroups:
- name: spot
  instanceTypes: ["c3.large","c4.large","c5.large","c5d.large","c5n.large","c5a.large"]
  spot: true


# `instanceTypes` defaults to [`m5.large`]
- name: spot-2
  spot: true

# On-Demand instances
- name: on-demand
  instanceTypes: ["c3.large", "c4.large", "c5.large"]
```

```
eksctl create cluster -f spot-cluster.yaml
```

> **Note**
>
> Unmanaged nodegroups do not support the `spot` and `instanceTypes` fields, instead the `instancesDistribution` field is used to configure Spot instances. See below

Further information

- EKS Spot Nodegroups
- EKS Managed Nodegroup Capacity Types

Unmanaged Nodegroups

`eksctl` has support for spot instances through the MixedInstancesPolicy for Auto Scaling Groups.

Here is an example of a nodegroup that uses 50% spot instances and 50% on demand instances:

```
nodeGroups:
  - name: ng-1
    minSize: 2
    maxSize: 5
    instancesDistribution:
      maxPrice: 0.017
      instanceTypes: ["t3.small", "t3.medium"] # At least one instance type should be
 specified
      onDemandBaseCapacity: 0
      onDemandPercentageAboveBaseCapacity: 50
      spotInstancePools: 2
```

Note that the `nodeGroups.X.instanceType` field shouldn't be set when using the `instancesDistribution` field.

This example uses GPU instances:

```
nodeGroups:
  - name: ng-gpu
    instanceType: mixed
    desiredCapacity: 1
    instancesDistribution:
```

```
      instanceTypes:
        - p2.xlarge
        - p2.8xlarge
        - p2.16xlarge
      maxPrice: 0.50
```

This example uses the capacity-optimized spot allocation strategy:

```
nodeGroups:
  - name: ng-capacity-optimized
    minSize: 2
    maxSize: 5
    instancesDistribution:
      maxPrice: 0.017
      instanceTypes: ["t3.small", "t3.medium"] # At least one instance type should be
 specified
      onDemandBaseCapacity: 0
      onDemandPercentageAboveBaseCapacity: 50
      spotAllocationStrategy: "capacity-optimized"
```

This example uses the capacity-optimized-prioritized spot allocation strategy:

```
nodeGroups:
  - name: ng-capacity-optimized-prioritized
    minSize: 2
    maxSize: 5
    instancesDistribution:
      maxPrice: 0.017
      instanceTypes: ["t3a.small", "t3.small"] # At least two instance types should be
 specified
      onDemandBaseCapacity: 0
      onDemandPercentageAboveBaseCapacity: 0
      spotAllocationStrategy: "capacity-optimized-prioritized"
```

Use the `capacity-optimized-prioritized` allocation strategy and then set the order of instance types in the list of launch template overrides from highest to lowest priority (first to last in the list). Amazon EC2 Auto Scaling honors the instance type priorities on a best-effort basis but optimizes for capacity first. This is a good option for workloads where the possibility of disruption must be minimized, but also the preference for certain instance types matters.For more information, see ASG Purchase Options.

Note that the `spotInstancePools` field shouldn't be set when using the `spotAllocationStrategy` field. If the `spotAllocationStrategy` is not specified, EC2 will default to use the `lowest-price` strategy.

Here is a minimal example:

```
nodeGroups:
  - name: ng-1
    instancesDistribution:
      instanceTypes: ["t3.small", "t3.medium"] # At least one instance type should be
 specified
```

To distinguish nodes between spot or on-demand instances you can use the kubernetes label `node-lifecycle` which will have the value `spot` or `on-demand` depending on its type.

Parameters in instancesDistribution

Please see the cluster config schema for details.

GPU Support

Eksctl supports selecting GPU instance types for nodegroups. Simply supply a compatible instance type to the create command, or via the config file.

```
eksctl create cluster --node-type=p2.xlarge
```

> **Note**
>
> It is no longer necessary to subscribe to the marketplace AMI for GPU support on EKS.

The AMI resolvers (`auto` and `auto-ssm`) will see that you want to use a GPU instance type and they will select the correct EKS optimized accelerated AMI.

Eksctl will detect that an AMI with a GPU-enabled instance type has been selected and will install the NVIDIA Kubernetes device plugin automatically.

> **Note**
>
> Windows and Ubuntu AMIs do not ship with GPU drivers installed, hence running GPU-accelerated workloads will not work out of the box.

To disable the automatic plugin installation, and manually install a specific version, use `--install-nvidia-plugin=false` with the create command. For example:

```
eksctl create cluster --node-type=p2.xlarge --install-nvidia-plugin=false
```

and, for versions 0.15.0 and above,

```
kubectl create -f https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/<VERSION>/
deployments/static/nvidia-device-plugin.yml
```

or, for older versions,

```
kubectl create -f https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/<VERSION>/
nvidia-device-plugin.yml
```

The installation of the NVIDIA Kubernetes device plugin will be skipped if the cluster only includes Bottlerocket nodegroups, since Bottlerocket already handles the execution of the device plugin. If you use different AMI families in your cluster's configurations, you may need to use taints and tolerations to keep the device plugin from running on Bottlerocket nodes.

ARM Support

This topic covers how to create a cluster with an ARM node group, and how to add an ARM node group to an existing cluster.

EKS supports 64-bit ARM architecture with its Graviton processors. To create a cluster, select one of the Graviton-based instance types (`a1`, `t4g`, `m6g`, `m7g`, `m6gd`, `c6g`, `c7g`, `c6gd`, `r6g`, `r7g`, `r6gd`, `m8g`, `r8g`, `c8g`) and run:

```
eksctl create cluster --node-type=a1.large
```

or use a config file:

```
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: cluster-arm-1
  region: us-west-2


nodeGroups:
  - name: ng-arm-1
    instanceType: m6g.medium
    desiredCapacity: 1
```

```
eksctl create cluster -f cluster-arm-1.yaml
```

ARM is also supported in managed nodegroups:

```
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: cluster-arm-2
  region: us-west-2

managedNodeGroups:
  - name: mng-arm-1
    instanceType: m6g.medium
    desiredCapacity: 1
```

```
eksctl create cluster -f cluster-arm-2.yaml
```

The AMI resolvers, `auto` and `auto-ssm`, will infer the correct AMI based on the ARM instance type. Only AmazonLinux2023, AmazonLinux2 and Bottlerocket families have EKS optimized AMIs for ARM.

> **Note**
>
> ARM is supported for clusters with version 1.15 and higher.

Auto Scaling

Enable Auto Scaling

You can create a cluster (or nodegroup in an existing cluster) with IAM role that will allow use of cluster autoscaler:

```
eksctl create cluster --asg-access
```

This flag also sets `k8s.io/cluster-autoscaler/enabled` and `k8s.io/cluster-autoscaler/<clusterName>` tags, so nodegroup discovery should work.

Once the cluster is running, you will need to install Cluster Autoscaler itself.

You should also add the following to your managed or unmanaged nodegroup definition(s) to add the tags required for the Cluster Autoscaler to scale the nodegroup:

```
nodeGroups:
  - name: ng1-public
    iam:
      withAddonPolicies:
        autoScaler: true
```

Scaling up from 0

If you would like to be able to scale your node group up from 0 and you have labels and/or taints defined on your nodegroups, you will need to propagate these as tags on your Auto Scaling Groups (ASGs).

One way to do this is by setting the ASG tags in the `tags` field of your nodegroup definitions. For example, given a nodegroup with the following labels and taints:

```
nodeGroups:
  - name: ng1-public
    ...
    labels:
      my-cool-label: pizza
    taints:
      key: feaster
      value: "true"
      effect: NoSchedule
```

You would need to add the following ASG tags:

```
nodeGroups:
  - name: ng1-public
    ...
    labels:
      my-cool-label: pizza
    taints:
      feaster: "true:NoSchedule"
    tags:
      k8s.io/cluster-autoscaler/node-template/label/my-cool-label: pizza
      k8s.io/cluster-autoscaler/node-template/taint/feaster: "true:NoSchedule"
```

For both managed and unmanaged nodegroups, this can be done automatically by setting `propagateASGTags` to `true`, which will add the labels and taints as tags to the Auto Scaling group:

```
nodeGroups:
  - name: ng1-public
    ...
    labels:
      my-cool-label: pizza
    taints:
      feaster: "true:NoSchedule"
    propagateASGTags: true
```

Zone-aware Auto Scaling

If your workloads are zone-specific you'll need to create separate nodegroups for each zone. This is because the `cluster-autoscaler` assumes that all nodes in a group are exactly equivalent. So, for example, if a scale-up event is triggered by a pod which needs a zone-specific PVC (e.g. an EBS volume), the new node might get scheduled in the wrong AZ and the pod will fail to start.

You won't need a separate nodegroup for each AZ if your environment meets the following criteria:

- No zone-specific storage requirements.
- No required podAffinity with topology other than host.
- No required nodeAffinity on zone label.
- No nodeSelector on a zone label.

(Read more here and here.)

If you meet all of the above requirements (and possibly others) then you should be safe with a single nodegroup which spans multiple AZs. Otherwise you'll want to create separate, single-AZ nodegroups:

BEFORE:

```
nodeGroups:
  - name: ng1-public
    instanceType: m5.xlarge
    # availabilityZones: ["eu-west-2a", "eu-west-2b"]
```

AFTER:

```
nodeGroups:
  - name: ng1-public-2a
    instanceType: m5.xlarge
    availabilityZones: ["eu-west-2a"]
  - name: ng1-public-2b
    instanceType: m5.xlarge
    availabilityZones: ["eu-west-2b"]
```

Custom AMI support

Setting the node AMI ID

The `--node-ami` flag enables a number of advanced use cases such as using a custom AMI or querying AWS in realtime to determine which AMI to use. The flag can be used for both non-GPU and GPU images.

The flag can take the AMI image id for an image to explicitly use. It also can take the following 'special' keywords:

Keyword	Description
auto	Indicates that the AMI to use for the nodes should be found by querying AWS EC2. This relates to the auto resolver.

Keyword	Description
auto-ssm	Indicates that the AMI to use for the nodes should be found by querying AWS SSM Parameter Store.

> **Note**
>
> At the moment, EKS managed nodegroups only support the following AMI Families when working with custom AMIs: `AmazonLinux2023`, `AmazonLinux2`, `Bottlerocket`, `Ubuntu2004`, `UbuntuPro2004`, `Ubuntu2204` and `Ubuntu2404`

When setting `--node-ami` to an ID string, `eksctl` will assume that a custom AMI has been requested. For AmazonLinux2 and Ubuntu nodes, both EKS managed and self-managed, this will mean that `overrideBootstrapCommand` is required. For AmazonLinux2023, since it stops using the `/etc/eks/bootstrap.sh` script for node bootstrapping, in favour of a nodeadm initialization process (for more information, please refer to node bootstrapping docs), `overrideBootstrapCommand` is not supported.

CLI flag examples:

```
eksctl create cluster --node-ami=auto

# with a custom ami id
eksctl create cluster --node-ami=ami-custom1234
```

Config file example:

```
nodeGroups:
  - name: ng1
    instanceType: p2.xlarge
    amiFamily: AmazonLinux2
    ami: auto
  - name: ng2
    instanceType: m5.large
    amiFamily: AmazonLinux2
    ami: ami-custom1234
```

```
managedNodeGroups:
  - name: m-ng-2
    amiFamily: AmazonLinux2
    ami: ami-custom1234
    instanceType: m5.large
    overrideBootstrapCommand: |
      #!/bin/bash
      /etc/eks/bootstrap.sh <cluster-name>
```

The `--node-ami` flag can also be used with `eksctl create nodegroup`.

Setting the node AMI Family

The `--node-ami-family` can take following keywords:

Keyword	Description
AmazonLinux2	Indicates that the EKS AMI image based on Amazon Linux 2 should be used (default).
AmazonLinux2023	Indicates that the EKS AMI image based on Amazon Linux 2023 should be used.
Ubuntu2004	Indicates that the EKS AMI image based on Ubuntu 20.04 LTS (Focal) should be used (supported for EKS ⇐ 1.29).
UbuntuPro2004	Indicates that the EKS AMI image based on Ubuntu Pro 20.04 LTS (Focal) should be used (available for EKS >= 1.27, ⇐ 1.29).
Ubuntu2204	Indicates that the EKS AMI image based on Ubuntu 22.04 LTS (Jammy) should be used (available for EKS >= 1.29).
UbuntuPro2204	Indicates that the EKS AMI image based on Ubuntu Pro 22.04 LTS (Jammy) should be used (available for EKS >= 1.29).

Keyword	Description
Ubuntu2404	Indicates that the EKS AMI image based on Ubuntu 24.04 LTS (Noble) should be used (available for EKS >= 1.31).
UbuntuPro2404	Indicates that the EKS AMI image based on Ubuntu Pro 24.04 LTS (Noble) should be used (available for EKS >= 1.31).
Bottlerocket	Indicates that the EKS AMI image based on Bottlerocket should be used.
WindowsServer2019FullContainer	Indicates that the EKS AMI image based on Windows Server 2019 Full Container should be used.
WindowsServer2019CoreContainer	Indicates that the EKS AMI image based on Windows Server 2019 Core Container should be used.
WindowsServer2022FullContainer	Indicates that the EKS AMI image based on Windows Server 2022 Full Container should be used.
WindowsServer2022CoreContainer	Indicates that the EKS AMI image based on Windows Server 2022 Core Container should be used.

CLI flag example:

```
eksctl create cluster --node-ami-family=AmazonLinux2
```

Config file example:

```
nodeGroups:
  - name: ng1
    instanceType: m5.large
    amiFamily: AmazonLinux2
```

```
managedNodeGroups:
  - name: m-ng-2
    instanceType: m5.large
    amiFamily: Ubuntu2204
```

The `--node-ami-family` flag can also be used with `eksctl create nodegroup`. `eksctl` requires AMI Family to be explicitly set via config file or via `--node-ami-family` CLI flag, whenever working with a custom AMI.

> **Note**
>
> At the moment, EKS managed nodegroups only support the following AMI Families when working with custom AMIs: `AmazonLinux2023`, `AmazonLinux2`, `Bottlerocket`, `Ubuntu2004`, `UbuntuPro2004`, `Ubuntu2204` and `Ubuntu2404`

Windows custom AMI support

Only self-managed Windows nodegroups can specify a custom AMI. `amiFamily` should be set to a valid Windows AMI family.

The following PowerShell variables will be available to the bootstrap script:

```
$EKSBootstrapScriptFile
$EKSClusterName
$APIServerEndpoint
$Base64ClusterCA
$ServiceCIDR
$KubeletExtraArgs
$KubeletExtraArgsMap: A hashtable containing arguments for the kubelet, e.g., @{ 'node-
labels' = ''; 'register-with-taints' = ''; 'max-pods' = '10'}
$DNSClusterIP
$ContainerRuntime
```

Config file example:

```
nodeGroups:
  - name: custom-windows
    amiFamily: WindowsServer2022FullContainer
    ami: ami-01579b74557facaf7
    overrideBootstrapCommand: |
```

```
    & $EKSBootstrapScriptFile -EKSClusterName "$EKSClusterName" -APIServerEndpoint
"$APIServerEndpoint" -Base64ClusterCA "$Base64ClusterCA" -ContainerRuntime
"containerd" -KubeletExtraArgs "$KubeletExtraArgs" 3>&1 4>&1 5>&1 6>&1
```

Bottlerocket custom AMI support

For Bottlerocket nodes, the `overrideBootstrapCommand` is not supported. Instead, to designate their own bootstrap container, one should use the `bottlerocket` field as part of the configuration file. E.g.

```
nodeGroups:
- name: bottlerocket-ng
  ami: ami-custom1234
  amiFamily: Bottlerocket
  bottlerocket:
    enableAdminContainer: true
    settings:
      bootstrap-containers:
        bootstrap:
          source: <MY-CONTAINER-URI>
```

Windows Worker Nodes

From version 1.14, Amazon EKS supports Windows Nodes that allow running Windows containers. In addition to having Windows nodes, a Linux node in the cluster is required to run CoreDNS, as Microsoft doesn't support host-networking mode yet. Thus, a Windows EKS cluster will be a mixture of Windows nodes and at least one Linux node. The Linux nodes are critical to the functioning of the cluster, and thus, for a production-grade cluster, it's recommended to have at least two `t2.large` Linux nodes for HA.

> **Note**
>
> You no longer need to install the VPC resource controller on Linux worker nodes to run Windows workloads in EKS clusters created after October 22, 2021. You can enable Windows IP address management on the EKS control plane via a ConfigMap setting (see link:eks/latest/userguide/windows-support.html for details). eksctl will automatically patch the ConfigMap to enable Windows IP address management when a Windows nodegroup is created.

Creating a new cluster with Windows support

The config file syntax allows creating a fully-functioning cluster with Windows support in a single command:

```
# cluster.yaml
# An example of ClusterConfig containing Windows and Linux node groups to support
 Windows workloads
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: windows-cluster
  region: us-west-2

nodeGroups:
  - name: windows-ng
    amiFamily: WindowsServer2019FullContainer
    minSize: 2
    maxSize: 3

managedNodeGroups:
  - name: linux-ng
    instanceType: t2.large
    minSize: 2
    maxSize: 3

  - name: windows-managed-ng
    amiFamily: WindowsServer2019FullContainer
    minSize: 2
    maxSize: 3
```

```
eksctl create cluster -f cluster.yaml
```

To create a new cluster with Windows un-managed nodegroup without using a config file, issue the following commands:

```
eksctl create cluster --managed=false --name=windows-cluster --node-ami-
family=WindowsServer2019CoreContainer
```

Adding Windows support to an existing Linux cluster

To enable running Windows workloads on an existing cluster with Linux nodes (`AmazonLinux2` AMI family), you need to add a Windows nodegroup.

NEW Support for Windows managed nodegroup has been added (--managed=true or omit the flag).

```
eksctl create nodegroup --managed=false --cluster=existing-cluster --node-ami-
family=WindowsServer2019CoreContainer
eksctl create nodegroup --cluster=existing-cluster --node-ami-
family=WindowsServer2019CoreContainer
```

To ensure workloads are scheduled on the right OS, they must have a `nodeSelector` targeting the OS it must run on:

```
# Targeting Windows
  nodeSelector:
    kubernetes.io/os: windows
    kubernetes.io/arch: amd64
```

```
# Targeting Linux
  nodeSelector:
    kubernetes.io/os: linux
    kubernetes.io/arch: amd64
```

If you are using a cluster older than `1.19` the `kubernetes.io/os` and `kubernetes.io/arch` labels need to be replaced with `beta.kubernetes.io/os` and `beta.kubernetes.io/arch` respectively.

Further information

- EKS Windows Support

Additional Volume Mappings

As an additional configuration option, when dealing with volume mappings, it's possible to configure extra mappings when the nodegroup is created.

To do this, set the field `additionalVolumes` as follows:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: dev-cluster
  region: eu-north-1

managedNodeGroups:
  - name: ng-1-workers
    labels: { role: workers }
    instanceType: m5.xlarge
    desiredCapacity: 10
    volumeSize: 80
    additionalVolumes:
      - volumeName: '/tmp/mount-1' # required
        volumeSize: 80
        volumeType: 'gp3'
        volumeEncrypted: true
        volumeKmsKeyID: 'id'
        volumeIOPS: 3000
        volumeThroughput: 125
      - volumeName: '/tmp/mount-2'  # required
        volumeSize: 80
        volumeType: 'gp2'
        snapshotID: 'snapshot-id'
```

For more details about selecting volumeNames, see the device naming documentation. To find out more about EBS volumes, Instance volume limits or Block device mappings visit this page.

EKS Hybrid Nodes

Introduction

Amazon EKS introduces Hybrid Nodes, a new feature that enables you to run on-premises and edge applications on customer-managed infrastructure with the same Amazon EKS clusters, features, and tools you use in AWS. Amazon EKS Hybrid Nodes brings an AWS-managed Kubernetes experience to on-premises environments for customers to simplify and standardize how you run applications across on-premises, edge and cloud environments. Read more at EKS Hybrid Nodes.

To facilitate support for this feature, eksctl introduces a new top-level field called `remoteNetworkConfig`. Any Hybrid Nodes related configuration shall be set up via this field,

as part of the config file; there are no CLI flags counterparts. Additionally, at launch, any remote network config can only be set up during cluster creation and cannot be updated afterwards. This means, you won't be able to update existing clusters to use Hybrid Nodes.

The `remoteNetworkConfig` section of the config file allows you to setup the two core areas when it comes to joining remote nodes to you EKS clusters: **networking** and **credentials**.

Networking

EKS Hybrid Nodes is flexible to your preferred method of connecting your on-premises network(s) to a VPC. There are several documented options available, including AWS Site-to-Site VPN and AWS Direct Connect, and you can choose the method that best fits your use case. In most of the methods you might choose, your VPC will be attached to either a virtual private gateway (VGW) or a transit gateway (TGW). If you rely on eksctl to create a VPC for you, eksctl will also configure, **within the scope of your VPC**, any networking related pre-requisites in order to facilitate communication between your EKS control plane and the remote nodes i.e.

- ingress/egress SG rules
- routes in the private subnets' route tables
- the VPC gateway attachment to the given TGW or VGW

Example config file:

```
remoteNetworkConfig:
  vpcGatewayID: tgw-xxxx # either VGW or TGW to be attached to your VPC
  remoteNodeNetworks:
    # eksctl will create, behind the scenes, SG rules, routes, and a VPC gateway
 attachment,
    # to facilitate communication between remote network(s) and EKS control plane, via
 the attached gateway
    - cidrs: ["10.80.146.0/24"]
  remotePodNetworks:
    - cidrs: ["10.86.30.0/23"]
```

If your connectivity method of choice does not involve using a TGW or VGW, you must not rely on eksctl to create the VPC for you, and instead provide a pre-existing one. On a related note, if you are using a pre-existing VPC, eksctl won't make any amendments to it, and ensuring all networking requirements are in place falls under your responsibility.

Note

eksctl does not setup any networking infrastructure outside your AWS VPC (i.e. any infrastructure from VGW/TGW to the remote networks)

Credentials

EKS Hybrid Nodes use the AWS IAM Authenticator and temporary IAM credentials provisioned by either **AWS SSM** or **AWS IAM Roles Anywhere** to authenticate with the EKS cluster. Similar to the self-managed nodegroups, if not otherwise provided, eksctl will create for you a Hybrid Nodes IAM Role to be assumed by the remote nodes. Additionally, when using IAM Roles Anywhere as your credentials provider, eksctl will setup a profile, and trust anchor based on a given certificate authority bundle (`iam.caBundleCert`) e.g.

```
remoteNetworkConfig:
  iam:
    # the provider for temporary IAM credentials. Default is SSM.
    provider: IRA
    # the certificate authority bundle that serves as the root of trust,
    # used to validate the X.509 certificates provided by your nodes.
    # can only be set when provider is IAMRolesAnywhere.
    caBundleCert: xxxx
```

The ARN of the Hybrid Nodes Role created by eksctl is needed later in the process of joining your remote nodes to the cluster, to setup `NodeConfig` for `nodeadm`, and to create activations (if using SSM). To fetch it, use:

```
aws cloudformation describe-stacks \
  --stack-name eksctl-<CLUSTER_NAME>-cluster \
  --query 'Stacks[].Outputs[?OutputKey==`RemoteNodesRoleARN`].[OutputValue]' \
  --output text
```

Similarly, if using IAM Roles Anywhere, you can fetch the ARN of the trust anchor and of the anywhere profile created by eksctl, amending the previous command by replacing `RemoteNodesRoleARN` with `RemoteNodesTrustAnchorARN` or `RemoteNodesAnywhereProfileARN`, respectively.

If you have a pre-existing IAM Roles Anywhere configuration in place, or you are using SSM, you can provide a IAM Role for Hybrid nodes via `remoteNetworkConfig.iam.roleARN`. Bear in mind that in this scenario, eksctl won't create the trust anchor and anywhere profile for you. e.g.

```
remoteNetworkConfig:
  iam:
    roleARN: arn:aws:iam::000011112222:role/HybridNodesRole
```

To map the role to a Kubernetes identity and authorise the remote nodes to join the EKS cluster, eksctl creates an access entry with Hybrid Nodes IAM Role as principal ARN and of type `HYBRID_LINUX`. i.e.

```
eksctl get accessentry --cluster my-cluster --principal-arn
 arn:aws:iam::000011112222:role/eksctl-my-cluster-clust-HybridNodesSSMRole-XiIAg0d29PkO
 --output json
[
    {
        "principalARN": "arn:aws:iam::000011112222:role/eksctl-my-cluster-clust-
HybridNodesSSMRole-XiIAg0d29PkO",
        "kubernetesGroups": [
            "system:nodes"
        ]
    }
]
```

Add-ons support

Container Networking Interface (CNI): The AWS VPC CNI can't be used with hybrid nodes. The core capabilities of Cilium and Calico are supported for use with hybrid nodes. You can manage your CNI with your choice of tooling such as Helm. For more information, see Configure a CNI for hybrid nodes.

> **Note**
>
> If you install VPC CNI in your cluster for your self-managed or EKS-managed nodegroups, you have to use `v1.19.0-eksbuild.1` or later, as this includes an update to the add-on's daemonset to exclude it from being installed on Hybrid Nodes.

Further references

- EKS Hybrid Nodes UserDocs
- Launch Announcement

Support Node Repair Configuration for EKS Managed Nodegroups

EKS Managed Nodegroups supports Node Repair, where the health of managed nodes are monitored, and unhealthy worker nodes are replaced or rebooted in response. eksctl now provides comprehensive configuration options for fine-grained control over node repair behavior.

Basic Node Repair Configuration

Using CLI flags

To create a cluster with a managed nodegroup using basic node repair, pass the `--enable-node-repair` flag:

```
eksctl create cluster --enable-node-repair
```

To create a managed nodegroup with node repair on an existing cluster:

```
eksctl create nodegroup --cluster=<clusterName> --enable-node-repair
```

Using configuration files

```
# basic-node-repair.yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: basic-node-repair-cluster
  region: us-west-2

managedNodeGroups:
- name: ng-1
```

```
  nodeRepairConfig:
    enabled: true
```

```
eksctl create cluster -f basic-node-repair.yaml
```

Enhanced Node Repair Configuration

Threshold Configuration

You can configure when node repair actions will stop using either percentage or count-based thresholds. **Note: You cannot use both percentage and count thresholds at the same time.**

CLI flags for thresholds

```
# Percentage-based threshold - repair stops when 20% of nodes are unhealthy
eksctl create cluster --enable-node-repair \
  --node-repair-max-unhealthy-percentage=20

# Count-based threshold - repair stops when 5 nodes are unhealthy
eksctl create cluster --enable-node-repair \
  --node-repair-max-unhealthy-count=5
```

Configuration file for thresholds

```
managedNodeGroups:
- name: threshold-ng
  nodeRepairConfig:
    enabled: true
    # Stop repair actions when 20% of nodes are unhealthy
    maxUnhealthyNodeThresholdPercentage: 20
    # Alternative: stop repair actions when 3 nodes are unhealthy
    # maxUnhealthyNodeThresholdCount: 3
    # Note: Cannot use both percentage and count thresholds simultaneously
```

Parallel Repair Limits

Control the maximum number of nodes that can be repaired concurrently or in parallel. This gives you finer-grained control over the pace of node replacements. **Note: You cannot use both percentage and count limits at the same time.**

CLI flags for parallel limits

```
# Percentage-based parallel limits - repair at most 15% of unhealthy nodes in parallel
eksctl create cluster --enable-node-repair \
  --node-repair-max-parallel-percentage=15

# Count-based parallel limits - repair at most 2 unhealthy nodes in parallel
eksctl create cluster --enable-node-repair \
  --node-repair-max-parallel-count=2
```

Configuration file for parallel limits

```
managedNodeGroups:
- name: parallel-ng
  nodeRepairConfig:
    enabled: true
    # Repair at most 15% of unhealthy nodes in parallel
    maxParallelNodesRepairedPercentage: 15
    # Alternative: repair at most 2 unhealthy nodes in parallel
    # maxParallelNodesRepairedCount: 2
    # Note: Cannot use both percentage and count limits simultaneously
```

Custom Repair Overrides

Specify granular overrides for specific repair actions. These overrides control the repair action and the repair delay time before a node is considered eligible for repair. **If you use this, you must specify all the values for each override.**

```
managedNodeGroups:
- name: custom-repair-ng
  instanceType: g4dn.xlarge  # GPU instances
  nodeRepairConfig:
    enabled: true
    maxUnhealthyNodeThresholdPercentage: 25
    maxParallelNodesRepairedCount: 1
    nodeRepairConfigOverrides:
      # Handle GPU-related failures with immediate termination
      - nodeMonitoringCondition: "AcceleratedInstanceNotReady"
        nodeUnhealthyReason: "NvidiaXID13Error"
        minRepairWaitTimeMins: 10
        repairAction: "Terminate"
      # Handle network issues with restart after waiting
```

```
      - nodeMonitoringCondition: "NetworkNotReady"
        nodeUnhealthyReason: "InterfaceNotUp"
        minRepairWaitTimeMins: 20
        repairAction: "Restart"
```

Complete Configuration Examples

For a comprehensive example with all configuration options, see examples/44-node-repair.yaml.

Example 1: Basic repair with percentage thresholds

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: basic-repair-cluster
  region: us-west-2

managedNodeGroups:
- name: basic-ng
  instanceType: m5.large
  desiredCapacity: 3
  nodeRepairConfig:
    enabled: true
    maxUnhealthyNodeThresholdPercentage: 20
    maxParallelNodesRepairedPercentage: 15
```

Example 2: Conservative repair for critical workloads

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: critical-workload-cluster
  region: us-west-2

managedNodeGroups:
- name: critical-ng
  instanceType: c5.2xlarge
  desiredCapacity: 6
  nodeRepairConfig:
    enabled: true
```

```
    # Very conservative settings
    maxUnhealthyNodeThresholdPercentage: 10
    maxParallelNodesRepairedCount: 1
    nodeRepairConfigOverrides:
      # Wait longer before taking action on critical workloads
      - nodeMonitoringCondition: "NetworkNotReady"
        nodeUnhealthyReason: "InterfaceNotUp"
        minRepairWaitTimeMins: 45
        repairAction: "Restart"
```

Example 3: GPU workload with specialized repair

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: gpu-workload-cluster
  region: us-west-2

managedNodeGroups:
- name: gpu-ng
  instanceType: g4dn.xlarge
  desiredCapacity: 4
  nodeRepairConfig:
    enabled: true
    maxUnhealthyNodeThresholdPercentage: 25
    maxParallelNodesRepairedCount: 1
    nodeRepairConfigOverrides:
      # GPU failures require immediate termination
      - nodeMonitoringCondition: "AcceleratedInstanceNotReady"
        nodeUnhealthyReason: "NvidiaXID13Error"
        minRepairWaitTimeMins: 5
        repairAction: "Terminate"
```

CLI Reference

Node Repair Flags

Flag	Description	Example
`--enable-node-repair`	Enable automatic node repair	`--enable-node-repair`
`--node-repair-max-unhealthy-percentage`	Maximum percentage of unhealthy nodes before repair	`--node-repair-max-unhealthy-percentage=20`
`--node-repair-max-unhealthy-count`	Maximum count of unhealthy nodes before repair	`--node-repair-max-unhealthy-count=5`
`--node-repair-max-parallel-percentage`	Maximum percentage of nodes to repair in parallel	`--node-repair-max-parallel-percentage=15`
`--node-repair-max-parallel-count`	Maximum count of nodes to repair in parallel	`--node-repair-max-parallel-count=2`

Note: Node repair config overrides are only supported through YAML configuration files due to their complexity.

Configuration Reference

nodeRepairConfig

Field	Type	Description	Constraints	Example
`enabled`	boolean	Enable/disable node repair	-	`true`
`maxUnhealthyNodeTh`	integer	Percentage threshold of unhealthy nodes, above which	Cannot be used with	`20`

Field	Type	Description	Constraints	Example
`resholdPercentage`		node auto repair actions will stop	`maxUnhealthyNodeThresholdCount`	
`maxUnhealthyNodeThresholdCount`	integer	Count threshold of unhealthy nodes, above which node auto repair actions will stop	Cannot be used with `maxUnhealthyNodeThresholdPercentage`	5
`maxParallelNodesRepairedPercentage`	integer	Maximum percentage of unhealthy nodes that can be repaired concurrently or in parallel	Cannot be used with `maxParallelNodesRepairedCount`	15
`maxParallelNodesRepairedCount`	integer	Maximum count of unhealthy nodes that can be repaired concurrently or in parallel	Cannot be used with `maxParallelNodesRepairedPercentage`	2
`nodeRepairConfigOverrides`	array	Granular overrides for specific repair actions controlling repair action and delay time	All values must be specified for each override	See examples above

nodeRepairConfigOverrides

Field	Type	Description	Valid Values
`nodeMonitoringCondition`	string	Unhealthy condition reported by the node monitoring agent that this override applies to	`"AcceleratedInstanceNotReady" , "NetworkNotReady"`
`nodeUnhealthyReason`	string	Reason reported by the node monitoring agent that this override applies to	`"NvidiaXID13Error" ,"InterfaceNotUp"`
`minRepairWaitTimeMins`	integer	Minimum time in minutes to wait before attempting to repair a node with the specified condition and reason	Any positive integer
`repairAction`	string	Repair action to take for nodes when all of the specified conditions are met	`"Terminate" , "Restart" , "NoAction"`

Further information

- EKS Managed Nodegroup Node Health

Networking

This chapter includes information about how Eksctl creates Virtual Private Cloud (VPC) networks for EKS clusters.

Topics:

- the section called “VPC Configuration”
  - Modify the VPC CIDR range and configure IPv6 addressing
  - Use an existing VPC
  - Customize the VPC, subnets, security groups, and NAT gateways for the new EKS cluster
- the section called “Subnet Settings”
  - Use private subnets for the initial nodegroup to isolate it from the public internet
  - Customize subnet topology by listing multiple subnets per availability zone and specifying subnets in nodegroup configurations
  - Restrict nodegroups to specific named subnets in the VPC configuration
  - When using private subnets for nodegroups, set `privateNetworking` to `true`
  - Provide a complete subnet specification with both `public` and `private` configurations in the VPC spec
  - Only one of `subnets` or `availabilityZones` can be provided in nodegroup configuration
- the section called “Cluster Access”
  - Manage public and private access to the Kubernetes API server endpoints in an EKS cluster
  - Restrict access to the EKS Kubernetes public API endpoint by specifying allowed CIDR ranges
  - Update the API server endpoint access configuration and public access CIDR restrictions for an existing cluster
- the section called “Control plane networking”
  - Update the subnets used by the EKS control plane for a cluster
- the section called “IPv6 Support”
  - Specify the IP version (IPv4 or IPv6) to be used when creating a VPC with EKS cluster

VPC Configuration

Dedicated VPC for Cluster

By default `eksctl create cluster` will create a dedicated VPC for the cluster. This is done in order to avoid interference with existing resources for a variety of reasons, including security, but also because it is challenging to detect all settings in an existing VPC.

- The default VPC CIDR used by `eksctl` is `192.168.0.0/16`.
  - It is divided into 8 (/19) subnets (3 private, 3 public & 2 reserved).
- The initial nodegroup is created in public subnets.
- SSH access is disabled unless `--allow-ssh` is specified.
- The nodegroup by default allows inbound traffic from the control plane security group on ports 1025 - 65535.

> **Note**
>
> In `us-east-1` eksctl only creates 2 public and 2 private subnets by default.

Change VPC CIDR

If you need to set up peering with another VPC, or simply need a larger or smaller range of IPs, you can use `--vpc-cidr` flag to change it. Please refer to the AWS docs for guides on choosing CIDR blocks which are permitted for use in a VPC.

If you are creating an IPv6 cluster you can configure `VPC.IPv6Cidr` in the cluster config file. This setting is only in the config file, not in a CLI flag.

If you own an IPv6 IP address block, you can also bring your own IPv6 pool. See Bring your own IP addresses (BYOIP) to Amazon EC2 on how to import your own pool. Then use the `VPC.IPv6Cidr` in the cluster config file to configure Eksctl.

Use an existing VPC: shared with kops

You can use the VPC of an existing Kubernetes cluster managed by kops. This feature is provided to facilitate migration and/or cluster peering.

If you have previously created a cluster with kops, e.g. using commands similar to this:

```
export KOPS_STATE_STORE=s3://kops
kops create cluster cluster-1.k8s.local --zones=us-west-2c,us-west-2b,us-west-2a --
networking=weave --yes
```

You can create an EKS cluster in the same AZs using the same VPC subnets (NOTE: at least 2 AZs/ subnets are required):

```
eksctl create cluster --name=cluster-2 --region=us-west-2 --vpc-from-kops-
cluster=cluster-1.k8s.local
```

Use existing VPC: other custom configuration

`eksctl` provides some, but not complete, flexibility for custom VPC and subnet topologies.

You can use an existing VPC by supplying private and/or public subnets using the `--vpc-private-subnets` and `--vpc-public-subnets` flags. It is up to you to ensure the subnets you use are categorised correctly, as there is no simple way to verify whether a subnet is actually private or public, because configurations vary.

Given these flags, `eksctl create cluster` will determine the VPC ID automatically, but it will not create any routing tables or other resources, such as internet/NAT gateways. It will, however, create dedicated security groups for the initial nodegroup and the control plane.

You must ensure to provide **at least 2 subnets in different AZs** and this condition is checked by EKS. If you use an existing VPC, the following requirements aren't enforced or checked by EKS or Eksctl and EKS creates the cluster. Some basic functions of the cluster work without these requirements. (For example, tagging is not strictly necessary, tests have shown that it is possible to create a functional cluster without any tags set on the subnets, however there is no guarantee that this will always hold and tagging is recommended.)

Standard requirements:

- all given subnets must be in the same VPC, within the same block of IPs
- a sufficient number IP addresses are available, based on needs
- sufficient number of subnets (minimum 2), based on needs
- subnets are tagged with at least the following:
  - `kubernetes.io/cluster/<name>` tag set to either `shared` or `owned`

  - `kubernetes.io/role/internal-elb` tag set to 1 for *private* subnets
  - `kubernetes.io/role/elb` tag set to 1 for *public* subnets
- correctly configured internet and/or NAT gateways
- routing tables have correct entries and the network is functional
- **NEW**: all public subnets should have the property `MapPublicIpOnLaunch` enabled (i.e. `Auto-assign public IPv4 address` in the AWS console). Managed node groups and Fargate don't assign public IPv4 addresses, the property must be set on the subnet.

There may be other requirements imposed by EKS or Kubernetes, and it is entirely up to you to stay up-to-date on any requirements and/or recommendations, and implement those as needed/possible.

Default security group settings applied by `eksctl` may or may not be sufficient for sharing access with resources in other security groups. If you wish to modify the ingress/egress rules of the security groups, you might need to use another tool to automate changes, or do it via EC2 console.

When in doubt, don't use a custom VPC. Using `eksctl create cluster` without any `--vpc-*` flags will always configure the cluster with a fully-functional dedicated VPC.

Examples

Create a cluster using a custom VPC with 2x private and 2x public subnets:

```
eksctl create cluster \
  --vpc-private-subnets=subnet-0ff156e0c4a6d300c,subnet-0426fb4a607393184 \
  --vpc-public-subnets=subnet-0153e560b3129a696,subnet-009fa0199ec203c37
```

or use the following equivalent config file:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: my-test
  region: us-west-2

vpc:
  id: "vpc-11111"
  subnets:
```

```
    private:
      us-west-2a:
          id: "subnet-0ff156e0c4a6d300c"
      us-west-2c:
          id: "subnet-0426fb4a607393184"
    public:
      us-west-2a:
          id: "subnet-0153e560b3129a696"
      us-west-2c:
          id: "subnet-009fa0199ec203c37"

nodeGroups:
  - name: ng-1
```

Create a cluster using a custom VPC with 3x private subnets and make initial nodegroup use those subnets:

```
eksctl create cluster \
  --vpc-private-
subnets=subnet-0ff156e0c4a6d300c,subnet-0549cdab573695c03,subnet-0426fb4a607393184 \
  --node-private-networking
```

or use the following equivalent config file:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: my-test
  region: us-west-2

vpc:
  id: "vpc-11111"
  subnets:
    private:
      us-west-2d:
          id: "subnet-0ff156e0c4a6d300c"
      us-west-2c:
          id: "subnet-0549cdab573695c03"
      us-west-2a:
          id: "subnet-0426fb4a607393184"

nodeGroups:
```

```
  - name: ng-1
    privateNetworking: true
```

Create a cluster using a custom VPC 4x public subnets:

```
eksctl create cluster \
  --vpc-public-
subnets=subnet-0153e560b3129a696,subnet-0cc9c5aebe75083fd,subnet-009fa0199ec203c37,subnet-018fa
```

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: my-test
  region: us-west-2

vpc:
  id: "vpc-11111"
  subnets:
    public:
      us-west-2d:
          id: "subnet-0153e560b3129a696"
      us-west-2c:
          id: "subnet-0cc9c5aebe75083fd"
      us-west-2a:
          id: "subnet-009fa0199ec203c37"
      us-west-2b:
          id: "subnet-018fa0176ba320e45"

nodeGroups:
  - name: ng-1
```

More examples can be found in the repo's `examples` folder:

- using an existing VPC
- using a custom VPC CIDR

Custom Shared Node Security Group

`eksctl` will create and manage a shared node security group that allows communication between unmanaged nodes and the cluster control plane and managed nodes.

If you wish to provide your own custom security group instead, you may override the `sharedNodeSecurityGroup` field in the config file:

```
vpc:
  sharedNodeSecurityGroup: sg-0123456789
```

By default, when creating the cluster, `eksctl` will add rules to this security group to allow communication to and from the default cluster security group that EKS creates. The default cluster security group is used by both the EKS control plane and managed node groups.

If you wish to manage the security group rules yourself, you may prevent `eksctl` from creating the rules by setting `manageSharedNodeSecurityGroupRules` to `false` in the config file:

```
vpc:
  sharedNodeSecurityGroup: sg-0123456789
  manageSharedNodeSecurityGroupRules: false
```

NAT Gateway

The NAT Gateway for a cluster can be configured to be `Disable`, `Single` (default) or `HighlyAvailable`. The `HighlyAvailable` option will deploy a NAT Gateway in each Availability Zone of the Region, so that if an AZ is down, nodes in the other AZs will still be able to communicate to the Internet.

It can be specified through the `--vpc-nat-mode` CLI flag or in the cluster config file like the example below:

```
vpc:
  nat:
    gateway: HighlyAvailable # other options: Disable, Single (default)
```

See the complete example here.

> **Note**
>
> Specifying the NAT Gateway is only supported during cluster creation. It isn't touched during a cluster upgrade.

Subnet Settings

Use private subnets for initial nodegroup

If you prefer to isolate the initial nodegroup from the public internet, you can use the `--node-private-networking` flag. When used in conjunction with the `--ssh-access` flag, the SSH port can only be accessed from inside the VPC.

> **Note**
>
> Using the `--node-private-networking` flag will result in outgoing traffic to go through the NAT gateway using its Elastic IP. On the other hand, if the nodes are in a public subnet, the outgoing traffic won't go through the NAT gateway and hence the outgoing traffic has the IP of each individual node.

Custom subnet topology

`eksctl` version `0.32.0` introduced further subnet topology customisation with the ability to:

- List multiple subnets per AZ in VPC configuration
- Specify subnets in nodegroup configuration

In earlier versions custom subnets had to be provided by availability zone, meaning just one subnet per AZ could be listed. From `0.32.0` the identifying keys can be arbitrary.

```
vpc:
  id: "vpc-11111"
  subnets:
    public:
      public-one:                          # arbitrary key
          id: "subnet-0153e560b3129a696"
      public-two:
          id: "subnet-0cc9c5aebe75083fd"
      us-west-2b:                          # or list by AZ
          id: "subnet-018fa0176ba320e45"
    private:
      private-one:
          id: "subnet-0153e560b3129a696"
```

```
      private-two:
          id: "subnet-0cc9c5aebe75083fd"
```

Important

If using the AZ as the identifying key, the `az` value can be omitted.

If using an arbitrary string as the identifying key, like above, either:

- `id` must be set (`az` and `cidr` optional)
- or `az` must be set (`cidr` optional)

If a user specifies a subnet by AZ without specifying CIDR and ID, a subnet in that AZ will be chosen from the VPC, arbitrarily if multiple such subnets exist.

Note

A complete subnet spec must be provided, i.e. both `public` and `private` configurations declared in the VPC spec.

Nodegroups can be restricted to named subnets via the configuration. When specifying subnets on nodegroup configuration, use the identifying key as given in the VPC spec **not** the subnet id. For example:

```
vpc:
  id: "vpc-11111"
  subnets:
    public:
      public-one:
          id: "subnet-0153e560b3129a696"
    ... # subnet spec continued

nodeGroups:
  - name: ng-1
    instanceType: m5.xlarge
    desiredCapacity: 2
    subnets:
```

```
      - public-one
```

> **Note**
>
> Only one of `subnets` or `availabilityZones` can be provided in nodegroup configuration.

When placing nodegroups inside a private subnet, `privateNetworking` must be set to `true` on the nodegroup:

```
vpc:
  id: "vpc-11111"
  subnets:
    public:
      private-one:
          id: "subnet-0153e560b3129a696"
    ... # subnet spec continued

nodeGroups:
  - name: ng-1
    instanceType: m5.xlarge
    desiredCapacity: 2
    privateNetworking: true
    subnets:
      - private-one
```

See 24-nodegroup-subnets.yaml in the eksctl GitHub repo for a full configuration example.

Cluster Access

Managing Access to the Kubernetes API Server Endpoints

By default, an EKS cluster exposes the Kubernetes API server publicly but not directly from within the VPC subnets (public=true, private=false). Traffic destined for the API server from within the VPC must first exit the VPC networks (but not Amazon's network) and then re-enter to reach the API server.

The Kubernetes API server endpoint access for a cluster can be configured for public and private access when creating the cluster using the cluster config file. Example below:

```
vpc:
  clusterEndpoints:
    publicAccess:  <true|false>
    privateAccess: <true|false>
```

There are some additional caveats when configuring Kubernetes API endpoint access:

1. EKS doesn't allow clusters without either private **or** public access enabled.
2. EKS does allow creating a configuration that allows only private access to be enabled, but eksctl doesn't support it during cluster creation as it prevents eksctl from being able to join the worker nodes to the cluster.
3. Updating a cluster to have private only Kubernetes API endpoint access means that Kubernetes commands, by default, (e.g. `kubectl`) as well as `eksctl delete cluster`, `eksctl utils write-kubeconfig`, and possibly the command `eksctl utils update-kube-proxy` must be run within the cluster VPC.
   - This requires some changes to various AWS resources. For more information, see Cluster API server endpoint.
   - You can provide `vpc.extraCIDRs` which will append additional CIDR ranges to the ControlPlaneSecurityGroup, allowing subnets outside the VPC to reach the kubernetes API endpoint. Similarly you can provide `vpc.extraIPv6CIDRs` to append IPv6 CIDR ranges as well.

The following is an example of how one could configure the Kubernetes API endpoint access using the `utils` sub-command:

```
eksctl utils update-cluster-vpc-config --cluster=<clustername> --private-access=true --
public-access=false
```

To update the setting using a `ClusterConfig` file, use:

```
eksctl utils update-cluster-vpc-config -f config.yaml --approve
```

Note that if you don't pass a flag, it will keep the current value. Once you are satisfied with the proposed changes, add the `approve` flag to make the change to the running cluster.

Restricting Access to the EKS Kubernetes Public API endpoint

The default creation of an EKS cluster exposes the Kubernetes API server publicly.

This feature only applies to the public endpoint. The API server endpoint access configuration options won't change, and you will still have the option to disable the public endpoint so your cluster is not accessible from the internet. (Source: https://github.com/aws/containers-roadmap/issues/108#issuecomment-552766489)

To restrict access to the public API endpoint to a set of CIDRs when creating a cluster, set the `publicAccessCIDRs` field:

```
vpc:
  publicAccessCIDRs: ["1.1.1.1/32", "2.2.2.0/24"]
```

To update the restrictions on an existing cluster, use:

```
eksctl utils update-cluster-vpc-config --cluster=<cluster> 1.1.1.1/32,2.2.2.0/24
```

To update the restrictions using a `ClusterConfig` file, set the new CIDRs in `vpc.publicAccessCIDRs` and run:

```
eksctl utils update-cluster-vpc-config -f config.yaml
```

> **Important**
>
> If setting `publicAccessCIDRs` and creating node-groups either `privateAccess` should be set to `true` or the nodes' IPs should be added to the `publicAccessCIDRs` list.

If nodes cannot access the cluster API endpoint due to restricted access, cluster creation will fail with `context deadline exceeded` due to the nodes being unable to access the public endpoint and failing to join the cluster.

To update both API server endpoint access and public access CIDRs for a cluster in a single command, run:

```
eksctl utils update-cluster-vpc-config --cluster=<cluster> --public-access=true --
private-access=true --public-access-cidrs=1.1.1.1/32,2.2.2.0/24
```

To update the setting using a config file:

```
vpc:
  clusterEndpoints:
    publicAccess:  <true|false>
    privateAccess: <true|false>
  publicAccessCIDRs: ["1.1.1.1/32"]
```

```
eksctl utils update-cluster-vpc-config --cluster=<cluster> -f config.yaml
```

Updating control plane subnets and security groups

This documentation explains how to modify the networking configuration of your EKS cluster's control plane after initial creation. This includes updating the control plane subnets and security groups.

Updating control plane subnets

When a cluster is created with eksctl, a set of public and private subnets are created and passed to the EKS API. EKS creates 2 to 4 cross-account elastic network interfaces (ENIs) in those subnets to enable communication between the EKS managed Kubernetes control plane and your VPC.

To update the subnets used by the EKS control plane, run:

```
eksctl utils update-cluster-vpc-config --cluster=<cluster> --control-plane-subnet-
ids=subnet-1234,subnet-5678
```

To update the setting using a config file:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: cluster
  region: us-west-2

vpc:
  controlPlaneSubnetIDs: [subnet-1234, subnet-5678]
```

```
eksctl utils update-cluster-vpc-config -f config.yaml
```

Without the `--approve` flag, eksctl only logs the proposed changes. Once you are satisfied with the proposed changes, rerun the command with the `--approve` flag.

Updating control plane security groups

To manage traffic between the control plane and worker nodes, EKS supports passing additional security groups that are applied to the cross-account network interfaces provisioned by EKS. To update the security groups for the EKS control plane, run:

```
eksctl utils update-cluster-vpc-config --cluster=<cluster> --control-plane-security-
group-ids=sg-1234,sg-5678
```

To update the setting using a config file:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: cluster
  region: us-west-2

vpc:
  controlPlaneSecurityGroupIDs: [sg-1234, sg-5678]
```

```
eksctl utils update-cluster-vpc-config -f config.yaml
```

To update both control plane subnets and security groups for a cluster, run:

```
eksctl utils update-cluster-vpc-config --cluster=<cluster> --control-plane-subnet-
ids=<> --control-plane-security-group-ids=<>
```

To update both fields using a config file:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: cluster
  region: us-west-2

vpc:
  controlPlaneSubnetIDs: [subnet-1234, subnet-5678]
```

```
  controlPlaneSecurityGroupIDs: [sg-1234, sg-5678]
```

```
eksctl utils update-cluster-vpc-config -f config.yaml
```

For a complete example, refer to cluster-subnets-sgs.yaml.

Without the `--approve` flag, eksctl only logs the proposed changes. Once you are satisfied with the proposed changes, rerun the command with the `--approve` flag.

IPv6 Support

Define IP Family

When `eksctl` creates a vpc, you can define the IP version that will be used. The following options are available to be configured:

- IPv4
- IPv6

The default value is `IPv4`.

To define it, use the following example:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: my-test
  region: us-west-2
  version: "1.21"

kubernetesNetworkConfig:
  ipFamily: IPv6 # or IPv4

addons:
  - name: vpc-cni
  - name: coredns
  - name: kube-proxy

iam:
```

```
  withOIDC: true
```

> **Note**
>
> This setting is only in the config file, not in a CLI flag.

If you use IPv6, you must configure the following requirements:

- OIDC is enabled
- managed addons are defined as shows above
- cluster version must be => 1.21
- vpc-cni addon version must be => 1.10.0
- self-managed nodegroups are not supported with IPv6 clusters
- managed nodegroups are not supported with un-owned IPv6 clusters
- `vpc.nat` and `serviceIPv4CIDR` fields are created by eksctl for ipv6 clusters and are not supported configuration options
- AutoAllocateIPv6 is not supported together with IPv6
- For IPv6 cluster, the IAM role for vpc-cni must have required IAM policies for IPv6 mode associated

Private networking can be done with IPv6 IP family as well. Please follow the instruction outlined under EKS Private Cluster.

IAM

This chapter includes information about working with AWS IAM.

Topics:

- the section called “Manage IAM users and roles”
  - Manage IAM user and role mappings to control access to an EKS cluster
  - Configure IAM identity mappings through the cluster config file or CLI commands
- the section called “IAM Roles for Service Accounts”
  - Manage fine-grained permissions for applications running on Amazon EKS that use other AWS services
  - Create and configure IAM Roles and Kubernetes Service Account pairs using eksctl
  - Enable IAM OpenID Connect Provider for an EKS cluster to enable IAM Roles for Service Accounts
- the section called “IAM permissions boundary”
  - Control the maximum permissions granted to IAM entities (users or roles) by setting a permissions boundary
- the section called “EKS Pod Identity Associations”
  - Configure IAM permissions for EKS add-ons using recommended pod identity associations
  - Enable Kubernetes applications to receive required IAM permissions to connect with AWS services outside the cluster
  - Simplify the process of automating IAM roles and service accounts across multiple EKS clusters
- the section called “IAM policies”
  - Manage IAM policies for EKS node groups, including support for various add-on policies like image builder, auto scaler, external DNS, cert manager, and more.
  - Attach custom instance roles or inline policies to node groups for additional permissions.
  - Attach specific AWS managed policies by ARN to node groups, ensuring required policies like AmazonEKSWorkerNodePolicy and AmazonEKS_CNI_Policy are included.
- the section called “Minimum IAM policies”
  - Manage Amazon EC2 resources, including load balancers, auto-scaling groups, and CloudWatch monitoring

- Create and manage AWS CloudFormation stacks
- Manage Amazon Elastic Kubernetes Service (EKS) clusters, node groups, and related resources like IAM roles and policies

Minimum IAM policies

This document describes the minimum IAM policies needed to run the main use cases of eksctl. These are the ones used to run the integration tests.

> **Note**
>
> Remember to replace <account_id> with your own.

> **Note**
>
> An AWS Managed Policy is created and administered by AWS. You cannot change the permissions defined in AWS managed policies.

AmazonEC2FullAccess (AWS Managed Policy)

View AmazonEC2FullAccess policy definition.

AWSCloudFormationFullAccess (AWS Managed Policy)

View AWSCloudFormationFullAccess policy definition.

EksAllAccess

```
# Error: No files found with UUID: 27ad3ff9-60be-4128-8b83-f8833a6e39aa
```

IamLimitedAccess

```
# Error: No files found with UUID: 5500eeb9-bf3d-498d-999b-7f8036e705a5
```

IAM permissions boundary

A permissions boundary is an advanced AWS IAM feature in which the maximum permissions that an identity-based policy can grant to an IAM entity have been set; where those entities are either users or roles. When a permissions boundary is set for an entity, that entity can only perform the actions that are allowed by both its identity-based policies and its permissions boundaries.

You can provide your permissions boundary so that all identity-based entities created by eksctl are created within that boundary. This example demonstrates how a permissions boundary can be provided to the various identity-based entities that are created by eksctl:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: cluster-17
  region: us-west-2

iam:
  withOIDC: true
  serviceRolePermissionsBoundary: "arn:aws:iam::11111:policy/entity/boundary"
  fargatePodExecutionRolePermissionsBoundary: "arn:aws:iam::11111:policy/entity/
boundary"
  serviceAccounts:
    - metadata:
        name: s3-reader
      attachPolicyARNs:
      - "arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess"
      permissionsBoundary: "arn:aws:iam::11111:policy/entity/boundary"

nodeGroups:
  - name: "ng-1"
    desiredCapacity: 1
    iam:
      instanceRolePermissionsBoundary: "arn:aws:iam::11111:policy/entity/boundary"
```

Warning

It is not possible to provide both a role ARN and a permissions boundary.

Setting the VPC CNI Permission Boundary

Please note that when you create a cluster with OIDC enabled eksctl will automatically create an `iamserviceaccount` for the VPC-CNI for security reasons. If you would like to add a permission boundary to it then you must specify the `iamserviceaccount` in your config file manually:

```
iam:
  serviceAccounts:
    - metadata:
        name: aws-node
        namespace: kube-system
      attachPolicyARNs:
      - "arn:aws:iam::<arn>:policy/AmazonEKS_CNI_Policy"
      permissionsBoundary: "arn:aws:iam::11111:policy/entity/boundary"
```

IAM policies

You can attach Instance Roles to node groups. Workloads running on the node will receive IAM permissions from the node. For more information, see IAM roles for Amazon EC2.

This page lists the pre-defined IAM policy templates available in eksctl. These templates simplify the process of granting your EKS nodes the appropriate AWS service permissions without having to manually create custom IAM policies.

Supported IAM add-on policies

Example of all supported add-on policies:

```
nodeGroups:
  - name: ng-1
    instanceType: m5.xlarge
    desiredCapacity: 1
    iam:
      withAddonPolicies:
        imageBuilder: true
        autoScaler: true
        externalDNS: true
        certManager: true
        appMesh: true
        appMeshPreview: true
```

```
        ebs: true
        fsx: true
        efs: true
        awsLoadBalancerController: true
        xRay: true
        cloudWatch: true
```

Image Builder Policy

The `imageBuilder` policy allows for full ECR (Elastic Container Registry) access. This is useful for building, for example, a CI server that needs to push images to ECR.

EBS Policy

The `ebs` policy enables the new EBS CSI (Elastic Block Store Container Storage Interface) driver.

Cert Manager Policy

The `certManager` policy enables the ability to add records to Route 53 in order to solve the DNS01 challenge. More information can be found here.

Adding a custom instance role

This example creates a nodegroup that reuses an existing IAM Instance Role from another cluster:

```
apiVersion: eksctl.io/v1alpha4
kind: ClusterConfig
metadata:
  name: test-cluster-c-1
  region: eu-north-1

nodeGroups:
  - name: ng2-private
    instanceType: m5.large
    desiredCapacity: 1
    iam:
      instanceProfileARN: "arn:aws:iam::123:instance-profile/eksctl-test-cluster-a-3-
nodegroup-ng2-private-NodeInstanceProfile-Y4YKHLNINMXC"
      instanceRoleARN: "arn:aws:iam::123:role/eksctl-test-cluster-a-3-nodegroup-
NodeInstanceRole-DNGMQTQHQHBJ"
```

Attaching inline policies

```
nodeGroups:
  - name: my-special-nodegroup
    iam:
      attachPolicy:
        Version: "2012-10-17"
        Statement:
        - Effect: Allow
          Action:
          - 's3:GetObject'
          Resource: 'arn:aws:s3:::example-bucket/*'
```

Attaching policies by ARN

```
nodeGroups:
  - name: my-special-nodegroup
    iam:
      attachPolicyARNs:
        - arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy
        - arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy
        - arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryPullOnly
        - arn:aws:iam::aws:policy/ElasticLoadBalancingFullAccess
        - arn:aws:iam::1111111111:policy/kube2iam
      withAddonPolicies:
        autoScaler: true
        imageBuilder: true
```

Warning

If a nodegroup includes the `attachPolicyARNs` it **must** also include the default node policies, like `AmazonEKSWorkerNodePolicy`, `AmazonEKS_CNI_Policy` and `AmazonEC2ContainerRegistryPullOnly` in this example.

Manage IAM users and roles

> **Note**
>
> AWS suggests migrating to the section called "EKS Pod Identity Associations" from the `aws-auth` ConfigMap.

EKS clusters use IAM users and roles to control access to the cluster. The rules are implemented in a config map

Edit ConfigMap with a CLI Command

called `aws-auth`. `eksctl` provides commands to read and edit this config map.

Get all identity mappings:

```
eksctl get iamidentitymapping --cluster <clusterName> --region=<region>
```

Get all identity mappings matching an arn:

```
eksctl get iamidentitymapping --cluster <clusterName> --region=<region> --arn
 arn:aws:iam::123456:role/testing-role
```

Create an identity mapping:

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

Delete an identity mapping:

```
eksctl delete iamidentitymapping --cluster  <clusterName> --region=<region> --arn
 arn:aws:iam::123456:role/testing
```

> **Note**
>
> Above command deletes a single mapping FIFO unless `--all` is given in which case it removes all matching. Will warn if more mappings matching this role are found.

Create an account mapping:

```
eksctl create iamidentitymapping --cluster  <clusterName> --region=<region> --account
user-account
```

Delete an account mapping:

```
eksctl delete iamidentitymapping --cluster  <clusterName> --region=<region> --account
user-account
```

Edit ConfigMap using a ClusterConfig file

The identity mappings can also be specified in ClusterConfig:

```
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: cluster-with-iamidentitymappings
  region: us-east-1

iamIdentityMappings:
  - arn: arn:aws:iam::000000000000:role/myAdminRole
    groups:
      - system:masters
    username: admin
    noDuplicateARNs: true # prevents shadowing of ARNs

  - arn: arn:aws:iam::000000000000:user/myUser
    username: myUser
    noDuplicateARNs: true # prevents shadowing of ARNs

  - serviceName: emr-containers
    namespace: emr # serviceName requires namespace

  - account: "000000000000" # account must be configured with no other options

nodeGroups:
  - name: ng-1
    instanceType: m5.large
    desiredCapacity: 1
```

```
eksctl create iamidentitymapping -f cluster-with-iamidentitymappings.yaml
```

IAM Roles for Service Accounts

Tip

`eksctl` supports configuring fine-grained permissions to EKS running apps via EKS Pod Identity Associations

Amazon EKS supports here Roles for Service Accounts (IRSA)] that allows cluster operators to map AWS IAM Roles to Kubernetes Service Accounts.

This provides fine-grained permission management for apps that run on EKS and use other AWS services. These could be apps that use S3, any other data services (RDS, MQ, STS, DynamoDB), or Kubernetes components like AWS Load Balancer controller or ExternalDNS.

You can easily create IAM Role and Service Account pairs with `eksctl`.

Note

If you used instance roles, and are considering to use IRSA instead, you shouldn't mix the two.

How it works

It works via IAM OpenID Connect Provider (OIDC) that EKS exposes, and IAM Roles must be constructed with reference to the IAM OIDC Provider (specific to a given EKS cluster), and a reference to the Kubernetes Service Account it will be bound to. Once an IAM Role is created, a service account should include the ARN of that role as an annotation (`eks.amazonaws.com/role-arn`). By default the service account will be created or updated to include the role annotation, this can be disabled using the flag `--role-only`.

Inside EKS, there is an admission controller that injects AWS session credentials into pods respectively of the roles based on the annotation on the Service Account used by the pod. The credentials will get exposed by `AWS_ROLE_ARN` & `AWS_WEB_IDENTITY_TOKEN_FILE` environment

variables. Given a recent version of AWS SDK is used (see here for details of exact version), the application will use these credentials.

In `eksctl` the name of the resource is *iamserviceaccount*, which represents an IAM Role and Service Account pair.

Usage from CLI

> **Note**
>
> IAM Roles for Service Accounts require Kubernetes version 1.13 or above.

The IAM OIDC Provider is not enabled by default, you can use the following command to enable it, or use config file (see below):

```
eksctl utils associate-iam-oidc-provider --cluster=<clusterName>
```

Once you have the IAM OIDC Provider associated with the cluster, to create a IAM role bound to a service account, run:

```
eksctl create iamserviceaccount --cluster=<clusterName> --name=<serviceAccountName> --
namespace=<serviceAccountNamespace> --attach-policy-arn=<policyARN>
```

> **Note**
>
> You can specify `--attach-policy-arn` multiple times to use more than one policy.

More specifically, you can create a service account with read-only access to S3 by running:

```
eksctl create iamserviceaccount --cluster=<clusterName> --name=s3-read-only --attach-
policy-arn=arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess
```

By default, it will be created in `default` namespace, but you can specify any other namespace, e.g.:

```
eksctl create iamserviceaccount --cluster=<clusterName> --name=s3-read-only --
namespace=s3-app --attach-policy-arn=arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess
```

> **Note**
>
> If the namespace doesn't exist already, it will be created.

If you have service account already created in the cluster (without an IAM Role), you will need to use `--override-existing-serviceaccounts` flag.

Custom tagging may also be applied to the IAM Role by specifying `--tags`:

```
eksctl create iamserviceaccount --cluster=<clusterName> --name=<serviceAccountName> --
tags "Owner=John Doe,Team=Some Team"
```

CloudFormation will generate a role name that includes a random string. If you prefer a predetermined role name you can specify `--role-name`:

```
eksctl create iamserviceaccount --cluster=<clusterName> --name=<serviceAccountName> --
role-name "custom-role-name"
```

When the service account is created and managed by some other tool, such as helm, use `--role-only` to prevent conflicts. The other tool is then responsible for maintaining the role ARN annotation. Note that `--override-existing-serviceaccounts` has no effect on `roleOnly`/`--role-only` service accounts, the role will always be created.

```
eksctl create iamserviceaccount --cluster=<clusterName> --name=<serviceAccountName> --
role-only --role-name=<customRoleName>
```

When you have an existing role which you want to use with a service account, you can provide the `--attach-role-arn` flag instead of providing the policies. To ensure the role can only be assumed by the specified service account, you should set a here relationship policy document].

```
eksctl create iamserviceaccount --cluster=<clusterName> --name=<serviceAccountName> --
attach-role-arn=<customRoleARN>
```

To update a service accounts roles permissions you can run `eksctl update iamserviceaccount`.

Note

`eksctl delete iamserviceaccount` deletes Kubernetes `ServiceAccounts` even if they were not created by `eksctl`.

Usage with config files

To manage `iamserviceaccounts` using config file, you will be looking to set `iam.withOIDC: true` and list account you want under `iam.serviceAccount`.

All of the commands support `--config-file`, you can manage *iamserviceaccounts* the same way as *nodegroups*. The `eksctl create iamserviceaccount` command supports `--include` and `--exclude` flags (see this section for more details about how these work). And the `eksctl delete iamserviceaccount` command supports `--only-missing` as well, so you can perform deletions the same way as nodegroups.

Note

IAM service accounts are scoped within a namespace, i.e. two service accounts with the same name may exist in different namespaces. Thus, to uniquely define a service account as part of `--include`, `--exclude` flags, you will need to pass the name string in the `namespace/name` format. E.g.

```
eksctl create iamserviceaccount --config-file=<path> --include backend-apps/s3-reader
```

The option to enable `wellKnownPolicies` is included for using IRSA with well-known use cases like `cluster-autoscaler` and `cert-manager`, as a shorthand for lists of policies.

Supported well-known policies and other properties of `serviceAccounts` are documented at the config schema.

You use the following config example with `eksctl create cluster`:

```
# An example of ClusterConfig with IAMServiceAccounts:
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
```

```
metadata:
  name: cluster-13
  region: us-west-2

iam:
  withOIDC: true
  serviceAccounts:
  - metadata:
      name: s3-reader
      # if no namespace is set, "default" will be used;
      # the namespace will be created if it doesn't exist already
      namespace: backend-apps
      labels: {aws-usage: "application"}
    attachPolicyARNs:
    - "arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess"
    tags:
      Owner: "John Doe"
      Team: "Some Team"
  - metadata:
      name: cache-access
      namespace: backend-apps
      labels: {aws-usage: "application"}
    attachPolicyARNs:
    - "arn:aws:iam::aws:policy/AmazonDynamoDBReadOnlyAccess"
    - "arn:aws:iam::aws:policy/AmazonElastiCacheFullAccess"
  - metadata:
      name: cluster-autoscaler
      namespace: kube-system
      labels: {aws-usage: "cluster-ops"}
    wellKnownPolicies:
      autoScaler: true
    roleName: eksctl-cluster-autoscaler-role
    roleOnly: true
  - metadata:
      name: some-app
      namespace: default
    attachRoleARN: arn:aws:iam::123:role/already-created-role-for-app
nodeGroups:
  - name: "ng-1"
    tags:
      # EC2 tags required for cluster-autoscaler auto-discovery
      k8s.io/cluster-autoscaler/enabled: "true"
      k8s.io/cluster-autoscaler/cluster-13: "owned"
```

```
    desiredCapacity: 1
```

If you create a cluster without these fields set, you can use the following commands to enable all you need:

```
eksctl utils associate-iam-oidc-provider --config-file=<path>
eksctl create iamserviceaccount --config-file=<path>
```

Further information

- Introducing Fine-grained IAM Roles For Service Accounts
- EKS User Guide - IAM Roles For Service Accounts
- Mapping IAM users and role to Kubernetes RBAC roles

EKS Pod Identity Associations

Amazon EKS has introduced a new enhanced mechanism called Pod Identity Association for cluster administrators to configure Kubernetes applications to receive IAM permissions required to connect with AWS services outside of the cluster. Pod Identity Association leverages IRSA, however, it makes it configurable directly through the EKS API, eliminating the need for using IAM API altogether.

As a result, IAM roles no longer need to reference an OIDC provider and hence won't be tied to a single cluster anymore. This means, IAM roles can now be used across multiple EKS clusters without the need to update the role trust policy each time a new cluster is created. This in turn, eliminates the need for role duplication and simplifies the process of automating IRSA altogether.

Prerequisites

Behind the scenes, the implementation of pod identity associations is running an agent as a daemonset on the worker nodes. To run the pre-requisite agent on the cluster, EKS provides a new add-on called EKS Pod Identity Agent. Therefore, creating pod identity associations (in general, and with `eksctl`) requires the `eks-pod-identity-agent` addon pre-installed on the cluster. This addon can be created using `eksctl` in the same fashion any other supported addon is.

```
eksctl create addon --cluster my-cluster --name eks-pod-identity-agent
```

Additionally, if using a pre-existing IAM role when creating a pod identity association, you must configure the role to trust the newly introduced EKS service principal (`pods.eks.amazonaws.com`). An example IAM trust policy can be found below:

```
# Error: No files found with UUID: 44d1085a-03ca-431a-9774-b786a9774200
```

If instead you do not provide the ARN of an existing role to the create command, `eksctl` will create one behind the scenes and configure the above trust policy.

Creating Pod Identity Associations

For manipulating pod identity associations, `eksctl` has added a new field under `iam.podIdentityAssociations`, e.g.

```
iam:
  podIdentityAssociations:
  - namespace: <string> #required
    serviceAccountName: <string> #required
    createServiceAccount: true #optional, default is false
    roleARN: <string> #required if none of permissionPolicyARNs, permissionPolicy and
 wellKnownPolicies is specified. Also, cannot be used together with any of the three
 other referenced fields.
    roleName: <string> #optional, generated automatically if not provided, ignored if
 roleARN is provided
    permissionPolicy: {} #optional
    permissionPolicyARNs: [] #optional
    wellKnownPolicies: {} #optional
    permissionsBoundaryARN: <string> #optional
    tags: {} #optional
```

For a complete example, refer to pod-identity-associations.yaml.

> **Note**
>
> Apart from `permissionPolicy` which is used as an inline policy document, all other fields have a CLI flag counterpart.

Creating pod identity associations can be achieved in the following ways. During cluster creation, by specifying the desired pod identity associations as part of the config file and running:

```
eksctl create cluster -f config.yaml
```

Post cluster creation, using either a config file e.g.

```
eksctl create podidentityassociation -f config.yaml
```

OR using CLI flags e.g.

```
eksctl create podidentityassociation \
    --cluster my-cluster \
    --namespace default \
    --service-account-name s3-reader \
    --permission-policy-arns="arn:aws:iam::111122223333:policy/permission-policy-1,
 arn:aws:iam::111122223333:policy/permission-policy-2" \
    --well-known-policies="autoScaler,externalDNS" \
    --permissions-boundary-arn arn:aws:iam::111122223333:policy/permissions-boundary
```

Note

Only a single IAM role can be associated with a service account at a time. Therefore, trying to create a second pod identity association for the same service account will result in an error.

Fetching Pod Identity Associations

To retrieve all pod identity associations for a certain cluster, run one of the following commands:

```
eksctl get podidentityassociation -f config.yaml
```

OR

```
eksctl get podidentityassociation --cluster my-cluster
```

Additionally, to retrieve only the pod identity associations within a given namespace, use the `--namespace` flag, e.g.

```
eksctl get podidentityassociation --cluster my-cluster --namespace default
```

Finally, to retrieve a single association, corresponding to a certain K8s service account, also include the `--service-account-name` to the command above, i.e.

```
eksctl get podidentityassociation --cluster my-cluster --namespace default --service-
account-name s3-reader
```

Updating Pod Identity Associations

To update the IAM role of one or more pod identity associations, either pass the new `roleARN(s)` to the config file e.g.

```
iam:
  podIdentityAssociations:
    - namespace: default
      serviceAccountName: s3-reader
      roleARN: new-role-arn-1
    - namespace: dev
      serviceAccountName: app-cache-access
      roleARN: new-role-arn-2
```

and run:

```
eksctl update podidentityassociation -f config.yaml
```

OR (to update a single association) pass the new `--role-arn` via CLI flags:

```
eksctl update podidentityassociation --cluster my-cluster --namespace default --
service-account-name s3-reader --role-arn new-role-arn
```

Deleting Pod Identity Associations

To delete one or more pod identity associations, either pass `namespace(s)` and `serviceAccountName(s)` to the config file e.g.

```
iam:
  podIdentityAssociations:
    - namespace: default
      serviceAccountName: s3-reader
    - namespace: dev
      serviceAccountName: app-cache-access
```

and run:

```
eksctl delete podidentityassociation -f config.yaml
```

OR (to delete a single association) pass the `--namespace` and `--service-account-name` via CLI flags:

```
eksctl delete podidentityassociation --cluster my-cluster --namespace default --
service-account-name s3-reader
```

EKS Add-ons support for pod identity associations

EKS Add-ons also support receiving IAM permissions via EKS Pod Identity Associations. The config file exposes three fields that allow configuring these: `addon.podIdentityAssociations`, `addonsConfig.autoApplyPodIdentityAssociations` and `addon.useDefaultPodIdentityAssociations`. You can either explicitly configure the desired pod identity associations, using `addon.podIdentityAssociations`, or have `eksctl` automatically resolve (and apply) the recommended pod identity configuration, using either `addonsConfig.autoApplyPodIdentityAssociations` or `addon.useDefaultPodIdentityAssociations`.

> **Note**
>
> Not all EKS Add-ons will support pod identity associations at launch. For this case, required IAM permissions shall continue to be provided using IRSA settings.

Creating addons with IAM permissions

When creating an addon that requires IAM permissions, `eksctl` will first check if either pod identity associations or IRSA settings are being explicitly configured as part of the config file, and if so, use one of those to configure the permissions for the addon. e.g.

```
addons:
- name: vpc-cni
  podIdentityAssociations:
  - serviceAccountName: aws-node
    permissionPolicyARNs: ["arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy"]
```

and run

```
eksctl create addon -f config.yaml
2024-05-13 15:38:58 [#] pod identity associations are set for "vpc-cni" addon; will use
 these to configure required IAM permissions
```

> **Note**
>
> Setting both pod identities and IRSA at the same time is not allowed, and will result in a validation error.

For EKS Add-ons that support pod identities, `eksctl` offers the option to automatically configure any recommended IAM permissions, on addon creation. This can be achieved by simply setting `addonsConfig.autoApplyPodIdentityAssociations: true` in the config file. e.g.

```
addonsConfig:
  autoApplyPodIdentityAssociations: true
# bear in mind that if either pod identity or IRSA configuration is explicitly set in
 the config file,
# or if the addon does not support pod identities,
# addonsConfig.autoApplyPodIdentityAssociations won't have any effect.
addons:
- name: vpc-cni
```

and run

```
eksctl create addon -f config.yaml
2024-05-13 15:38:58 [#] "addonsConfig.autoApplyPodIdentityAssociations" is set to true;
 will lookup recommended pod identity configuration for "vpc-cni" addon
```

Equivalently, the same can be done via CLI flags e.g.

```
eksctl create addon --cluster my-cluster --name vpc-cni --auto-apply-pod-identity-
associations
```

To migrate an existing addon to use pod identity with the recommended IAM policies, use

```
addons:
```

```
- name: vpc-cni
  useDefaultPodIdentityAssociations: true
```

```
eksctl update addon -f config.yaml
```

Updating addons with IAM permissions

When updating an addon, specifying `addon.PodIdentityAssociations` will represent the single source of truth for the state that the addon shall have, after the update operation is completed. Behind the scenes, different types of operations are performed in order to achieve the desired state i.e.

- create pod identites that are present in the config file, but missing on the cluster
- delete existing pod identites that were removed from the config file, together with any associated IAM resources
- update existing pod identities that are also present in the config file, and for which the set of IAM permissions has changed

> **Note**
>
> The lifecycle of pod identity associations owned by EKS Add-ons is directly handled by the EKS Addons API.

You can't use `eksctl update podidentityassociation` (to update IAM permissions) or `eksctl delete podidentityassociations` (to remove the association) for associations used with an Amazon EKS Add-on. Instead, `eksctl update addon` or `eksctl delete addon` shall be used.

Let's see an example for the above, starting by analyzing the initial pod identity config for the addon:

```
eksctl get podidentityassociation --cluster my-cluster --namespace opentelemetry-
operator-system --output json
[
    {
        ...
        "ServiceAccountName": "adot-col-prom-metrics",
```

```
        "RoleARN": "arn:aws:iam::111122223333:role/eksctl-my-cluster-addon-adot-
podident-Role1-JwrGA4mn1Ny8",
        # OwnerARN is populated when the pod identity lifecycle is handled by the EKS
 Addons API
        "OwnerARN": "arn:aws:eks:us-west-2:111122223333:addon/my-cluster/adot/
b2c7bb45-4090-bf34-ec78-a2298b8643f6"
    },
    {
        ...
        "ServiceAccountName": "adot-col-otlp-ingest",
        "RoleARN": "arn:aws:iam::111122223333:role/eksctl-my-cluster-addon-adot-
podident-Role1-Xc7qVg5fgCqr",
        "OwnerARN": "arn:aws:eks:us-west-2:111122223333:addon/my-cluster/adot/
b2c7bb45-4090-bf34-ec78-a2298b8643f6"
    }
]
```

Now use the below configuration:

```
addons:
- name: adot
  podIdentityAssociations:

  # For the first association, the permissions policy of the role will be updated
  - serviceAccountName: adot-col-prom-metrics
    permissionPolicyARNs:
    #- arn:aws:iam::aws:policy/AmazonPrometheusRemoteWriteAccess
    - arn:aws:iam::aws:policy/CloudWatchAgentServerPolicy

  # The second association will be deleted, as it's been removed from the config file
  #- serviceAccountName: adot-col-otlp-ingest
  #  permissionPolicyARNs:
  #  - arn:aws:iam::aws:policy/AWSXrayWriteOnlyAccess

  # The third association will be created, as it's been added to the config file
  - serviceAccountName: adot-col-container-logs
    permissionPolicyARNs:
    - arn:aws:iam::aws:policy/CloudWatchAgentServerPolicy
```

and run

```
eksctl update addon -f config.yaml
...
```

```
# updating the permission policy for the first association
2024-05-14 13:27:43 [#]  updating IAM resources stack "eksctl-my-cluster-addon-
adot-podidentityrole-adot-col-prom-metrics" for pod identity association "a-
reaxk2uz1iknwazwj"
2024-05-14 13:27:44 [#]  waiting for CloudFormation changeset "eksctl-opentelemetry-
operator-system-adot-col-prom-metrics-update-1715682463" for stack "eksctl-my-cluster-
addon-adot-podidentityrole-adot-col-prom-metrics"
2024-05-14 13:28:47 [#]  waiting for CloudFormation stack "eksctl-my-cluster-addon-
adot-podidentityrole-adot-col-prom-metrics"
2024-05-14 13:28:47 [#]  updated IAM resources stack "eksctl-my-cluster-addon-adot-
podidentityrole-adot-col-prom-metrics" for "a-reaxk2uz1iknwazwj"
# creating the IAM role for the second association
2024-05-14 13:28:48 [#]  deploying stack "eksctl-my-cluster-addon-adot-podidentityrole-
adot-col-container-logs"
2024-05-14 13:28:48 [#]  waiting for CloudFormation stack "eksctl-my-cluster-addon-
adot-podidentityrole-adot-col-container-logs"
2024-05-14 13:29:19 [#]  waiting for CloudFormation stack "eksctl-my-cluster-addon-
adot-podidentityrole-adot-col-container-logs"
# updating the addon, which handles the pod identity config changes behind the scenes
2024-05-14 13:29:19 [#]  updating addon
# deleting the IAM role for the third association
2024-05-14 13:29:19 [#]  deleting IAM resources for pod identity service account adot-
col-otlp-ingest
2024-05-14 13:29:20 [#]  will delete stack "eksctl-my-cluster-addon-adot-
podidentityrole-adot-col-otlp-ingest"
2024-05-14 13:29:20 [#]  waiting for stack "eksctl-my-cluster-addon-adot-
podidentityrole-adot-col-otlp-ingest" to get deleted
2024-05-14 13:29:51 [#]  waiting for CloudFormation stack "eksctl-my-cluster-addon-
adot-podidentityrole-adot-col-otlp-ingest"
2024-05-14 13:29:51 [#]  deleted IAM resources for addon adot
```

now check that pod identity config was updated correctly

```
eksctl get podidentityassociation --cluster my-cluster --output json
[
    {
        ...
        "ServiceAccountName": "adot-col-prom-metrics",
        "RoleARN": "arn:aws:iam::111122223333:role/eksctl-my-cluster-addon-adot-
podident-Role1-nQAlp0KktS2A",
        "OwnerARN": "arn:aws:eks:us-west-2:111122223333:addon/my-cluster/
adot/1ec7bb63-8c4e-ca0a-f947-310c4b55052e"
    },
```

```
    {
        ...
        "ServiceAccountName": "adot-col-otlp-ingest",
        "RoleARN": "arn:aws:iam::111122223333:role/eksctl-my-cluster-addon-adot-
podident-Role1-1k1XhAdziGzX",
        "OwnerARN": "arn:aws:eks:us-west-2:111122223333:addon/my-cluster/
adot/1ec7bb63-8c4e-ca0a-f947-310c4b55052e"
    }
]
```

To remove all pod identity associations from an addon, `addon.PodIdentityAssociations` must be explicitly set to `[]`, e.g.

```
addons:
- name: vpc-cni
  # omitting the `podIdentityAssociations` field from the config file,
  # instead of explicitly setting it to [], will result in a validation error
  podIdentityAssociations: []
```

and run

```
eksctl update addon -f config.yaml
```

Deleting addons with IAM permissions

Deleting an addon will also remove all pod identities associated with the addon. Deleting the cluster will achieve the same effect, for all addons. Any IAM roles for pod identities, created by `eksctl`, will be deleted as-well.

Migrating existing iamserviceaccounts and addons to pod identity associations

There is an `eksctl` utils command for migrating existing IAM Roles for service accounts to pod identity associations, i.e.

```
eksctl utils migrate-to-pod-identity --cluster my-cluster --approve
```

Behind the scenes, the command will apply the following steps:

- install the `eks-pod-identity-agent` addon if not already active on the cluster

- identify all IAM Roles that are associated with iamserviceaccounts
- identify all IAM Roles that are associated with EKS addons that support pod identity associations
- update the IAM trust policy of all identified roles, with an additional trusted entity, pointing to the new EKS Service principal (and, optionally, remove exising OIDC provider trust relationship)
- create pod identity associations for filtered roles associated with iamserviceaccounts
- update EKS addons with pod identities (EKS API will create the pod identities behind the scenes)

Running the command without the `--approve` flag will only output a plan consisting of a set of tasks reflecting the steps above, e.g.

```
[#]  (plan) would migrate 2 iamserviceaccount(s) and 2 addon(s) to pod identity
 association(s) by executing the following tasks
[#]  (plan)

3 sequential tasks: { install eks-pod-identity-agent addon,
    ## tasks for migrating the addons
    2 parallel sub-tasks: {
        2 sequential sub-tasks: {
            update trust policy for owned role "eksctl-my-cluster--Role1-DDuMLoeZ8weD",
            migrate addon aws-ebs-csi-driver to pod identity,
        },
        2 sequential sub-tasks: {
            update trust policy for owned role "eksctl-my-cluster--Role1-xYiPFOVp1aeI",
            migrate addon vpc-cni to pod identity,
        },
    },
    ## tasks for migrating the iamserviceaccounts
    2 parallel sub-tasks: {
        2 sequential sub-tasks: {
            update trust policy for owned role "eksctl-my-cluster--Role1-QLXqHcq9O1AR",
            create pod identity association for service account "default/sa1",
        },
        2 sequential sub-tasks: {
            update trust policy for unowned role "Unowned-Role1",
            create pod identity association for service account "default/sa2",
        },
    }
}
[#]  all tasks were skipped
[!]  no changes were applied, run again with '--approve' to apply the changes
```

The existing OIDC provider trust relationship is always being removed from IAM Roles associated with EKS Add-ons. Additionally, to remove the existing OIDC provider trust relationship from IAM Roles associated with iamserviceaccounts, run the command with `--remove-oidc-provider-trust-relationship` flag, e.g.

```
eksctl utils migrate-to-pod-identity --cluster my-cluster --approve --remove-oidc-
provider-trust-relationship
```

Cross Account Pod Identity Support

eksctl supports EKS Pod Identity cross-account access. This feature allows pods running in your EKS cluster to access AWS resources in a different AWS account.

Usage

To create a pod identity association with cross-account access, first set up IAM Roles and Policies allowing access from a source AWS account (with the cluster) to a target AWS account (with the resources the cluster can access). For an example of this, see "Amazon EKS Pod Identity streamlines cross account access."

Once an IAM Role is configured in each account, use eksctl to create the pod identity associations:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  # The cluster name and service account name should match the target
  # account policy's trust relationship.
  name: my-cluster
  region: us-west-2
  version: "1.32"

addons:
  - name: vpc-cni
  - name: coredns
  - name: kube-proxy
  - name: eks-pod-identity-agent

iam:
  podIdentityAssociations:
  - namespace: default
    serviceAccountName: demo-app-sa
```

```
    createServiceAccount: true
    # The source role in the same account as the cluster
    roleARN: arn:aws:iam::1111111111:role/account-a-role
    # The target role in a different account
    targetRoleARN: arn:aws:iam::2222222222:role/account-b-role
    # Optional: Disable session tags
    disableSessionTags: false

managedNodeGroups:
  - name: my-cluster
    instanceType: m6a.large
    privateNetworking: true
    minSize: 2
    desiredCapacity: 2
    maxSize: 3
```

Further references

Official AWS Userdocs for EKS Add-ons support for pod identities

Official AWS Blog Post on Pod Identity Associations

Official AWS userdocs for Pod Identity Associations

Deployment options

This chapter covers using eksctl to manage EKS clusters deployed to alternate environments.

For the most accurate information about EKS deployment options, see Deploy Amazon EKS clusters across cloud and on-premises environments in the *EKS User Guide*.

Topics:

- the section called “EKS Anywhere”
  - Use eksctl with Amazon EKS Anywhere clusters.
  - Amazon EKS Anywhere is container management software built by AWS that makes it easier to run and manage Kubernetes on-premises and at the edge.
- the section called “AWS Outposts Support”
  - Use eksctl with EKS clusters on AWS Outposts.
  - AWS Outposts is a family of fully managed solutions delivering AWS infrastructure and services to virtually any on-premises or edge location for a truly consistent hybrid experience.
  - AWS Outposts support in eksctl lets you create local clusters with the entire Kubernetes cluster, including the EKS control plane and worker nodes, running locally on AWS Outposts.
- the section called “EKS Hybrid Nodes”
  - Run on-premises and edge applications on customer-managed infrastructure with the same Amazon EKS clusters, features, and tools you use in AWS.

EKS Anywhere

`eksctl` provides access to AWS' feature called `EKS Anywhere` with the sub command `eksctl anywhere`. This requires the `eksctl-anywhere` binary present on `PATH`. Please follow the instruction outlined here Install eksctl-anywhere to install it.

Once done, execute anywhere commands by running:

```
eksctl anywhere version
v0.5.0
```

For more information about EKS Anywhere, please visit EKS Anywhere Website.

AWS Outposts Support

> **Warning**
>
> EKS Managed Nodegroups are not supported on Outposts.

Extending existing clusters to AWS Outposts

You can extend an existing EKS cluster running in an AWS region to AWS Outposts by setting `nodeGroup.outpostARN` for new nodegroups to create nodegroups on Outposts, as in:

```
# extended-cluster.yaml
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: existing-cluster
  region: us-west-2

nodeGroups:
  # Nodegroup will be created in an AWS region.
  - name: ng

  # Nodegroup will be created on the specified Outpost.
  - name: outpost-ng
    privateNetworking: true
    outpostARN: "arn:aws:outposts:us-west-2:1234:outpost/op-1234"
```

```
eksctl create nodegroup -f extended-cluster.yaml
```

In this setup, the EKS control plane runs in an AWS region while nodegroups with `outpostARN` set run on the specified Outpost. When a nodegroup is being created on Outposts for the first time, eksctl extends the VPC by creating subnets on the specified Outpost. These subnets are used to create nodegroups that have `outpostARN` set.

Customers with a pre-existing VPC are required to create the subnets on Outposts and pass them in `nodeGroup.subnets`, as in:

```
# extended-cluster-vpc.yaml
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: extended-cluster-vpc
  region: us-west-2

vpc:
  id: vpc-1234
    subnets:
      private:
        outpost-subnet-1:
          id: subnet-1234

nodeGroups:
  # Nodegroup will be created in an AWS region.
  - name: ng

  # Nodegroup will be created on the specified Outpost.
  - name: outpost-ng
    privateNetworking: true
    # Subnet IDs for subnets created on Outpost.
    subnets: [subnet-5678]
    outpostARN: "arn:aws:outposts:us-west-2:1234:outpost/op-1234"
```

Creating a local cluster on AWS Outposts

Note

Local clusters support Outpost racks only.

Note

Only Amazon Linux 2 is supported for nodegroups when the control plane is on Outposts. Only EBS gp2 volume types are supported for nodegroups on Outposts.

AWS Outposts support in eksctl lets you create local clusters with the entire Kubernetes cluster, including the EKS control plane and worker nodes, running locally on AWS Outposts. Customers can either create a local cluster with both the EKS control plane and worker nodes running locally on AWS Outposts, or they can extend an existing EKS cluster running in an AWS region to AWS Outposts by creating worker nodes on Outposts.

To create the EKS control plane and nodegroups on AWS Outposts, set `outpost.controlPlaneOutpostARN` to the Outpost ARN, as in:

```
# outpost.yaml
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: outpost
  region: us-west-2

outpost:
  # Required.
  controlPlaneOutpostARN: "arn:aws:outposts:us-west-2:1234:outpost/op-1234"
  # Optional, defaults to the smallest available instance type on the Outpost.
  controlPlaneInstanceType: m5d.large
```

```
eksctl create cluster -f outpost.yaml
```

This instructs eksctl to create the EKS control plane and subnets on the specified Outpost. Since an Outposts rack exists in a single availability zone, eksctl creates only one public and private subnet. eksctl does not associate the created VPC with a local gateway and, as such, eksctl will lack connectivity to the API server and will be unable to create nodegroups. Therefore, if the `ClusterConfig` contains any nodegroups during cluster creation, the command must be run with `--without-nodegroup`, as in:

```
eksctl create cluster -f outpost.yaml --without-nodegroup
```

It is the customer's responsibility to associate the eksctl-created VPC with the local gateway after cluster creation to enable connectivity to the API server. After this step, nodegroups can be created using `eksctl create nodegroup`.

You can optionally specify the instance type for the control plane nodes in `outpost.controlPlaneInstanceType` or for the nodegroups in `nodeGroup.instanceType`, but the instance type must exist on Outpost or eksctl will return an error. By default, eksctl attempts to choose the smallest available instance type on Outpost for the control plane nodes and nodegroups.

When the control plane is on Outposts, nodegroups are created on that Outpost. You can optionally specify the Outpost ARN for the nodegroup in `nodeGroup.outpostARN` but it must match the control plane's Outpost ARN.

```
# outpost-fully-private.yaml
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: outpost-fully-private
  region: us-west-2

privateCluster:
  enabled: true

outpost:
  # Required.
  controlPlaneOutpostARN: "arn:aws:outposts:us-west-2:1234:outpost/op-1234"
  # Optional, defaults to the smallest available instance type on the Outpost.
  controlPlaneInstanceType: m5d.large
```

```
# outpost.yaml
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: outpost
  region: us-west-2

outpost:
  # Required.
  controlPlaneOutpostARN: "arn:aws:outposts:us-west-2:1234:outpost/op-1234"
  # Optional, defaults to the smallest available instance type on the Outpost.
  controlPlaneInstanceType: m5d.large
```

```
  controlPlanePlacement:
    groupName: placement-group-name
```

Existing VPC

Customers with an existing VPC can create local clusters on AWS Outposts by specifying the subnet configuration in `vpc.subnets`, as in:

```
# outpost-existing-vpc.yaml
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: outpost
  region: us-west-2

vpc:
  id: vpc-1234
  subnets:
    private:
      outpost-subnet-1:
        id: subnet-1234

nodeGroups:
  - name: outpost-ng
    privateNetworking: true

outpost:
    # Required.
    controlPlaneOutpostARN: "arn:aws:outposts:us-west-2:1234:outpost/op-1234"
    # Optional, defaults to the smallest available instance type on the Outpost.
    controlPlaneInstanceType: m5d.large
```

```
eksctl create cluster -f outpost-existing-vpc.yaml
```

The subnets must exist on the Outpost specified in `outpost.controlPlaneOutpostARN` or eksctl will return an error. You can also specify nodegroups during cluster creation if you have access to the local gateway for the subnet, or have connectivity to VPC resources.

Features unsupported on local clusters

- Addons
- IAM Roles for Service Accounts
- IPv6
- Identity Providers
- Fargate
- KMS Encryption
- Local Zones
- Karpenter
- Instance Selector
- Availability Zones cannot be specified as it defaults to the Outpost availability zone.
- `vpc.publicAccessCIDRs` and `vpc.autoAllocateIPv6` are not supported.
- Public endpoint access to the API server is not supported as a local cluster can only be created with private-only endpoint access.

Further information

- Amazon EKS on AWS Outposts
- Local clusters for Amazon EKS on AWS Outposts
- Creating local clusters
- Launching self-managed Amazon Linux nodes on an Outpost

Security

`eksctl` provides some options that can improve the security of your EKS cluster.

`withOIDC`

Enable `withOIDC` to automatically create an IRSA for the amazon CNI plugin and limit permissions granted to nodes in your cluster, instead granting the necessary permissions only to the CNI service account.

The background is described in this AWS documentation.

`disablePodIMDS`

For managed and unmanaged nodegroups, `disablePodIMDS` option is available prevents all non host networking pods running in this nodegroup from making IMDS requests.

> **Note**
>
> This can not be used together with `withAddonPolicies`.

KMS Envelope Encryption for EKS clusters

> **Note**
>
> Amazon Elastic Kubernetes Service (Amazon EKS) provides default envelope encryption for all Kubernetes API data in EKS clusters running Kubernetes version 1.28 or higher. For more information, see Default envelope encryption for all Kubernetes API Data in the *EKS User Guide*.

EKS supports using AWS KMS keys to provide envelope encryption of Kubernetes secrets stored in EKS. Envelope encryption adds an additional, customer-managed layer of encryption for application secrets or user data that is stored within a Kubernetes cluster.

Previously, Amazon EKS supported enabling envelope encryption using KMS keys only during cluster creation. Now, you can enable envelope encryption for Amazon EKS clusters at any time.

Read more about Using EKS encryption provider support for defense-in-depth post on the AWS containers blog.

Creating a cluster with KMS encryption enabled

```
# kms-cluster.yaml
# A cluster with KMS encryption enabled
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: kms-cluster
  region: us-west-2

managedNodeGroups:
- name: ng
# more config

secretsEncryption:
  # KMS key used for envelope encryption of Kubernetes secrets
  keyARN: arn:aws:kms:us-west-2:<account>:key/<key>
```

```
eksctl create cluster -f kms-cluster.yaml
```

Enabling KMS encryption on an existing cluster

To enable KMS encryption on a cluster that doesn't already have it enabled, run

```
eksctl utils enable-secrets-encryption -f kms-cluster.yaml
```

or without a config file:

```
eksctl utils enable-secrets-encryption --cluster=kms-cluster --key-arn=arn:aws:kms:us-
west-2:<account>:key/<key> --region=<region>
```

In addition to enabling KMS encryption on the EKS cluster, eksctl also re-encrypts all existing Kubernetes secrets using the new KMS key by updating them with the annotation `eksctl.io/kms-encryption-timestamp`. This behaviour can be disabled by passing `--encrypt-existing-secrets=false`, as in:

```
eksctl utils enable-secrets-encryption --cluster=kms-cluster --key-arn=arn:aws:kms:us-
west-2:<account>:key/<key> --encrypt-existing-secrets=false --region=<region>
```

If a cluster already has KMS encryption enabled, eksctl will proceed to re-encrypting all existing secrets.

> **Note**
>
> Once KMS encryption is enabled, it cannot be disabled or updated to use a different KMS key.

Troubleshooting

This topic includes instructions on how to resolve common errors with Eksctl.

Failed stack creation

You can use the `--cfn-disable-rollback` flag to stop Cloudformation from rolling back failed stacks to make debugging easier.

subnet ID "subnet-11111111" is not the same as "subnet-22222222"

Given a config file specifying subnets for a VPC like the following:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: test
  region: us-east-1

vpc:
  subnets:
    public:
      us-east-1a: {id: subnet-11111111}
      us-east-1b: {id: subnet-22222222}
    private:
      us-east-1a: {id: subnet-33333333}
      us-east-1b: {id: subnet-44444444}

nodeGroups: []
```

An error `subnet ID "subnet-11111111" is not the same as "subnet-22222222"` means that the subnets specified are not placed in the right Availability zone. Check in the AWS console which is the right subnet ID for each Availability Zone.

In this example, the correct configuration for the VPC would be:

```
vpc:
```

```
  subnets:
    public:
      us-east-1a: {id: subnet-22222222}
      us-east-1b: {id: subnet-11111111}
    private:
      us-east-1a: {id: subnet-33333333}
      us-east-1b: {id: subnet-44444444}
```

Deletion issues

If your delete does not work, or you forget to add `--wait` on the delete, you may need to go to use amazon's other tools to delete the cloudformation stacks. This can be accomplished via the gui or with the aws cli.

kubectl logs and kubectl run fails with Authorization Error

If your nodes are deployed in a private subnet and `kubectl logs` or `kubectl run` fail with an error such as the following:

```
Error attaching, falling back to logs: unable to upgrade connection: Authorization
 error (user=kube-apiserver-kubelet-client, verb=create, resource=nodes,
 subresource=proxy)
```

```
Error from server (InternalError): Internal error occurred: Authorization error
 (user=kube-apiserver-kubelet-client, verb=get, resource=nodes, subresource=proxy)
```

Then you might need to set enableDnsHostnames. More details can be found in this issue.

Announcements

This topic covers past announcements of new Eksctl features.

Managed Nodegroups Default

As of eksctl v0.58.0, eksctl creates managed nodegroups by default when a `ClusterConfig` file isn't specified for `eksctl create cluster` and `eksctl create nodegroup`. To create a self-managed nodegroup, pass `--managed=false`. This may break scripts not using a config file if a feature not supported in managed nodegroups, e.g., Windows nodegroups, is being used. To fix this, pass `--managed=false`, or specify your nodegroup config in a `ClusterConfig` file using the `nodeGroups` field which creates a self-managed nodegroup.

Nodegroup Bootstrap Override For Custom AMIs

This change was announced in the issue Breaking: overrideBootstrapCommand soon…. Now, it has come to pass in this PR. Please read the attached issue carefully about why we decided to move away from supporting custom AMIs without bootstrap scripts or with partial bootstrap scripts.

We still provide a helper! Migrating hopefully is not that painful. `eksctl` still provides a script, which when sourced, will export a couple of helpful environment properties and settings. This script is located here.

The following environment properties will be at your disposal:

```
API_SERVER_URL
B64_CLUSTER_CA
INSTANCE_ID
INSTANCE_LIFECYCLE
CLUSTER_DNS
NODE_TAINTS
MAX_PODS
NODE_LABELS
CLUSTER_NAME
CONTAINER_RUNTIME # default is docker
KUBELET_EXTRA_ARGS # for details, look at the script
```

The minimum that needs to be used when overriding so `eksctl` doesn't fail, is labels! `eksctl` relies on a specific set of labels to be on the node, so it can find them. When defining the override, please provide this **bare minimum** override command:

```
    overrideBootstrapCommand: |
      #!/bin/bash

      source /var/lib/cloud/scripts/eksctl/bootstrap.helper.sh

      # Note "--node-labels=${NODE_LABELS}" needs the above helper sourced to work,
 otherwise will have to be defined manually.
      /etc/eks/bootstrap.sh ${CLUSTER_NAME} --container-runtime containerd --kubelet-
extra-args "--node-labels=${NODE_LABELS}"
```

For nodegroups that have no outbound internet access, you'll need to supply `--apiserver-endpoint` and `--b64-cluster-ca` to the bootstrap script as follows:

```
    overrideBootstrapCommand: |
      #!/bin/bash

      source /var/lib/cloud/scripts/eksctl/bootstrap.helper.sh

      # Note "--node-labels=${NODE_LABELS}" needs the above helper sourced to work,
 otherwise will have to be defined manually.
      /etc/eks/bootstrap.sh ${CLUSTER_NAME} --container-runtime containerd --kubelet-
extra-args "--node-labels=${NODE_LABELS}" \
        --apiserver-endpoint ${API_SERVER_URL} --b64-cluster-ca ${B64_CLUSTER_CA}
```

Note the `*--node-labels*` setting. If this is not defined, the node will join the cluster, but `eksctl` will ultimately time out on the last step when it's waiting for the nodes to be `Ready`. It's doing a Kubernetes lookup for nodes that have the label `alpha.eksctl.io/nodegroup-name=<cluster-name>`. This is only true for unmanaged nodegroups. For managed it's using a different label.

If, at all, it's possible to switch to managed nodegroups to avoid this overhead, the time has come now to do that. Makes all the overriding a lot easier.