Terraform:

Terraform can provision infrastructure across public cloud providers such as Amazon Web Services (AWS), Azure, Google Cloud, and DigitalOcean, as well as private cloud and virtualization platforms such as OpenStack and VMWare.

Terraform Basics:

Provider:

A provider is responsible for understanding API interactions and exposing resources. Providers generally are an IaaS (e.g. Alibaba Cloud, AWS, GCP, Microsoft Azure, OpenStack), PaaS (e.g. Heroku), or SaaS services (e.g. Terraform Cloud, DNSimple, Cloudflare)

init

Cmd: *terraform init*

terraform init tells Terraform to scan the code, figure out which providers you're using, and download the code for them.

Plan

Cmd: *terraform plan*

The plan command lets you see what Terraform will do before actually making any changes. This is a great way to sanity check your code before unleashing it onto the world

Apply

Cmd: *terraform apply*

The *apply* command applies the actual changes in the terraform script.

Destroy

Cmd: *terraform destroy*

To cleanup the resources created by apply command use destroy command

Don't Repeat Yourself (DRY) **principle:**

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system. If you have the port number in two places, it's easy to update it in one place but forget to make the same change in the other place.

To allow you to make your code more DRY and more configurable, Terraform allows you to define ***input variables***

Data Source

A data source represents a piece of read-only information that is fetched from the provider (in this case, AWS) every time you run Terraform. Adding a data source to your Terraform configurations does not create anything new; it's just a way to query the provider's APIs for data and to make that data available to the rest of your Terraform code

For instance,

```
data "aws_vpc" "default" {
  default = true
}
```

You can get the id of the vpc from aws_vpc data source, you would use the following

```
data.aws_vpc.default.id
```

Terraform state

Every time you run Terraform, it records information about what infrastructure it created in a *Terraform state file*.

It is recommended to have terraform state files in the remote backend to avoid issues like manual errors, locking and secrets. Remote backends allow you to store the state file in a remote, shared store. We have used S3 to store the terraform state files by enabling encryption and accidental delete. It is highly recommended that you enable Bucket Versioning on the S3 bucket to allow for state recovery in the case of accidental deletions and human error.

It is also recommended to isolate the state files at environment and component level like we do in this example

To run terraform apply, Terraform will automatically acquire a lock; if someone else is already running apply, they will already have the lock, and you will have to wait.

Kubernetes

Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications.

A kubernetes cluster consists of two primary components

1. Control Plane
2. Worker Nodes that are registered with the control plane

Control plane

The **control plane** is responsible for managing the entire cluster. It consists of one or more master nodes (typically 3 master nodes for high availability), where each master node runs several components:

Components of the Control Plane

The Control Plane is what controls and makes the whole cluster function. To refresh your memory, the components that make up the Control Plane are

1. The etcd distributed persistent storage
2. The API server
3. The Scheduler
4. The Controller Manager

These components store and manage the state of the cluster, but they aren't what runs the application containers.

Kubernetes API Server

The **Kubernetes API Server** is the endpoint you're talking to when you use the Kubernetes API (e.g., via kubectl).

Scheduler

The **scheduler** is responsible for figuring out which of the worker nodes to use to run your container(s). It tries to pick the "best" worker node based on a number of factors, such as high availability (try to run copies of the same container on different nodes so a failure in one node doesn't take them all down), resource utilization (try to run the

container on the least utilized node), container requirements (try to find nodes that meet the container's requirements in terms of CPU, memory, port numbers, etc), and so on.

Controller Manager

The **controller manager** runs all the *controllers*, each of which is a control loop that continuously watches the state of the cluster and makes changes to move the cluster towards the desired state (you define the desired state via API calls). For example, the *node controller* watches worker nodes and tries to ensure the requested number of Nodes is always running and the *replication controller* watches containers and tries to ensure the requested number of containers is always running.

etcd

etcd is a distributed key-value store that the master nodes use as a persistent way to store the cluster configuration.

Worker nodes

The ***worker nodes*** (or just *nodes*, for short) are the servers that run your containers.

Components running on the worker nodes

The task of running your containers is up to the components running on each worker node:

1. The Kubelet
2. The Kubernetes Service Proxy (kube-proxy)
3. The Container Runtime (Docker, rkt, or others)

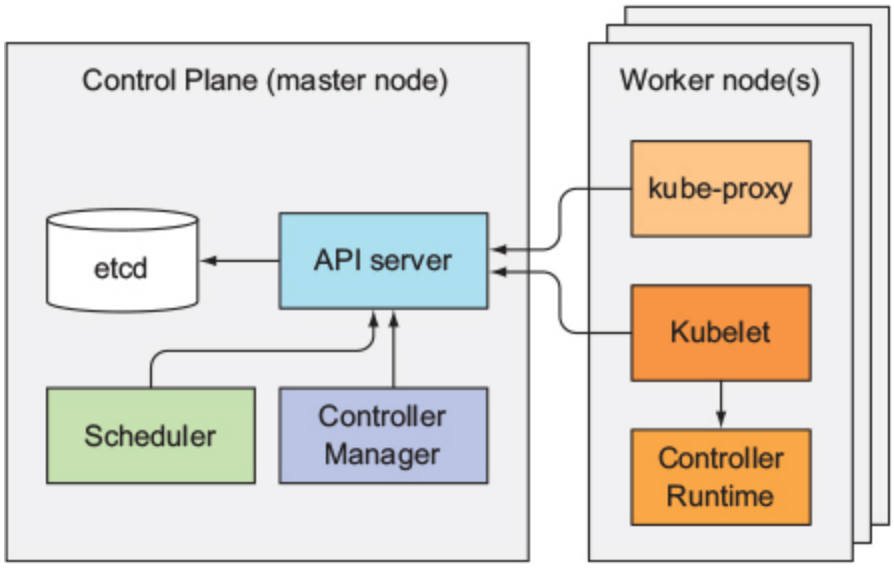
Kubelet

The **kubelet** is the primary agent that you run on each worker node. It is responsible for talking to the Kubernetes API Server, figuring out the containers that are supposed to be on its worker node, and deploying those containers, monitoring them, and restarting any containers that are unhealthy.

kube-proxy

The *Kubernetes Service Proxy (kube-proxy)* also runs on each worker node. It is responsible for talking to the Kubernetes API Server, figuring out which containers live at which IPs, and proxying requests from containers on the same worker node to those IPs. This is used for Service Discovery within Kubernetes, a topic we'll discuss later.

The components and their inter-dependencies are shown in the diagram below.



Kubernetes access control

Authentication

When a request is received by the API server, it goes through the list of authentication plugins, so they can each examine the request and try to determine who's sending the request. The first plugin that can extract that information from the request returns the username, user ID, and the groups the client belongs to back to the API server core. The API server stops invoking the remaining authentication plugins and continues onto the authorization phase.

Several authentication plugins are available. They obtain the identity of the client using the following methods:

- From the client certificate
- From an authentication token passed in an HTTP header
- Basic HTTP authentication
- Others

The authentication plugins are enabled through command-line options when starting the API server.

User accounts

User accounts are used by humans or other services outside of the Kubernetes cluster. For example, an admin at your company may distribute X509 certificates to your team members, or if you're using a Kubernetes service managed by your cloud provider (e.g., EKS in AWS or GKE in GCP), the user accounts may be the IAM user accounts you have in that cloud. Users are meant to be managed by an external system, such as a Single Sign On (SSO) system

Service accounts

Every pod is associated with a Service-Account, which represents the identity of the app running in the pod. The token file holds the ServiceAccount's authentication token. When an app uses this token to connect to the API server, the authentication plugin authenticates the ServiceAccount and passes the ServiceAccount's username back to the API server core. ServiceAccounts are nothing more than a way for an application running inside a pod to authenticate itself with the API server. ServiceAccounts are resources just like Pods, Secrets, ConfigMaps, and so on, and are scoped to individual namespaces

ROLE-BASED ACCESS CONTROL

The Kubernetes API server can be configured to use an authorization plugin to check whether an action is allowed to be performed by the user requesting the action. As you know, REST clients send GET, POST, PUT, DELETE, and other types of HTTP requests to specific URL paths, which represent specific REST resources. In Kubernetes, those resources are Pods, Services, Secrets, and so on. Here are a few examples of actions in Kubernetes:

- Get Pods

- Create Services
- Update Secrets
- And so on

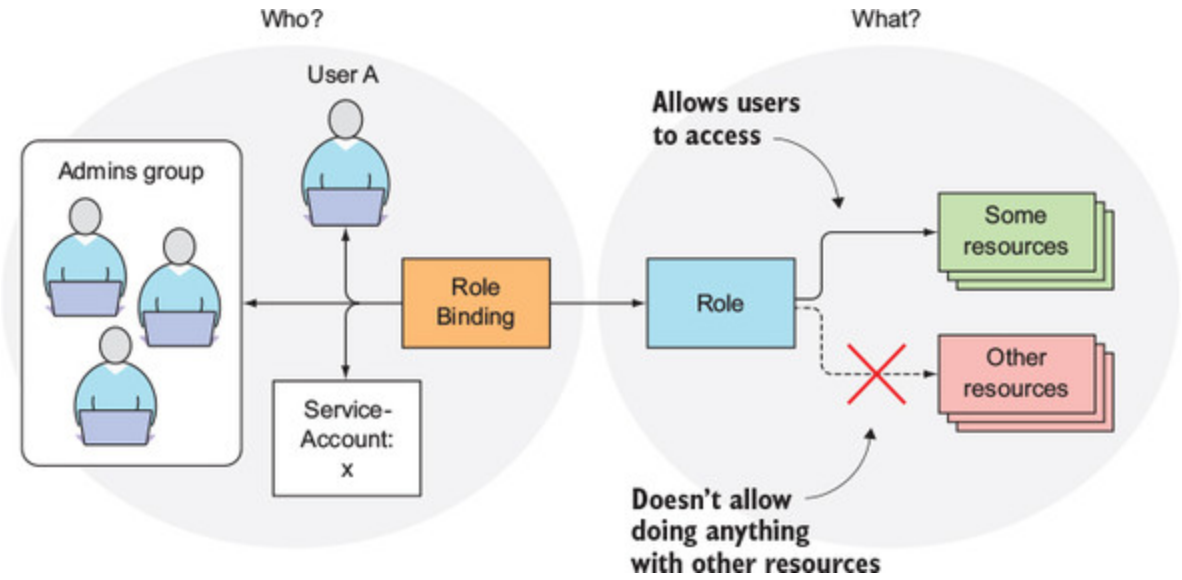
The verbs in those examples (get, create, update) map to HTTP methods (GET, POST, PUT) performed by the client (the complete mapping is shown in table 12.1). The nouns (Pods, Service, Secrets) obviously map to Kubernetes resources. An authorization plugin such as RBAC, which runs inside the API server, determines whether a client is allowed to perform the requested verb on the requested resource or not.

RBAC resources

The RBAC authorization rules are configured through four resources, which can be grouped into two groups:

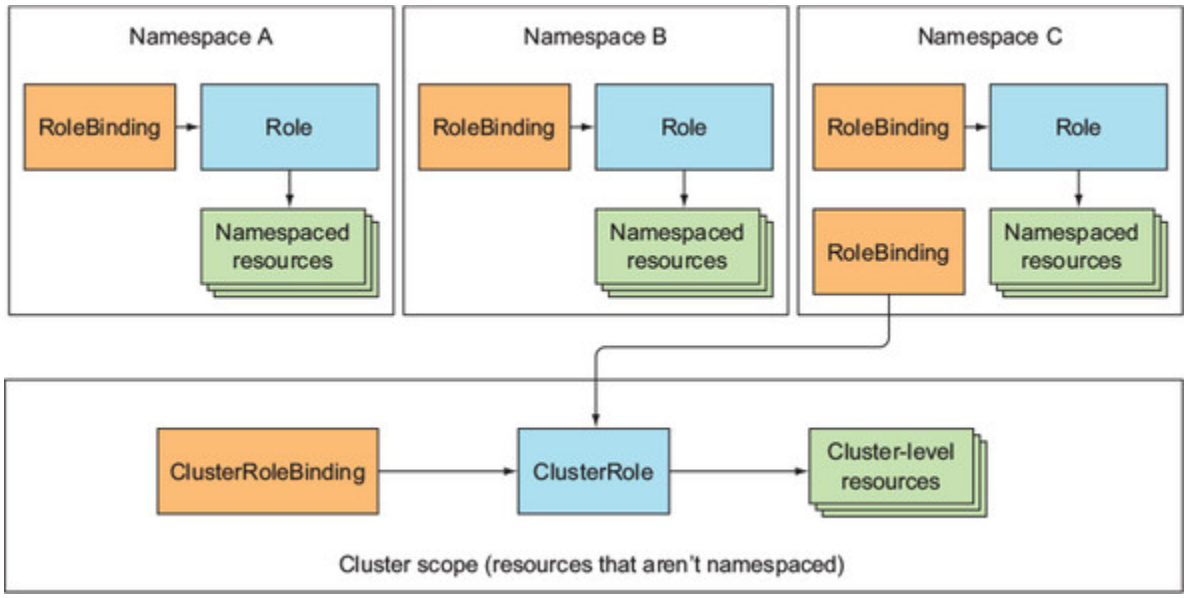
- Roles and ClusterRoles, which specify which verbs can be performed on which resources.
- RoleBindings and ClusterRoleBindings, which bind the above roles to specific users, groups, or ServiceAccounts.

Roles define *what* can be done, while bindings define *who* can do it. (see below diagram)



The distinction between a Role and a ClusterRole, or between a RoleBinding and a ClusterRoleBinding, is that the Role and RoleBinding are namespaced resources, whereas the ClusterRole and ClusterRoleBinding are cluster-level resources (not namespaced)

See the below figure



Kubernetes Core concepts

Pods

With Kubernetes, you don't run containers directly. Instead, the basic building block in Kubernetes is a *pod*, which is a group of one or more related containers that are always deployed together. For example, you could have a pod with just a single container, such as a container that runs a Node.js app, or a pod with several related containers, such as one container that runs a Node.js app, another container that runs a logs and metrics agent for the Node.js app, and a third container that runs nginx as a reverse proxy for the Node.js app.

Controllers

Pods are the basic building blocks of Kubernetes, but you typically don't deploy them directly either. A single pod on a single worker node is a single point of failure: the kubelet on that node can restart the pod if it crashes, but if the entire node crashes, or if you want to run multiple pods for high availability and scalability, you need a higher level construct. This is where controllers come in. Controllers allow you to manage multiple pods across multiple nodes.

The most common controller you're likely to use is the **Deployment**, which allows you to specify:

- What pod to deploy.
- How many *replicas* (copies) of that pod you want running.
- How to roll out updates to the replicas whenever you make a change.

VPC:

Amazon Virtual Private Cloud (Amazon VPC) lets you provision a logically isolated section of the AWS Cloud where you can launch AWS resources in a virtual network that

you define. You have complete control over your virtual networking environment, including selection of your own IP address range, creation of subnets, and configuration of route tables and network gateways. You can use both IPv4 and IPv6 in your VPC for secure and easy access to resources and applications.

AWS comes with the default VPC while creating an eks cluster. But we need to create a custom vpc for production.

VPC IP addresses

Here's how IP addresses work with AWS VPCs:

Private IP addresses

Every VPC defines an isolated network that has its own range of *private IP addresses*. For example, the Default VPC in AWS is configured to use all the IP addresses between 172.31.0.0 and 172.31.255.255; if you create a custom VPC, you can pick a custom IP address range to use, such as 10.10.0.0 to 10.10.255.255. These private IPs should be from the IP address ranges defined in **RFC 1918** (more on this later). Private IP addresses are only accessible from within the VPC, and not from the public Internet.

Public IP addresses

VPCs can also optionally be configured to assign *public IP addresses* to your resources (as is the case with the Default VPC). Public IPs are not associated with your VPC or even your AWS account; instead, they come from a pool of IP addresses shared by AWS across all of its customers (see **AWS IP Address Ranges**), so the IPs you get are

unpredictable, and may change (if you need consistent, predictable public IP addresses, you will need to use **elastic IP addresses**).

Assigning IP addresses

AWS will automatically assign IP addresses to resources you launch in a VPC. For example, in the Default VPC, one EC2 instance you launch might get the private IP address `172.31.0.2` and public IP address `203.0.113.25`, while another instance might get the private IP address `172.31.5.3` and the public IP address `54.154.202.112`.

CIDR notation

When dealing with networking, you often need to reason about ranges of IPs, such as "all IP addresses between `172.31.0.0` and `172.31.255.255`" (there are 65,536 IP addresses in this range). The de facto standard for representing IP address ranges is called ***Classless Inter-Domain Routing (CIDR) notation***. For example, the same 65,536 IP addresses can be represented in CIDR notation as `172.31.0.0/16`. This notation includes the *IP address* (`172.31.0.0`) and the *number of bits in the mask* (`/16`). To understand what the notation means, you:

1. Convert the IP address to binary: e.g., `172.31.0.0` in binary is `10101100.00011111.00000000.00000000`.
2. The mask tells you how many bits of the binary IP address identify the network (and stay constant for everything in that network) and how many bits identify unique hosts (and therefore, can vary). For a `/16` mask, the left-most 16 bits stay constant, while the right-most 16 bits are allowed to vary.

3. Putting that together, 172.31.0.0/16 represents all IP addresses from 10101100.00011111.00000000.00000000 (172.31.0.0) to 10101100.00011111.11111111.11111111 (172.31.255.255).

Refer **cidr** to get a better picture of cidr range

Subnets:

Each VPC is partitioned into one or more *subnets* (sub-networks). Each subnet controls a portion of the VPC's CIDR range. For example, a VPC with the CIDR block 10.10.0.0/16 (all IPs from 10.10.0.0 - 10.10.255.255) might be partitioned into two subnets, one with the CIDR block 10.10.0.0/17 (all IPs from 10.10.0.0 - 10.10.127.255) and one with CIDR block 10.10.128.0/17 (all IPs from 10.10.128.0 - 10.10.255.255). Note that subnets in the same VPC are not allowed to have overlapping CIDR ranges.

Route tables

Every subnet must define a ***route table*** that defines how to route traffic within that subnet. A route table consists of one or more *routes*, where each route specifies a *destination*, which is the range of IP addresses (in CIDR notation) to route, and the *target*, which is where to send the traffic for that range of IP addresses.

Here's an example route table:

Destination	Target
10.0.0.0/24	Local
0.0.0.0/0	igw-12345

This route table sends all traffic within the subnet's CIDR block, 10.0.0.0/24, to the *Local* route, which means it will be automatically routed within the subnet by AWS. This table then adds a fallback route for all other IPs (0.0.0.0/0) to send traffic to the Internet Gateway with ID igw-12345. We'll discuss Internet Gateways next.

Internet Gateways, public subnets, and private subnets

An Internet Gateway is a service managed by AWS that runs in your VPC. It allows access to and from the public Internet for resources in your subnet that have a public IP address (assuming you configure a route table entry in that subnet pointing to the Internet Gateway).

Subnets that have routes to Internet Gateways are called ***public subnets***, as the public IP addresses in those subnets can be accessed directly from the public Internet. Subnets that do not have routes to Internet Gateways are called ***private subnets***, as they will rely solely on routing to private IP addresses, which can only be accessed from within the VPC.

NAT Gateways

Resources in your public subnets can access the public Internet via an Internet Gateway. But what about resources in a private subnet? These resources don't have public IP addresses, nor a route to an Internet Gateway, so what do you do?

The solution is to deploy a ***NAT Gateway***. The NAT Gateway should run in a public subnet and have its own public IP address. It can perform *network address translation*, taking network requests from a resource in a private subnet, swapping in its own public IP address in those requests, sending them out to the public Internet (via the Internet Gateway in the public subnet), getting back a response, and sending the response back to the original sender in the private subnet.

In order for the NAT Gateway to work, you'll need to add a route to the route table for your private subnets:

Destination	Target
10.10.0.0/24	Local
0.0.0.0/0	nat-67890

This route table sends all traffic within the private subnet's CIDR block, 10.10.0.0/24, to the Local route, and the traffic for all other IPs, 0.0.0.0/0, to a NAT Gateway with ID nat-67890.

Security Groups

Most resources in AWS allow you to attach one or more ***security groups***, which are virtual firewalls that you can use to control which ports that resources opens for inbound

and outbound network traffic. By default, all ports are blocked, so to allow network communication, you can add inbound and outbound *rules*. Each rule in a security group specifies a port range, the IP addresses or other security groups that will be allowed to access that port range, and the protocol that will be allowed on those port range.

Here's an example of inbound rules:

Port range	Source	Protocol	Comment
80	10.0.0.0/16	tcp	Allow HTTP requests from within the VPC
443	10.0.0.0/16	tcp	Allow HTTPS requests from within the VPC
4000 - 5000	sg-abcd1234	tcp	Open a range of ports (e.g., for debugging) to another security group with ID sg-abcd1234

And here's an example of outbound rules:

Port range	Destination	Protocol	Comment
443	0.0.0.0/0	tcp	Allow all outbound requests over HTTPS so you can talk to the public Internet

Note that every VPC has a ***Default Security Group*** that will be used if you don't specify any other security group for your resources. We recommend always attaching a custom security group with rules that exactly match your use case, rather than relying on this default, global one.

Network ACLs

In addition to security groups, which act as firewalls on individual resources (e.g., on an EC2 instance), you can also create ***network access control lists (NACLs)***, which act as firewalls for an entire subnet. Just as with security groups, NACLs have inbound and outbound rules that specify a port range, the IP addresses that can talk to that port range, and the protocol that will be allowed on that port range.

However, there are two main differences with NACLs:

Allow/Deny

Each NACL rule can either ALLOW or DENY the traffic defined in that rule.

Stateful/Stateless

Security groups are *stateful*, so if they have a rule that allows an inbound connection on, say, port 80, the security group will automatically also open up an outbound port for that specific connection so it can respond. With a NACL, if you have a rule that allows an inbound connection on port 80, that connection will not be able to respond unless you also manually add another rule that allows outbound connections for the response. You normally don't know exactly which port will be used to respond: these are called ***ephemeral ports***, and the rules depend on the operating system.

For example, the networking stack on Linux usually picks any available port from the range 32768-61000, whereas Windows Server 2003 uses 1025-5000, NAT Gateways use

1024-65535, and so on. Therefore, in practice, you typically have to open ephemeral ports 1024-65535 in your NACL, both for inbound and outbound (as when you establish outbound connections, anyone responding will likely do so on an ephemeral port), making them primarily useful for locking down the low-numbered ports (< 1024) used for standard protocols (e.g., HTTP uses port 80), and locking down source/destination IP addresses.

VPC Considerations:

When you create an Amazon EKS cluster, you specify the VPC subnets for your cluster to use. Amazon EKS requires subnets in at least two Availability Zones. We recommend a VPC with public and private subnets so that Kubernetes can create public load balancers in the public subnets that load balance traffic to pods running on worker nodes that are in private subnets.

Add tags to the VPC and subnets:

When you create an Amazon EKS cluster earlier than version 1.15, Amazon EKS tags the VPC containing the subnets you specify in the following way so that Kubernetes can discover it:

Key	Value
`kubernetes.io/cluster/<cluster-name>`	`shared`

- **Key**: The `<cluster-name>` value matches your Amazon EKS cluster's name.
- **Value**: The `shared` value allows more than one cluster to use this VPC.

Subnet tagging requirement

When you create your Amazon EKS cluster, Amazon EKS tags the subnets you specify in the following way so that Kubernetes can discover them:

Key	Value
`kubernetes.io/cluster/<cluster-name>`	`shared`

- **Key**: The `<cluster-name>` value matches your Amazon EKS cluster.
- **Value**: The `shared` value allows more than one cluster to use this subnet.

Public subnet tagging option for external load balancers

You must tag the public subnets in your VPC so that Kubernetes knows to use only those subnets for external load balancers instead of choosing a public subnet in each Availability Zone (in lexicographical order by subnet ID)

Key	Value
`kubernetes.io/role/elb`	`1`

Control plane

To have EKS manage the control plane for you, you need to create an ***EKS cluster***. When you create an EKS cluster, behind the scenes, AWS fires up several master nodes in a highly available configuration, complete with the Kubernetes API Server, scheduler, controller manager, etcd. Here are the key considerations for your EKS cluster:

Subnets

Your EKS cluster will run in the subnets you specify. We strongly recommend running solely in private subnets that are NOT directly accessible from the public Internet.

Endpoint access

You can configure whether the **API endpoint for your EKS cluster** is accessible from (a) within the same VPC and/or (b) from the public Internet. We recommend allowing access from within the VPC, but not from the public Internet. If you need to talk to your Kubernetes cluster from your own computer (e.g., to issue commands via kubectl), use a bastion host or VPN server.

Cluster IAM Role

To be able to make API calls to other AWS services, **your EKS cluster must have an IAM role** with the following managed IAM policies: AmazonEKSServicePolicy and AmazonEKSClusterPolicy.

Security group

You should define a security group that controls what traffic can go in and out of the control plane. The worker nodes must be able to talk to the control plane and vice versa: see **Cluster Security Group Considerations** for the ports you should open up between them.

Worker nodes

Auto Scaling Group

We recommend using an **Auto Scaling Group** to run your worker nodes. This way, failed nodes will be automatically replaced, and you can use auto scaling policies to automatically scale the number of nodes up and down in response to load.

Tags

EKS requires that all worker node EC2 instances have a tag with the key kubernetes.io/cluster/<CLUSTER_NAME> and value owned.

Subnets

We strongly recommend running the Auto Scaling Group for your worker nodes in private subnets that are NOT directly accessible from the public Internet.

User Data:

Since unmanaged nodegroup do not register by itself to EKS clusters, we need to explicitly run bootstrap script while creating worker nodes. Note that if the bootstrap script is not executed, worker nodes do register to the cluster.

IAM role

In order for the kubelet on each worker node to be able to make API calls, each **worker node must have an IAM role** with the following managed IAM policies: AmazonEKSWorkerNodePolicy, AmazonEKS_CNI_Policy, AmazonEC2ContainerRegistryReadOnly.

Security group

You should define a security group that controls what traffic can go in and out of the control plane. The worker nodes must be able to talk to the control plane and vice versa: see **Cluster Security Group Considerations** for the ports you should open up between them.

IAM role mapping and RBAC

You've seen that to determine *who* the user is (authentication), EKS uses IAM. The next step is to determine *what* the user can do (authorization). Kubernetes uses its own roles and RBAC for authorization, so the question is, how does EKS know which IAM entities (that is, IAM users or roles) are associated with which Kubernetes roles?

The answer is that EKS expects you to define a ConfigMap called aws-auth that defines the mapping from IAM entities to Kubernetes roles. When you first provision an EKS cluster, the IAM user or role that you used to authenticate is automatically granted admin level permissions (the system:master role). You can use this role to add additional role mappings in the aws-auth ConfigMap.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: aws-auth
  namespace: kube-system
data:
  mapRoles: |
```

```
    - rolearn: arn:aws:iam::11122223333:role/example-role
      username: system:node:{{EC2PrivateDNSName}}
      groups:
        - system:bootstrappers
        - system:nodes
  mapUsers: |
    - userarn: arn:aws:iam::11122223333:user/example-user
      username: example-user
      groups:
        - system:masters
```

Production-grade design:

Architecture: