EKS Deep Dive

Bryan Landes, Solutions Architect

May 6th, 2020

Table of contents

- ~~Welcome~~
- ~~Introduction to Containers~~
- ~~Configuration & Setup~~
- ~~EKS Architecture Overview and EKS Fargate~~
- Security
- Operations
- DevOps
- Networking
- Storage
- Logging
- Monitoring (Observability)
- Application communication

aws

What’s New? - Containers

- ECR now supports Manifest Lists for multi-architecture images
- Amazon EKS now supports Kubernetes version 1.16
- EKS Adds Fargate Support in Frankfurt, Oregon, Singapore, and Sydney AWS Regions

aws

What’s New? – Blog Posts

- Introducing multi-architecture container images for Amazon ECR
- Fault tolerant distributed machine learning training with the TorchElastic Controller for Kubernetes
- Optimizing Spark performance on Kubernetes
- Under the hood: AWS Fargate data plane

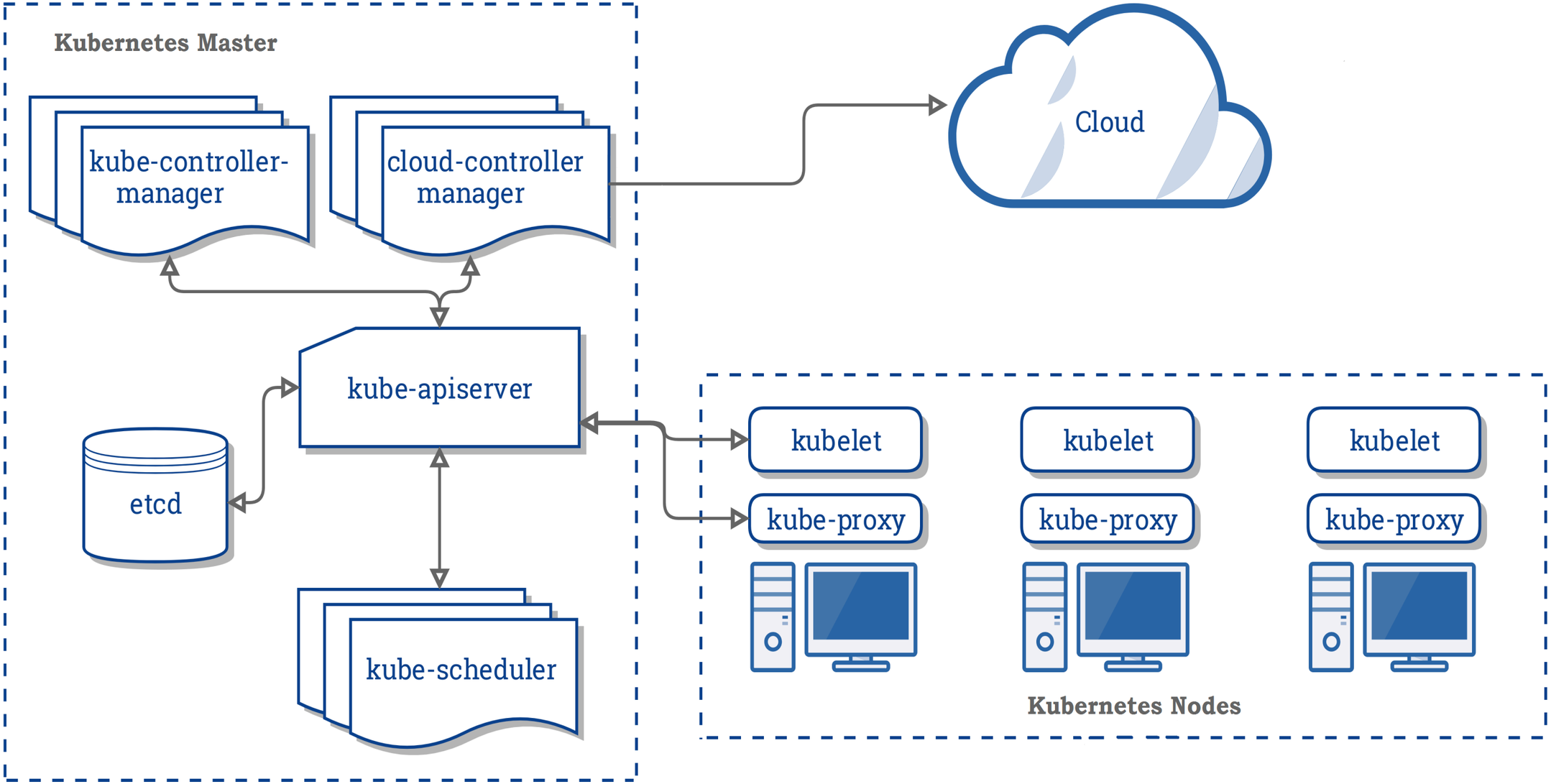
aws

Key Kubernetes concepts

- Kubernetes control plane
  - Provided by master node objects/components
- Kubernetes data plane
  - Provided by worker nodes objects/components
- Kubernetes master node
  - Kube-apiserver
  - Kube-controller-manager
  - Kube-scheduler
  - etcd
- Kubernetes worker nodes
  - Kubelet
  - Kube-proxy
  - Container runtime
  - Pods

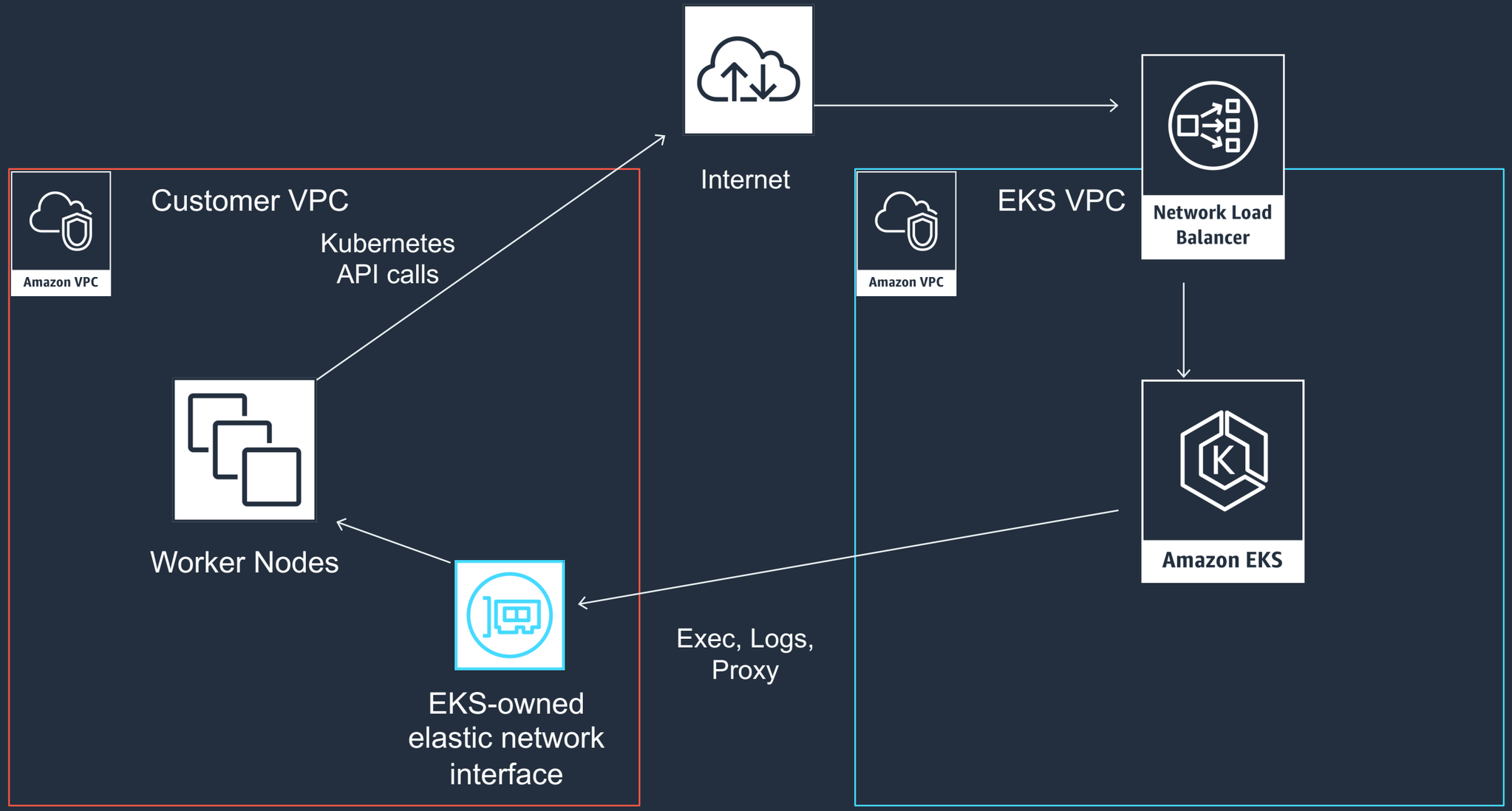
aws

Kubernetes architecture



https://kubernetes.io/docs/concepts/overview/components/

Amazon EKS Architecture



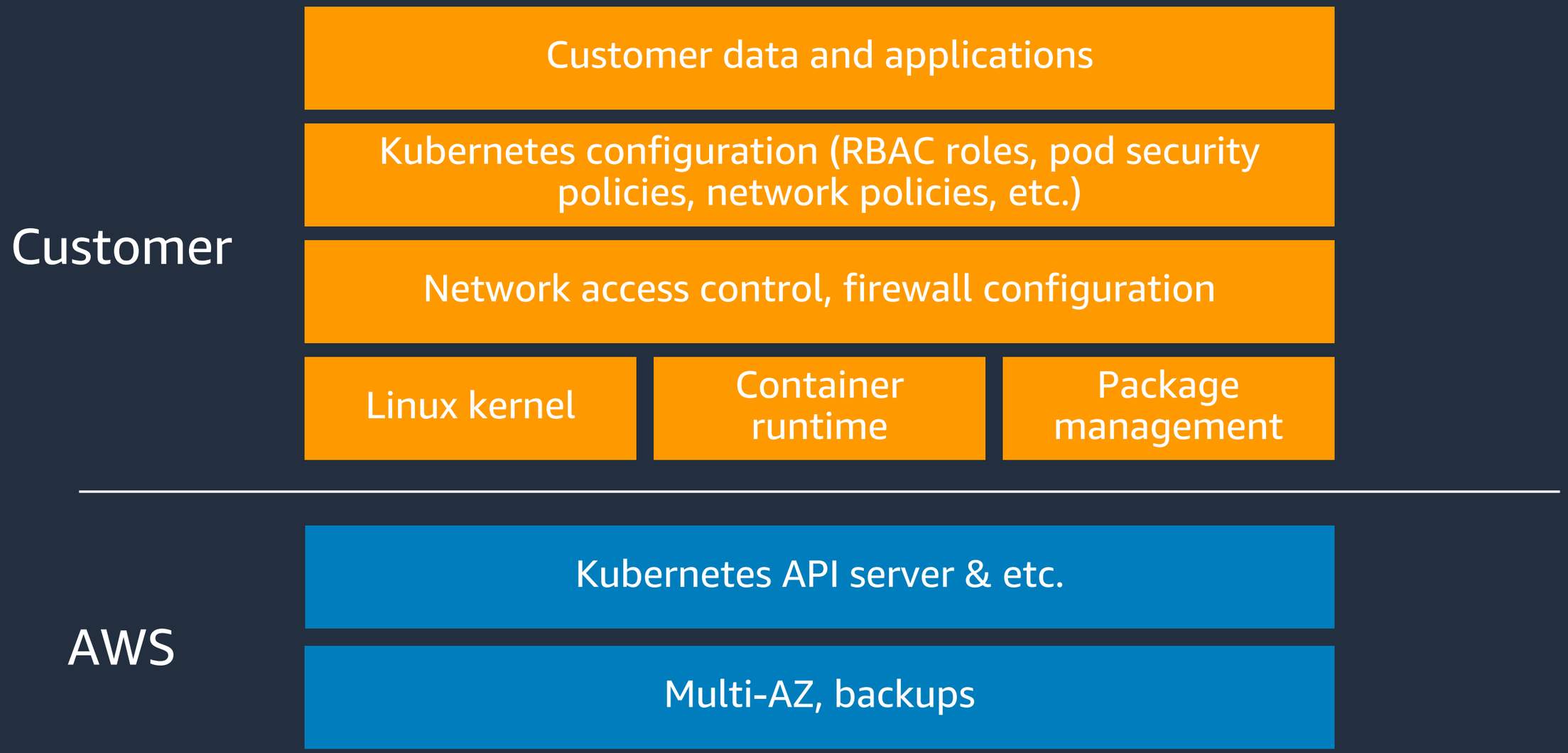
Fargate vs. (Un)Managed Nodes

	Fargate	Managed nodes	Unmanaged nodes
Units of work	Pod	Pod and EC2	Pod and EC2
Unit of charge	Pod	EC2	EC2
Host lifecycle	There is no visible host	AWS (SSH is allowed)	Customer
Host AMI	There is no visible host	AWS vetted AMIs	Customer BYO
Host : Pods	1 : 1	1 : many	1 : many

aws

Security

The shared responsibility model

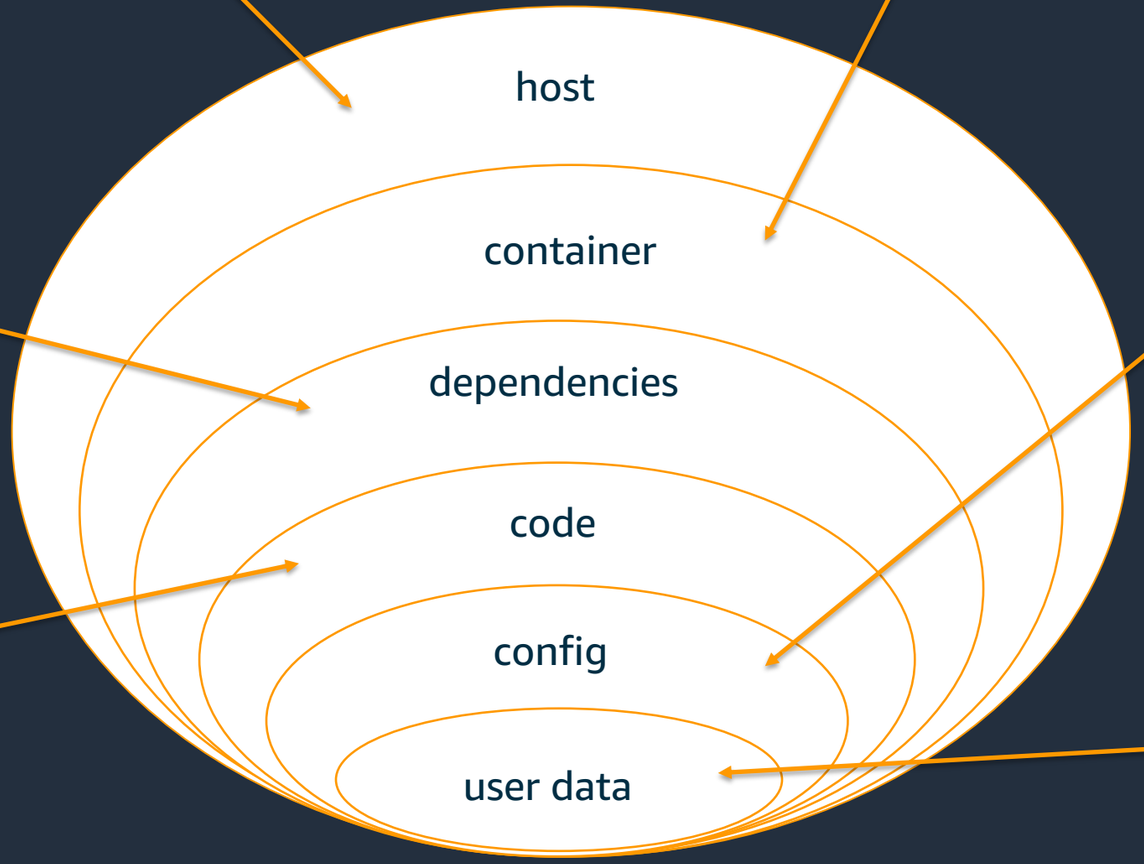


aws

Container security onion model: Defense in depth

- full blown distro (Ubuntu, AL) vs. minimal environment (container-optimized distribution)
- multi-tenancy requirements
- **gotchas**: Linux packages/CVEs, leaks, GDPR (in Europe)

- runtime/standards (OCI)
- immutability of images
- all containers share a kernel (mitigation: Firecracker)
- **gotchas**: unnecessary privileged users, no scans, trust



- code analysis
- source available?
- **gotchas**: big surface, many languages

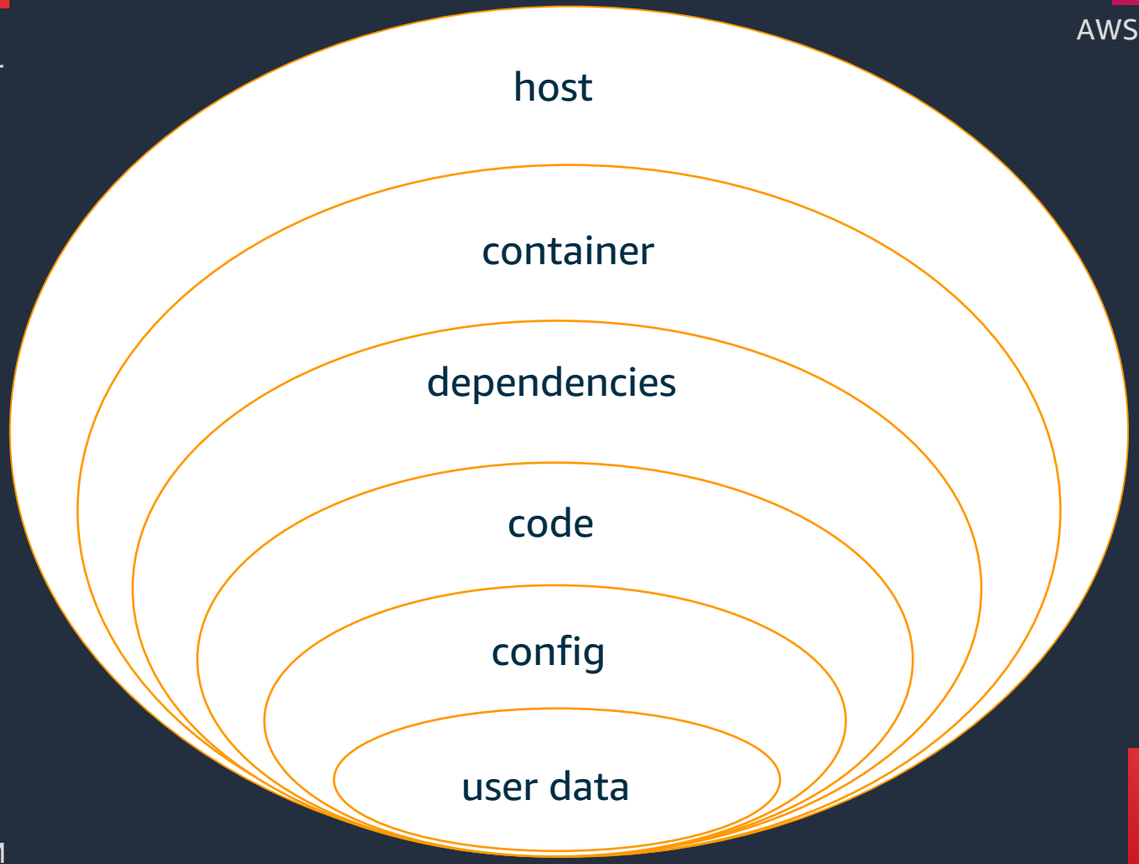
- sensitive config (passwords, API keys, etc.)
- **gotchas**: commits-to-source, non-separated access (dev has cleartext password)

- sanitizing user input
- static code analysis
- **gotchas**: log-leaking

- business core data
- Personal Identifiable Information (PII)
- **gotchas**: leaks, GDPR (in Europe)

aws

Security tooling



AWS Certificate Manager (ACM)

AWS Security Hub

IAM

Amazon Inspector

AWS CloudTrail

Amazon GuardDuty

AWS WAF

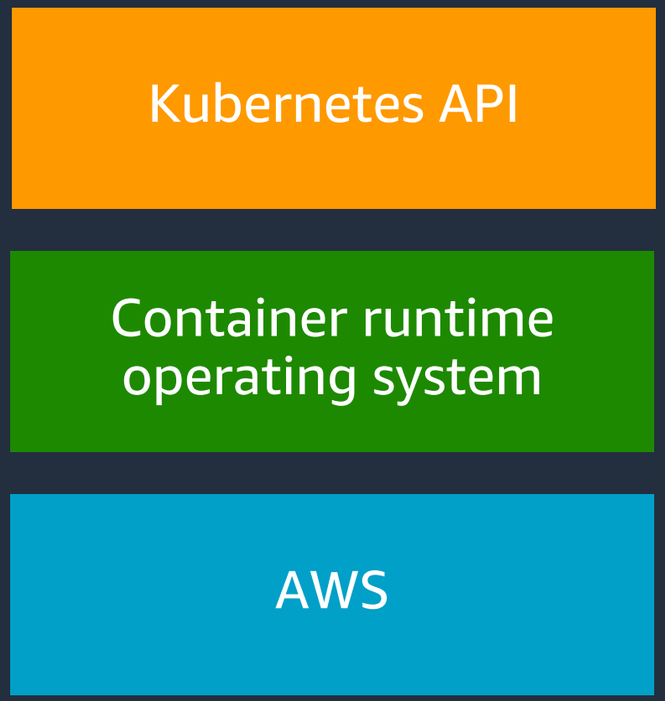
AWS KMS

AWS Secrets Manager

AWS CloudHSM

Amazon Macie

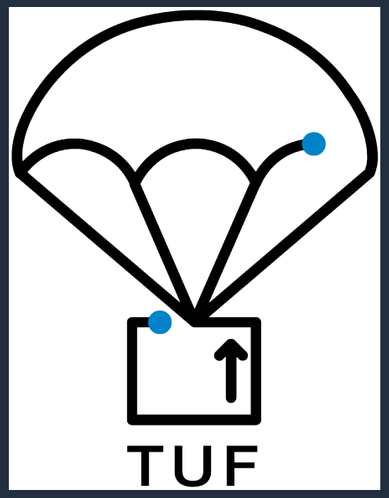
Areas of control



aws

Bottlerocket

- API access for configuring your system, with secure out-of-band access methods when you need them.
- Updates based on partition flips, for fast and reliable system updates.
- Modeled configuration that's automatically migrated through updates.
- Security as a top priority.
- **The Update Framework**



aws

Kubernetes controls

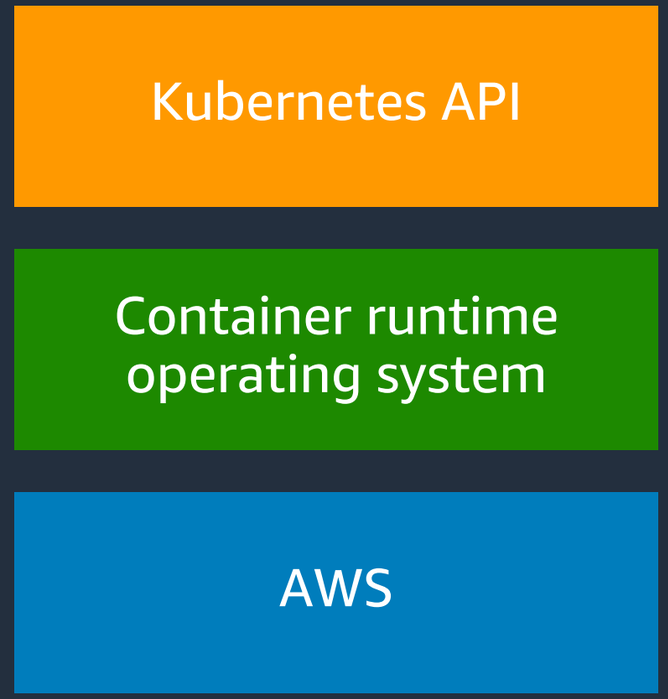
Namespaces
- Logical partition within a cluster

Service accounts
- Kubernetes identities assigned to pods

Resource quotas
- Limit total resources in a namespace

Network policies
- Pod-aware network controls

Limit ranges
- Limit pod, container, or volume resource sizes



aws

Kubernetes controls

Role-based access control (RBAC)

- Authorization policy API

Dynamic admission webhooks

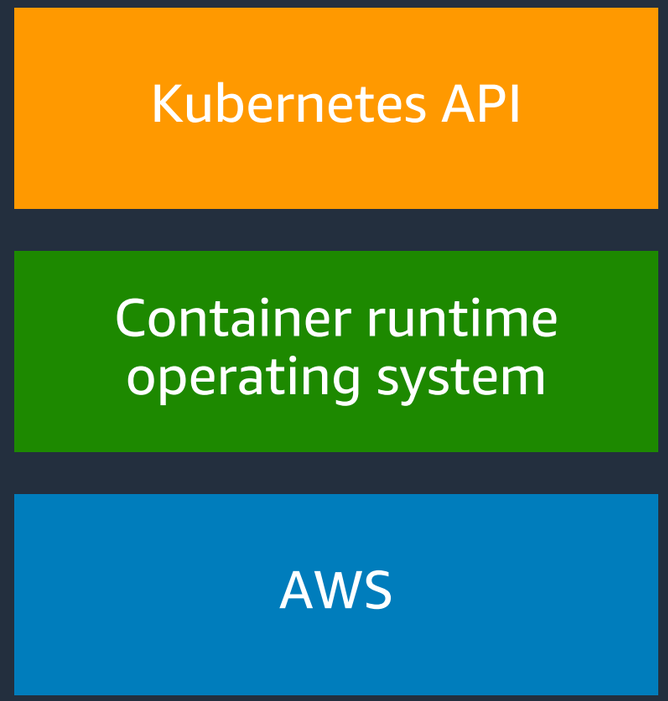
- Can place arbitrary restrictions on requests made to the API

API audit log

- API logging, including request or response bodies

Pod security policies

- Restrictions on pod and container permissions



aws

Pod security policies

Defines the conditions for a pod to run

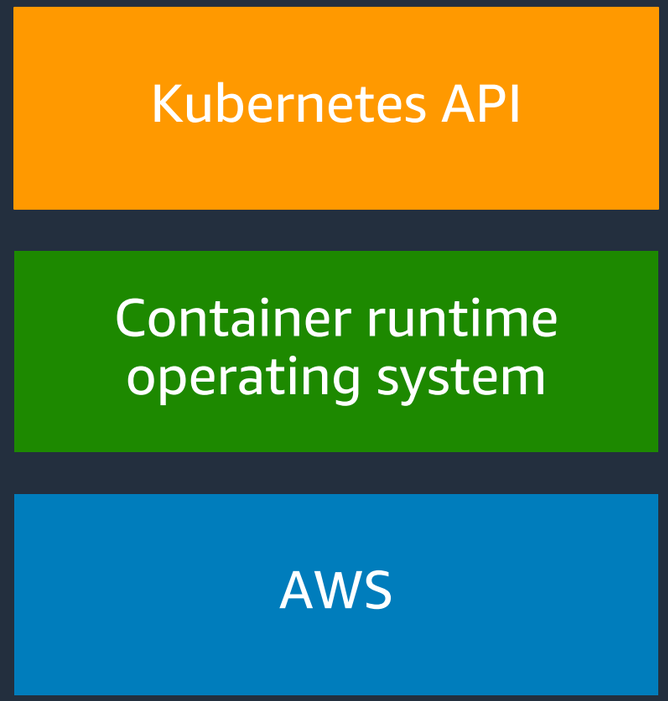
- Disallow running containers as root
- Disallow containers that require root privileges

Enforcement in the Kubernetes API

- A combination of an admission controller and authorization to grant service account access

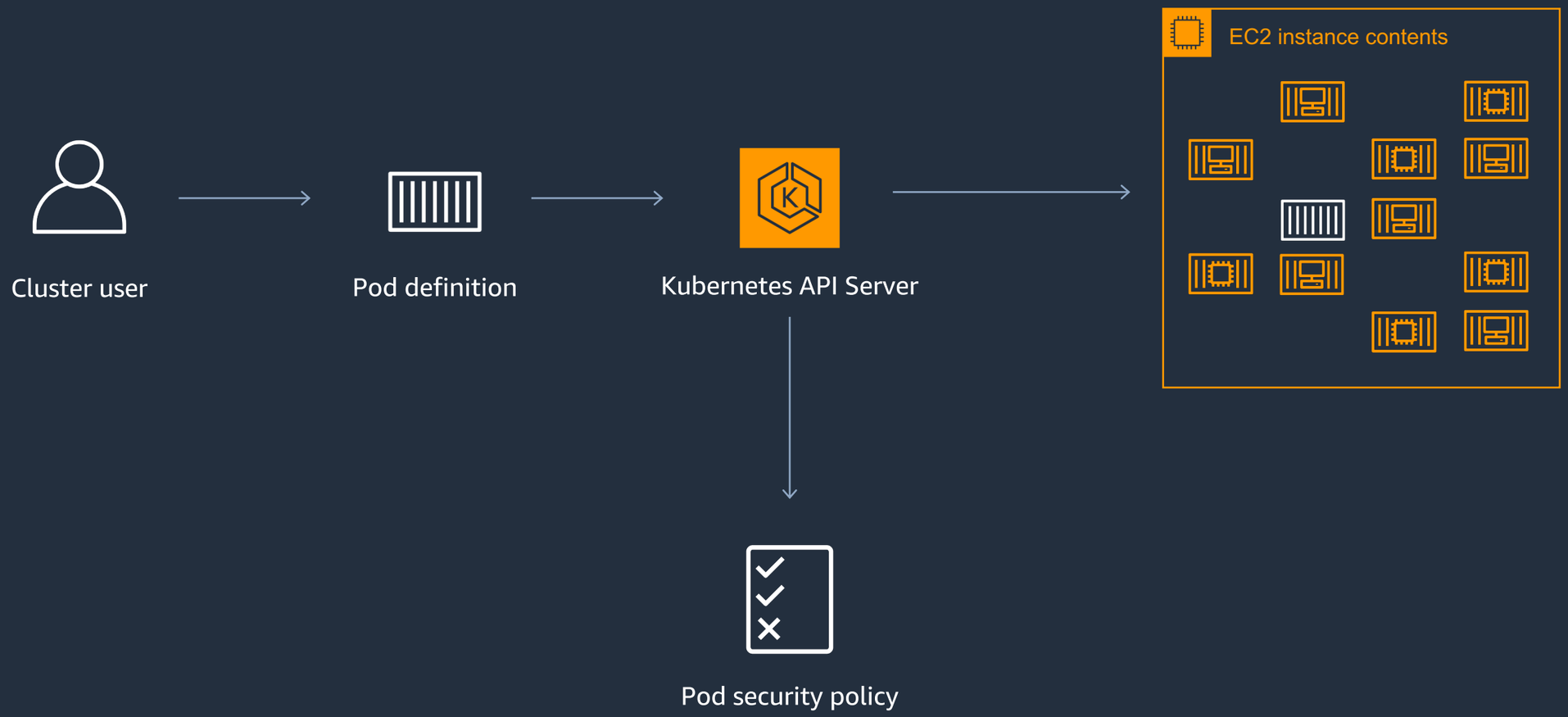
Supports multiple policies

- A policy for privileged containers
- A policy for unprivileged containers



aws

Pod security policies



A restrictive pod security policy example

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restrictive
spec:
  privileged: false
  allowPrivilegeEscalation: false
  requiredDropCabilities: ['ALL']
  volumes: ['configMap', 'secret', 'projected’]
  hostnetwork: false
  hostIPC: false
  hostPID: false
  readOnlyRootFilesystem: true
```

A restrictive pod security policy example

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restrictive
spec:
  privileged: false # Prevent escalation to root
  allowPrivilegeEscalation: false
  requiredDropCabilities: ['ALL’]
  volumes: ['configMap', 'secret', 'projected']
  hostnetwork: false
  hostIPC: false
  hostPID: false
  readOnlyRootFilesystem: true
```

A restrictive pod security policy example

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restrictive
spec:
  privileged: false
  allowPrivilegeEscalation: false
  requiredDropCabilities: ['ALL']
  volumes: ['configMap', 'secret', 'projected'] # Volumes are restricted to a whitelist
  hostnetwork: false
  hostIPC: false
  hostPID: false
  readOnlyRootFilesystem: true
```

A restrictive pod security policy example

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restrictive
spec:
  privileged: false
  allowPrivilegeEscalation: false
  requiredDropCabilities: ['ALL']
  volumes: ['configMap', 'secret', 'projected']
  hostnetwork: false  # Disallows direct use of host networking namespace and resources
  hostIPC: false
  hostPID: false
  readOnlyRootFilesystem: true
```

A restrictive pod security policy example

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restrictive
spec:
  privileged: false
  allowPrivilegeEscalation: false
  requiredDropCabilities: ['ALL']
  volumes: ['configMap', 'secret', 'projected']
  hostnetwork: false
  hostIPC: false
  hostPID: false
  readOnlyRootFilesystem: true # Prevents writes to root filesystem
```

Container image management controls

Amazon Elastic Container Registry (Amazon ECR) endpoint policy

- Set condition "PrincipalOrgID"
- Prevents the pushing and pulling of images outside

Open policy agent (OPA)

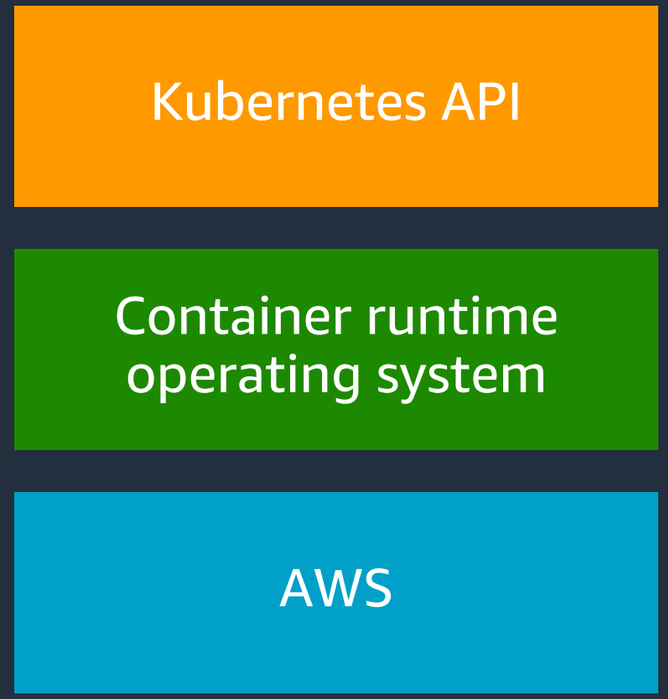
- Restricts the pulling to whitelisted image registries

OPA for MITM attack mitigation to enforce signing

- Restrict the pulling of signed images only

Container runtime API access restriction

- Ensure API is not accessible outside of unix socket



aws

AWS controls

Control plane logging

- Including Kubernetes API audit logs

Endpoint access

- Private endpoint access
- Public CIDR restrictions

Authentication control

- Manage which AWS Identity and Access Management (IAM) identities can authenticate to the cluster

Amazon ECR image scanning

- Scan container images on push

IAM roles for service accounts

- Assign IAM roles to pods via the service account

aws

IAM roles for Service Accounts

Secure

- IAM policy restrictions can restrict roles to service accounts or namespaces
- Enables isolated AWS permissions per service account
- Credentials are automatically rotated
- The cluster's signing key is automatically rotated

Easy integration

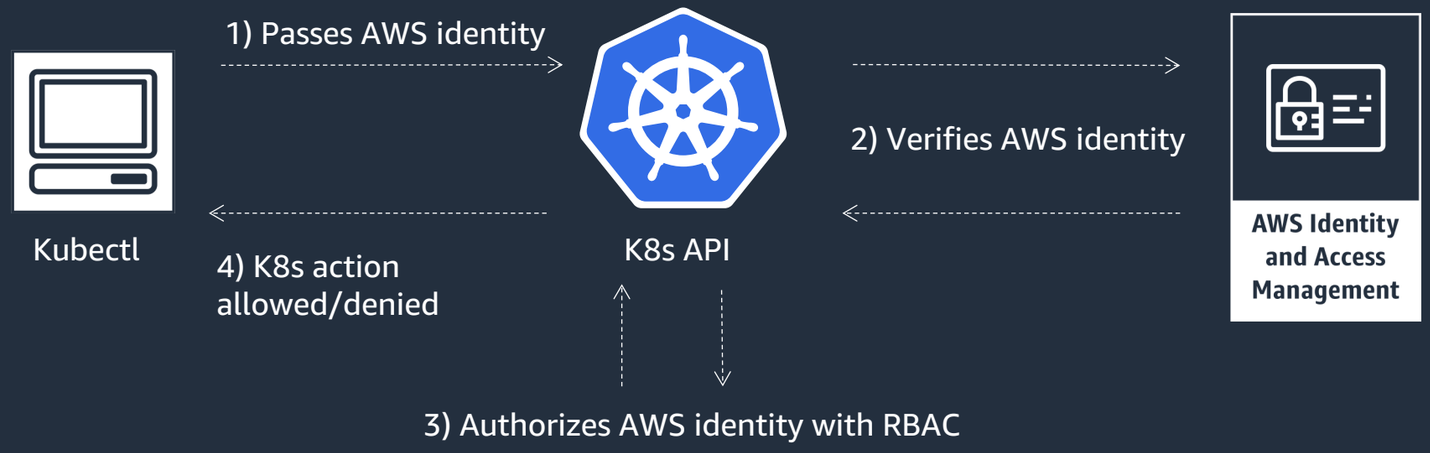
- Annotate the service account
- Built into the default credential chains in the AWS SDKs and AWS CLI

Auditable

- Service account names are logged in AWS CloudTrail

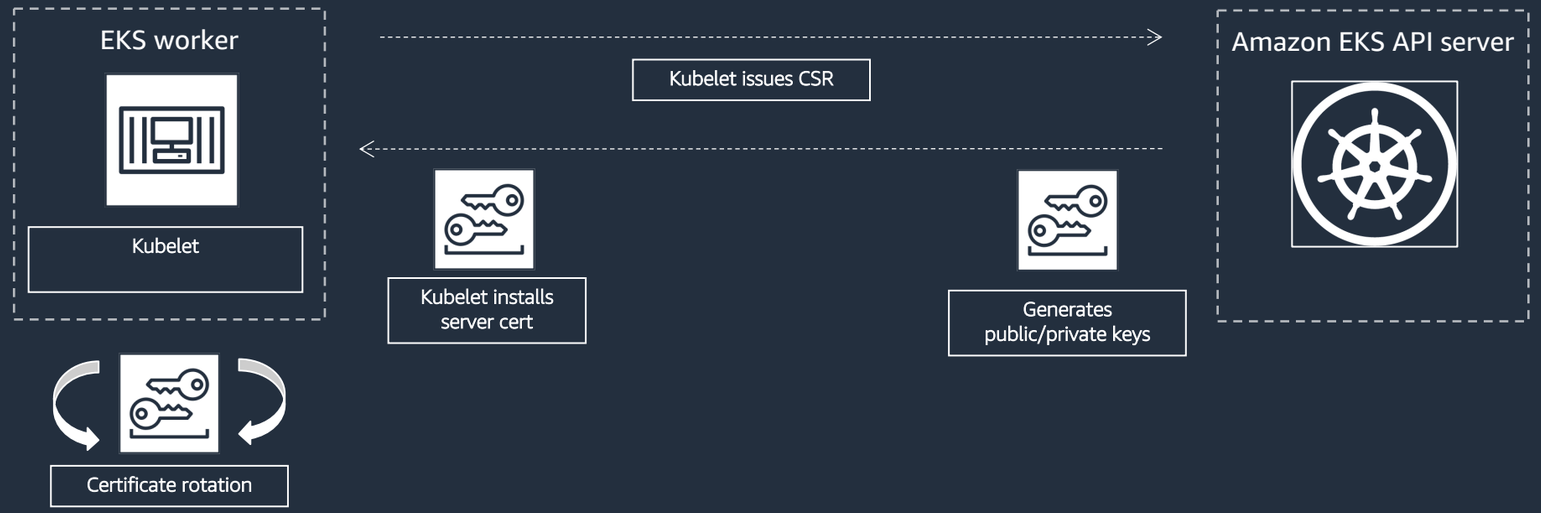
aws

IAM authentication



PKI configuration

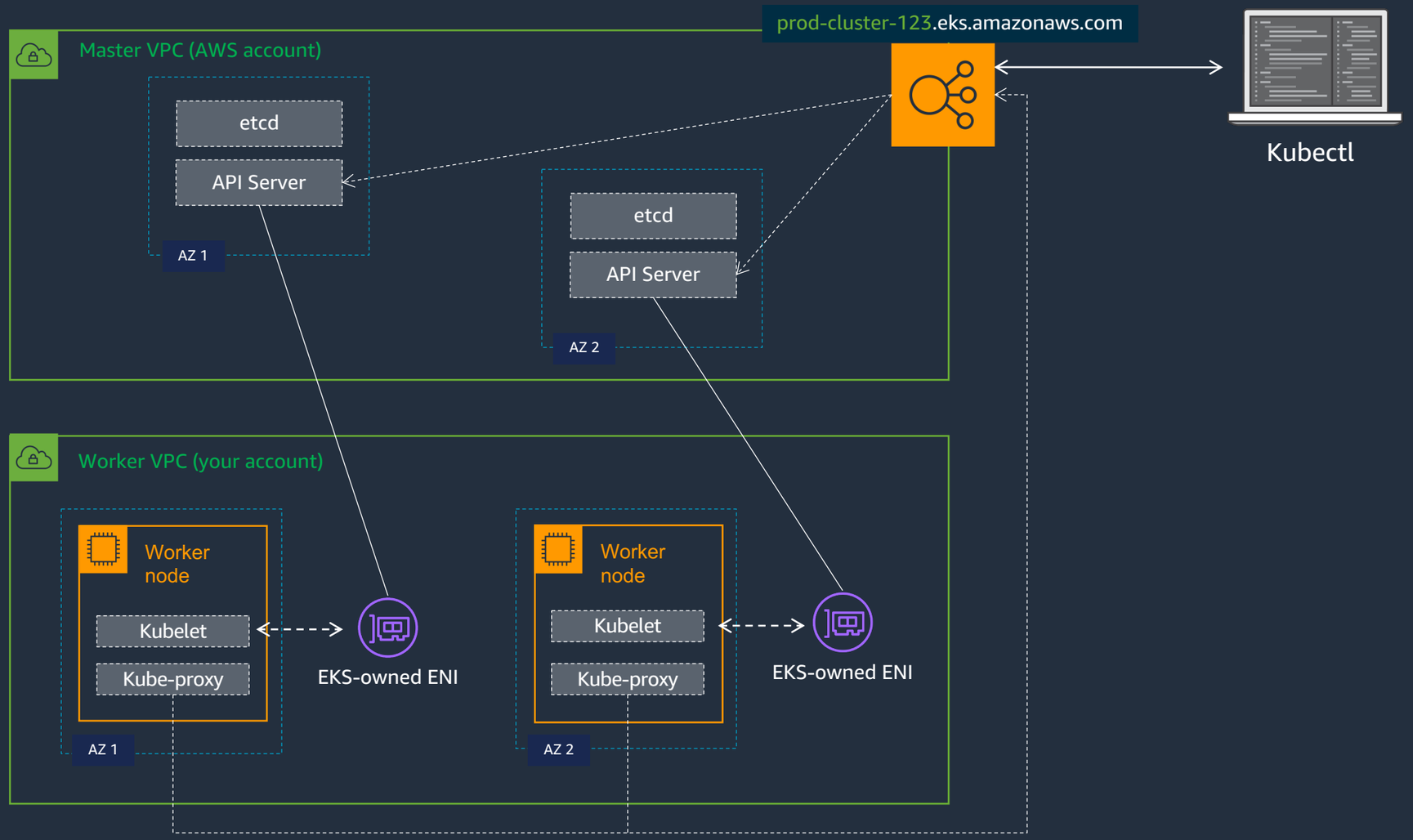
Each Amazon EKS cluster is a unique CA



API-server endpoint access control

Public == true

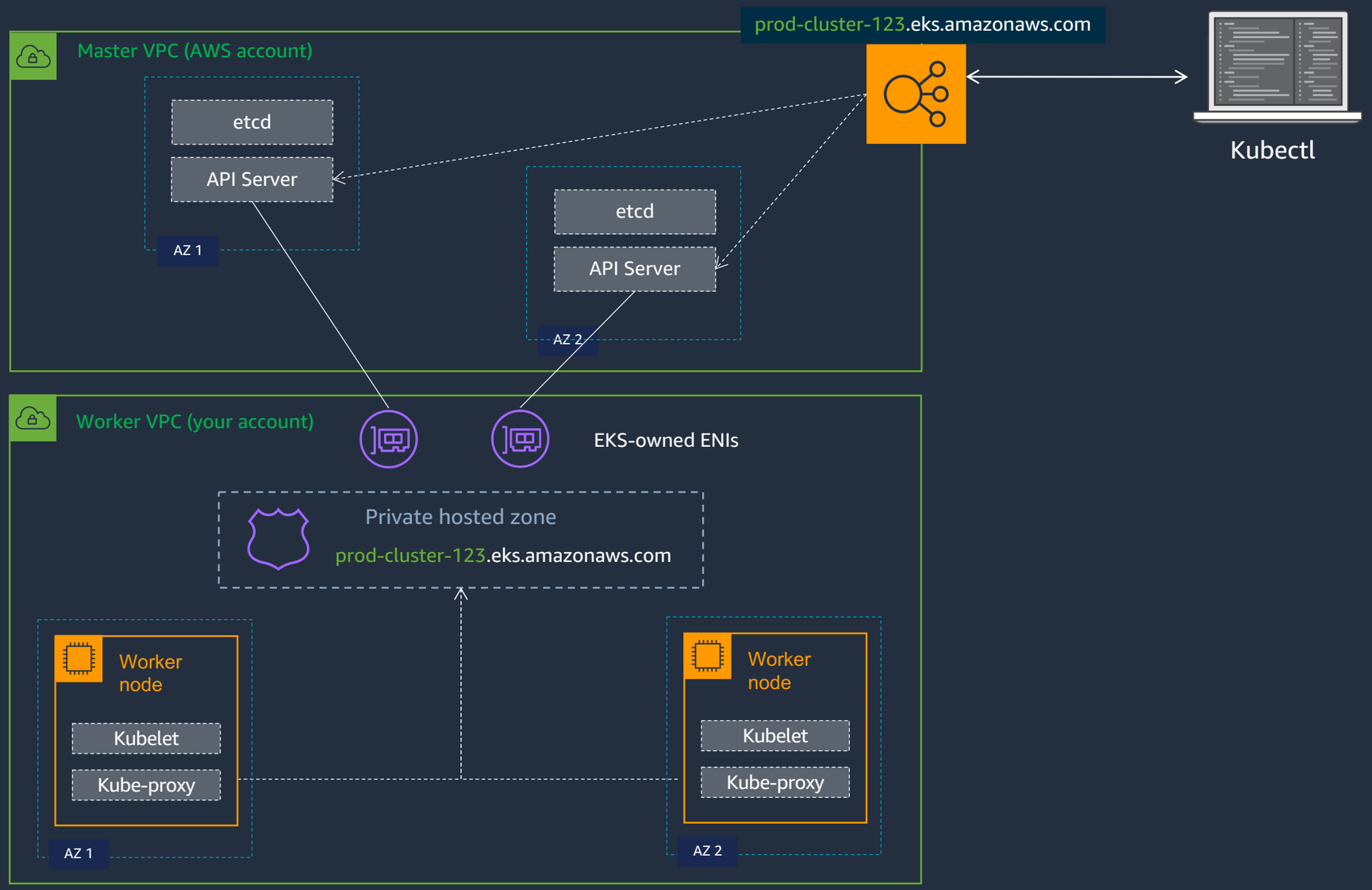
Private == false



API-server endpoint access control

Public == true

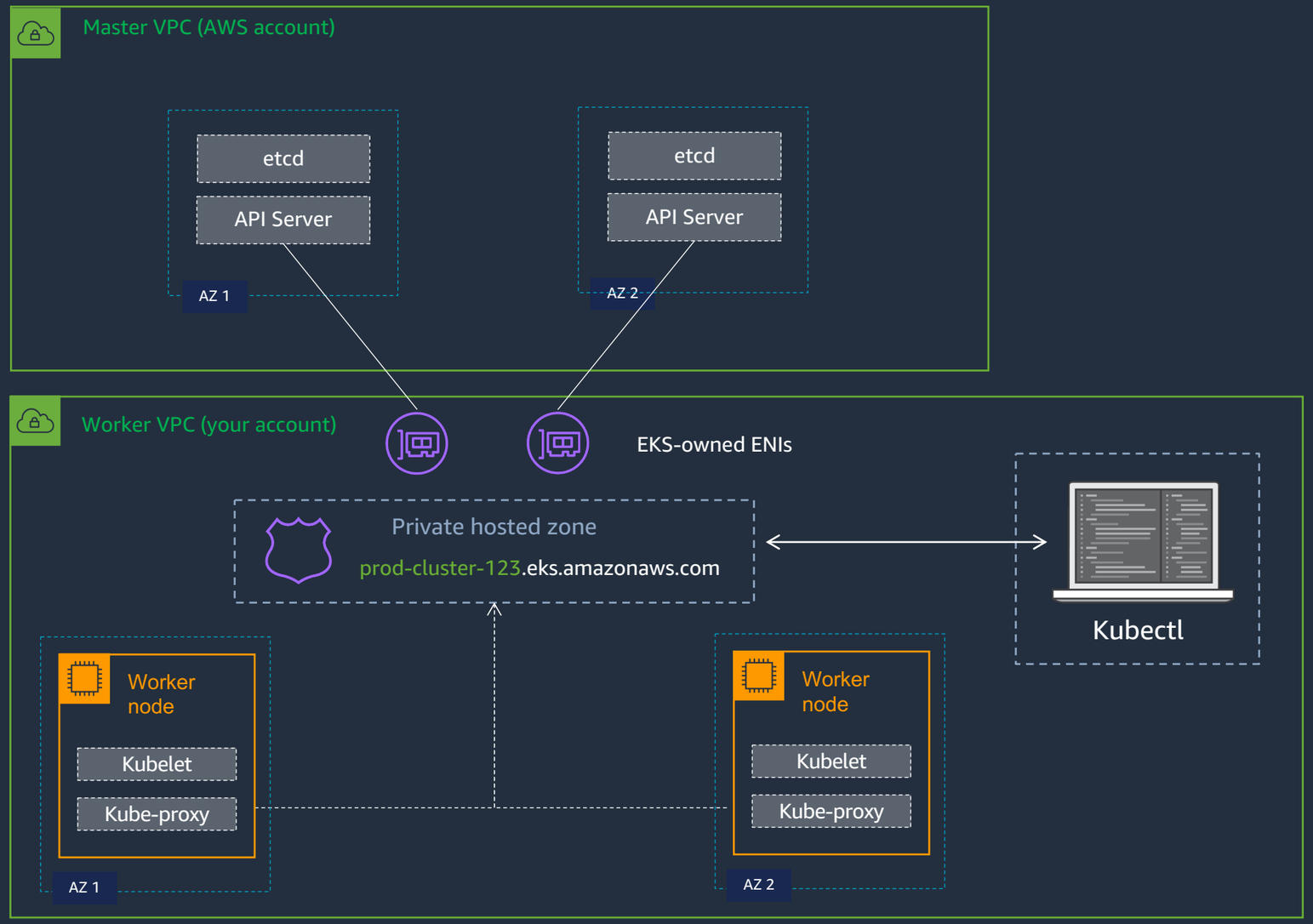
Private == true



API-server endpoint access control

Public == false

Private == true



Encrypting Kubernetes Secrets with KMS

1. The user (typically in an admin role) creates a secret.

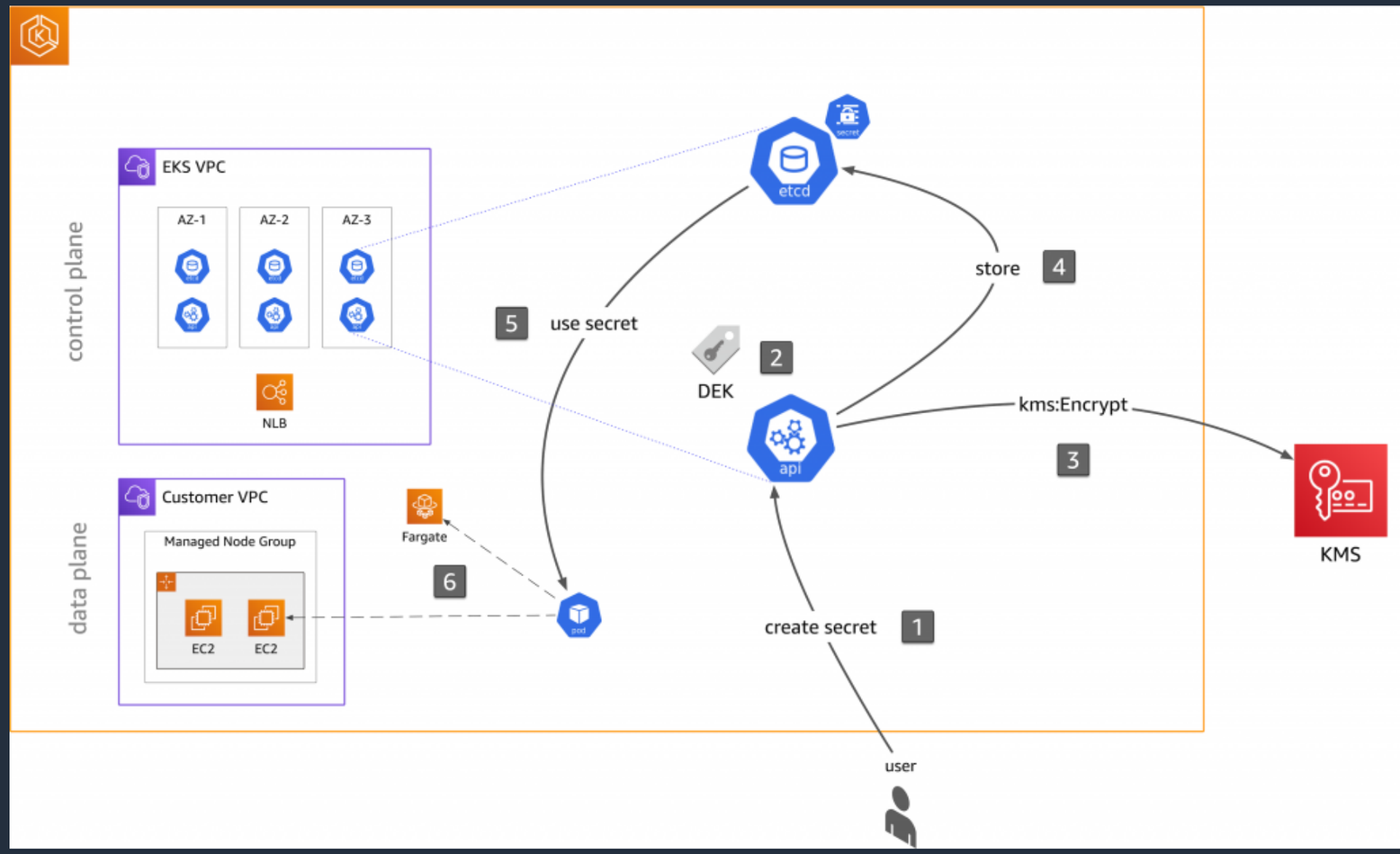
2. The Kubernetes API server in the EKS control plane generates a Data Encryption Key (DEK) locally and uses it to encrypt the plaintext payload in the secret. Note that the control plane generates a unique DEK for every single write, and the plaintext DEK is never saved to disk.

3. The Kubernetes API server calls kms:Encrypt to encrypt the DEK with the CMK. This key is the root of the key hierarchy, and, in the case of KMS, it creates the CMK on a hardware security module (HSM). In this step, the API server uses the CMK to encrypt the DEK and also caches the base64 of the encrypted DEK.

4. The API server stores the DEK-encrypted secret in etcd.

5. If one now wants to use the secret in, say a pod via a volume (read-path), the reverse process takes place. That is, the API server reads the encrypted secret from etcd and decrypts the secret with the DEK.

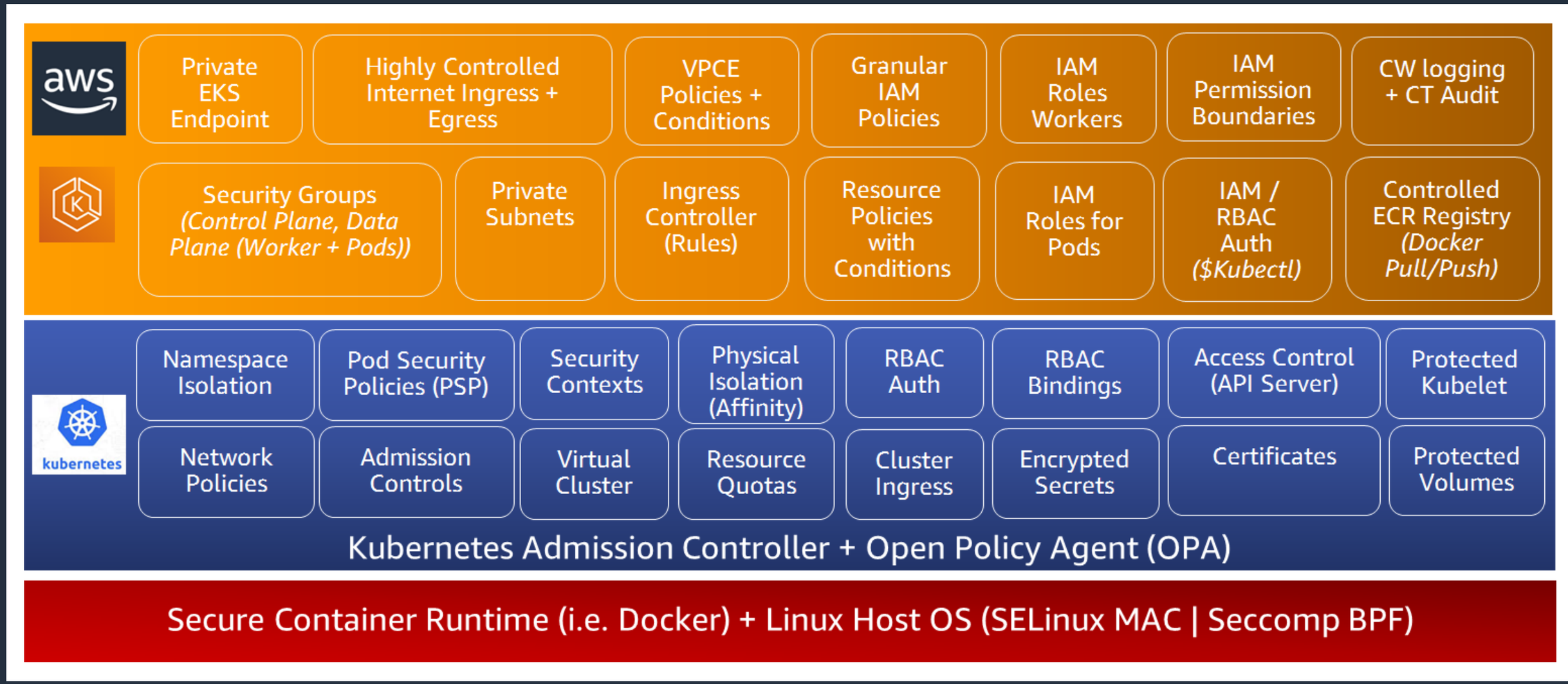
6. The application, running in a pod on either EC2 or Fargate, can then consume the secret as usual.



aws

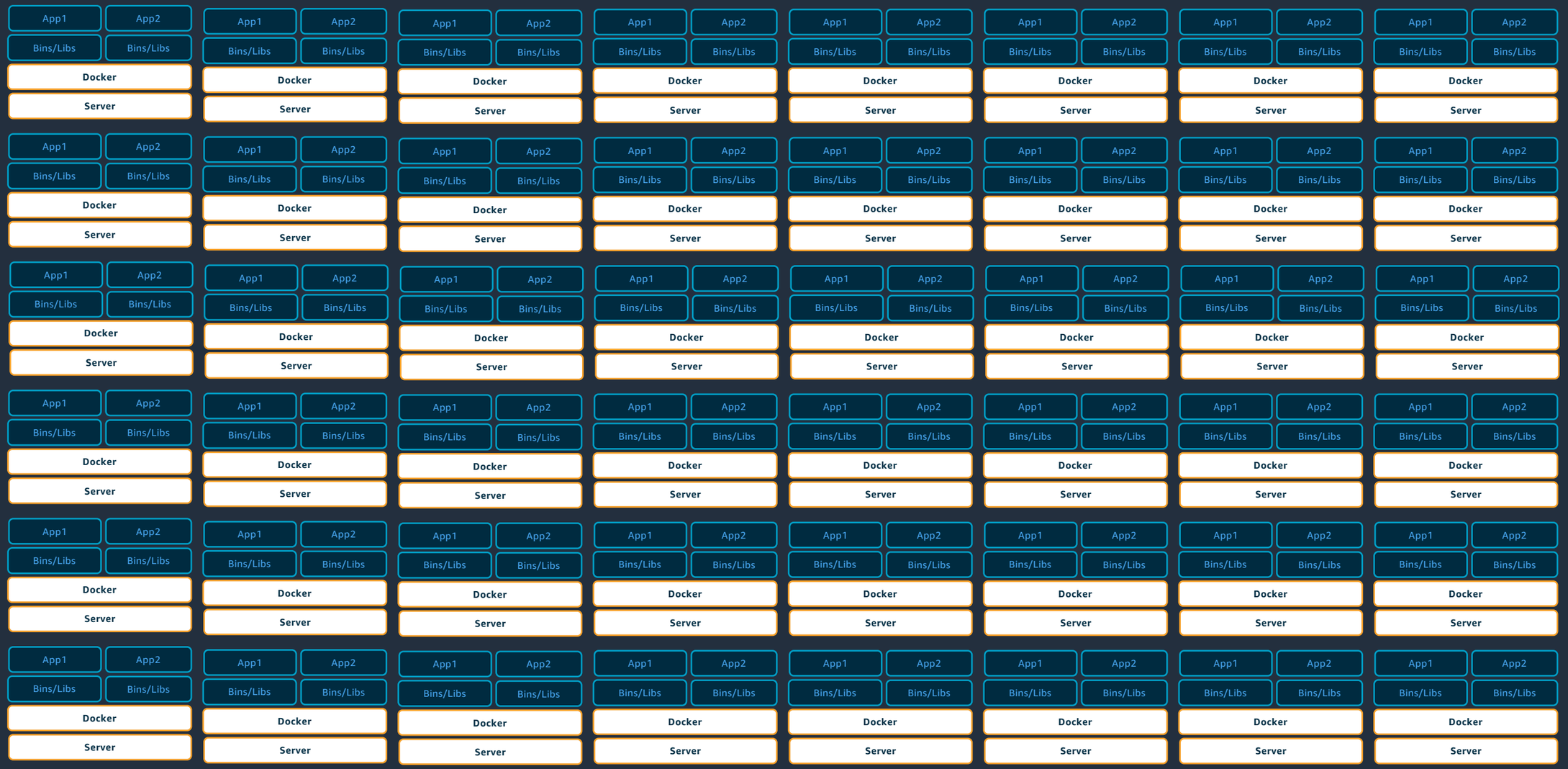
Example of defense-in-depth strategy

Multi-layered approach - AWS, Kubernetes, Container Runtime and Linux Security Controls

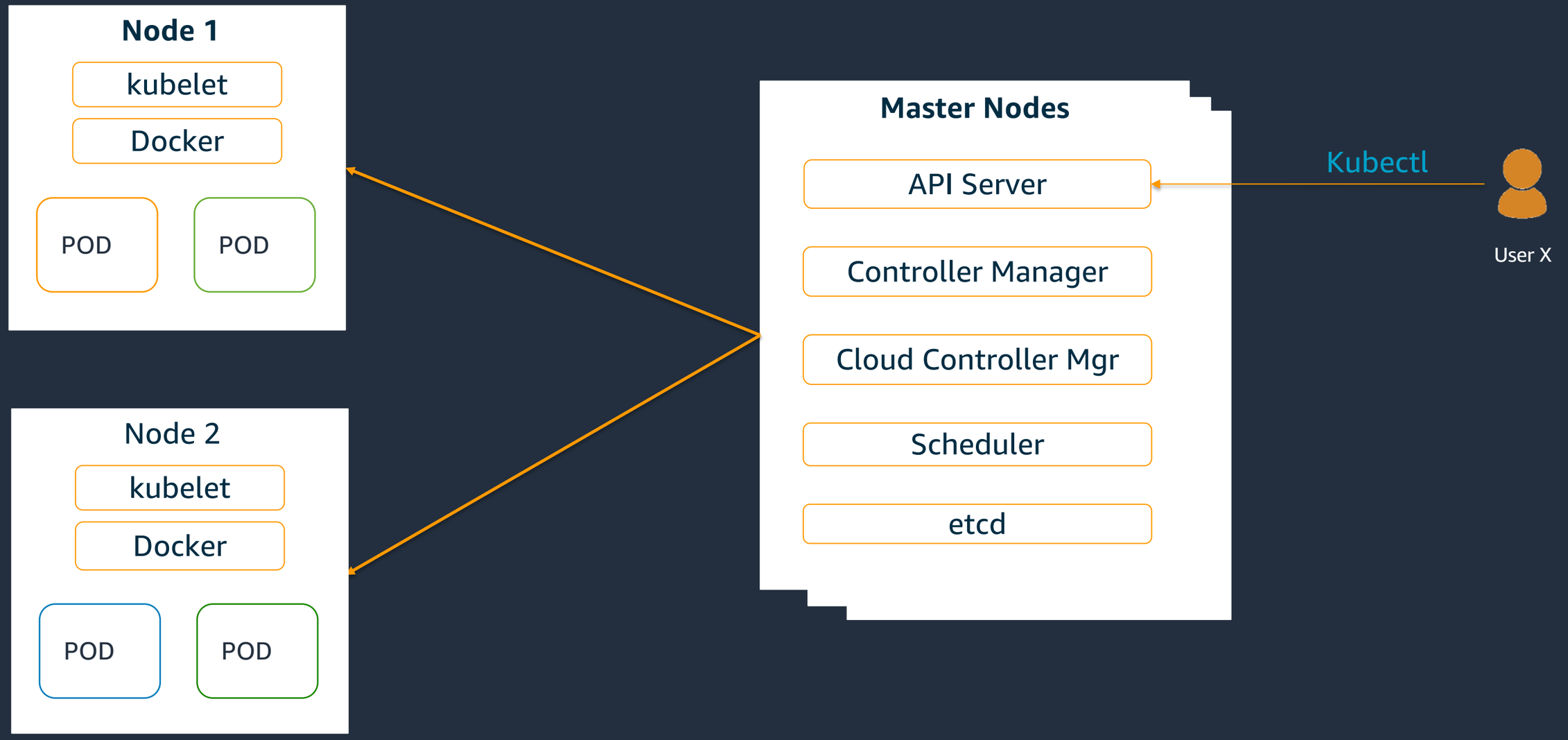


Operations

Containers



Kubernetes Architecture



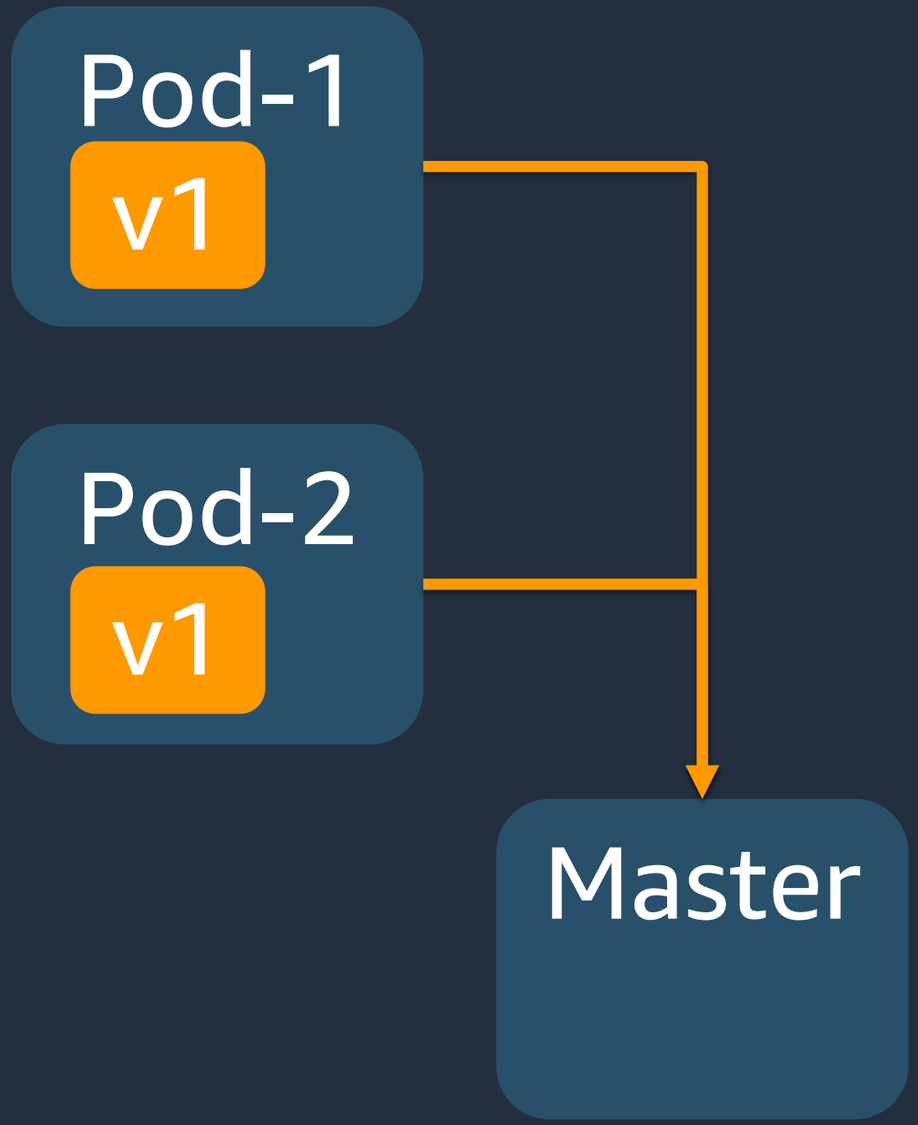
Worker Nodes

Control Plane

aws

What is a Kubernetes Pod?

- Smallest unit of deployment
- Grouping of containers
- Share storage and networking
- Unique IP per pod
- Co-located and co-scheduled on nodes



aws

Kubernetes Core Concepts

Manifest File - YAML/JSON used to deploy Kubernetes objects

Deployment – Run specified # of Pods of your application

Service - Maps a fixed IP address to a logical group of pods

Annotation - Key/Value pairs to hold non-identifying information

Label - Key/Value pair used for association and filtering

DaemonSet - Implements a single instance of a pod on a worker node

aws

Deploying Applications

Application Implementation Overview

- Create required container image & image repo / use existing
- Deploy Pods across Worker Nodes
    - Create pod-manifest.yml file
    - Deploy from the kubectl node: **“kubectl apply –f pod-manifest.yaml”**
- Create an Ingress “Service” to route traffic to the Pods
    - Create svc-manifest.yml file
    - Deploy from the kubectl node: **“kubectl apply –f svc-manifest.yaml”**

aws

Example nginx-pods.yaml

```
…
kind: Deployment
replicas: 2
  template:
    metadata:
      labels:
        app: nginx
spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Create a “ReplicaSet” containing 2 “Pods”

App Name label

Container Image

Listener Port

Implement from kubectl node with one command:

“kubectl apply –f nginx-pods.yaml”

aws

Example nginx-svc.yaml (Classic Load Balancer)

```
…
kind: Service
spec:
 selector:
    app: nginx
  type: LoadBalancer
  ports:
  - name: http
    port: 80
    targetPort: 80
```

Route traffic to Apps named “nginx”

Deploy an AWS Load Balancer

Listener and Target Config

Implement from kubectl node with one command:

“kubectl apply –f. nginx-svc.yaml”

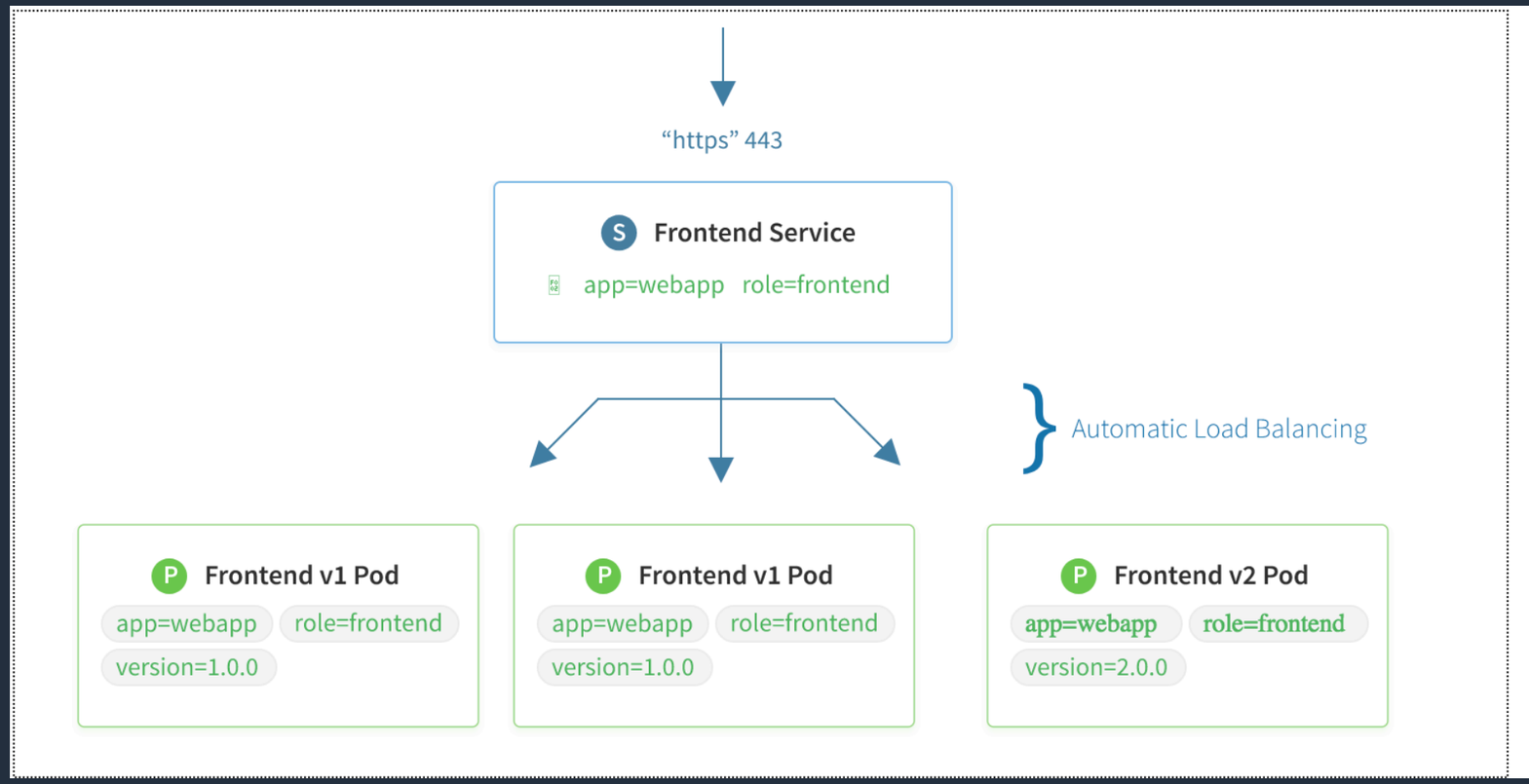
aws

Kubernetes Service Types and Ingress Options

aws

What is a Service?

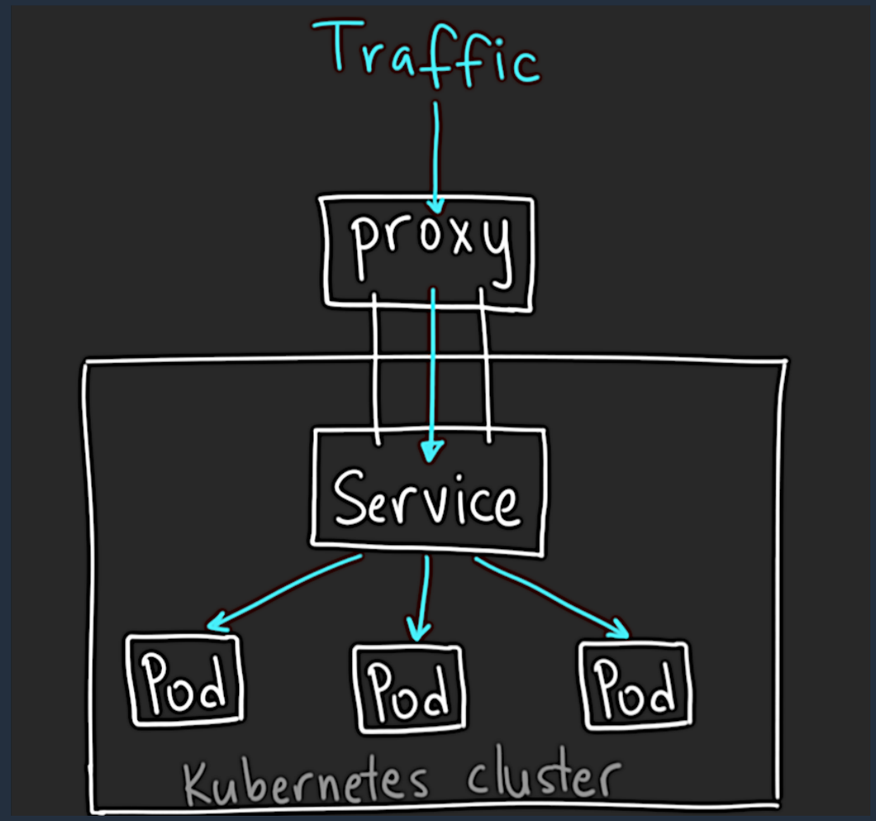
- Grouping of pods.
- Identifies the set of pods using a LabelSelector.
- Services can be exposed in different ways.



aws

ServiceType: ClusterIP

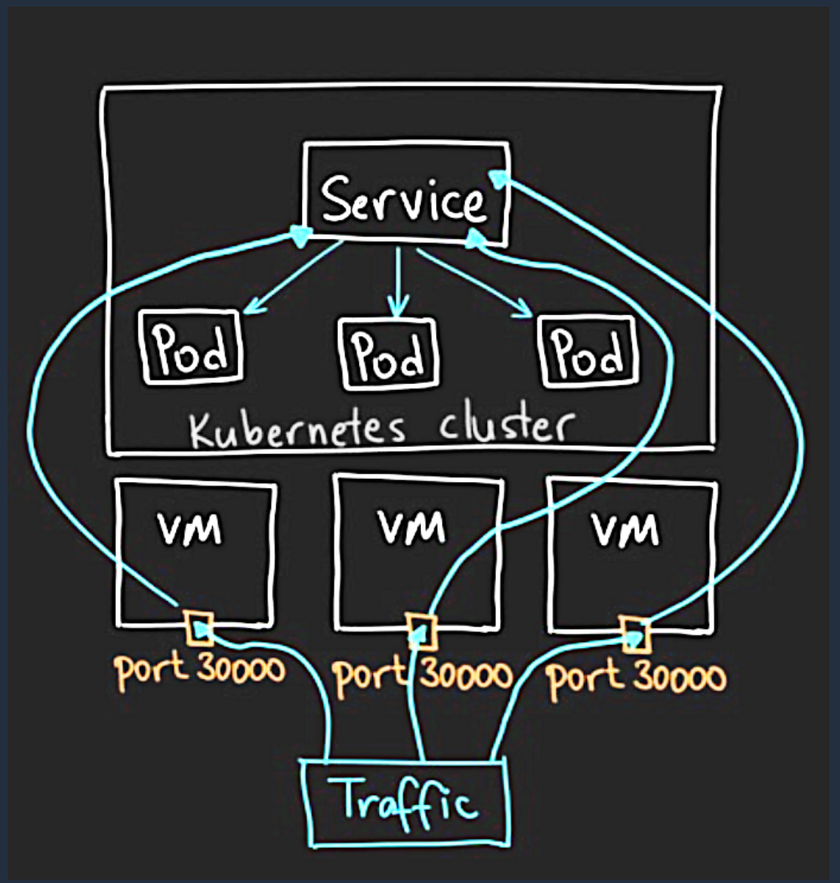
- Exposes the service on a cluster-internal IP
- Only reachable from within the cluster
- Access possible via kube-proxy
- Useful for debugging services, connecting from your laptop or displaying internal dashboards



aws

ServiceType: NodePort

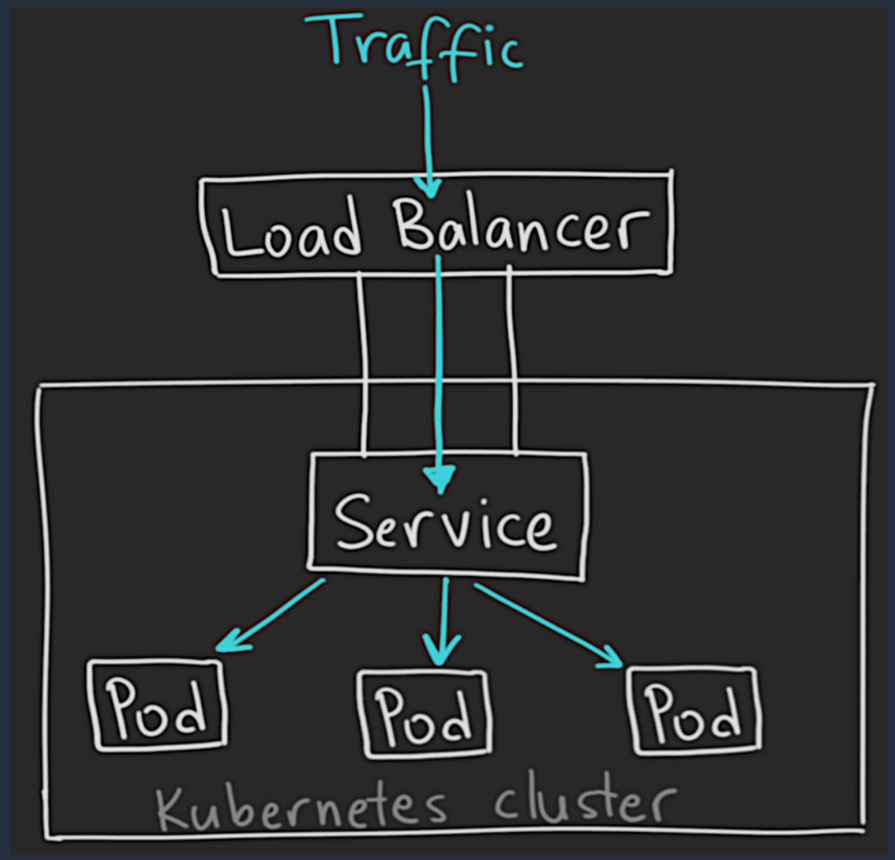
- Exposes the service on each Node's IP at a static port
- Routes to a ClusterIP service, which is automatically created.
- From outside the cluster: <NodeIP>:<NodePort>
- 1 service per port
- Uses ports 30000-32767



aws

ServiceType: LoadBalancer

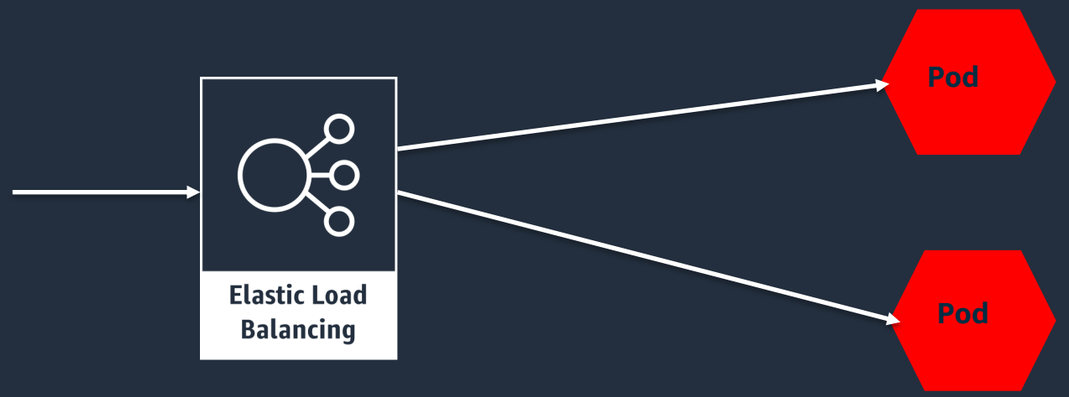
- Exposes the service externally using a cloud provider's load balancer
- NodePort and ClusterIP services (to which LB will route) automatically created
- Each service exposed with a LoadBalancer (ELB or NLB) will get its own IP address
- Exposes L4 (TCP) or L7 (HTTP) services



aws

Implement Kubernetes service by CLB

```
apiVersion: v1
kind: Service
metadata:
 name: nginx
 namespace: default
 annotations: {}
spec:
 ports:
 - name: http
   port: 80
   protocol: TCP
   targetPort: 80
 selector:
   app: nginx
type: LoadBalancer
```



aws

Implement Kubernetes service by NLB

```
apiVersion: v1
kind: Service
metadata:
 name: nginx
 namespace: default
 annotations:
   service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
spec:
 ports:
 - name: http
   port: 80
   protocol: TCP
   targetPort: 80
 selector:
   app: nginx
type: LoadBalancer
```


aws

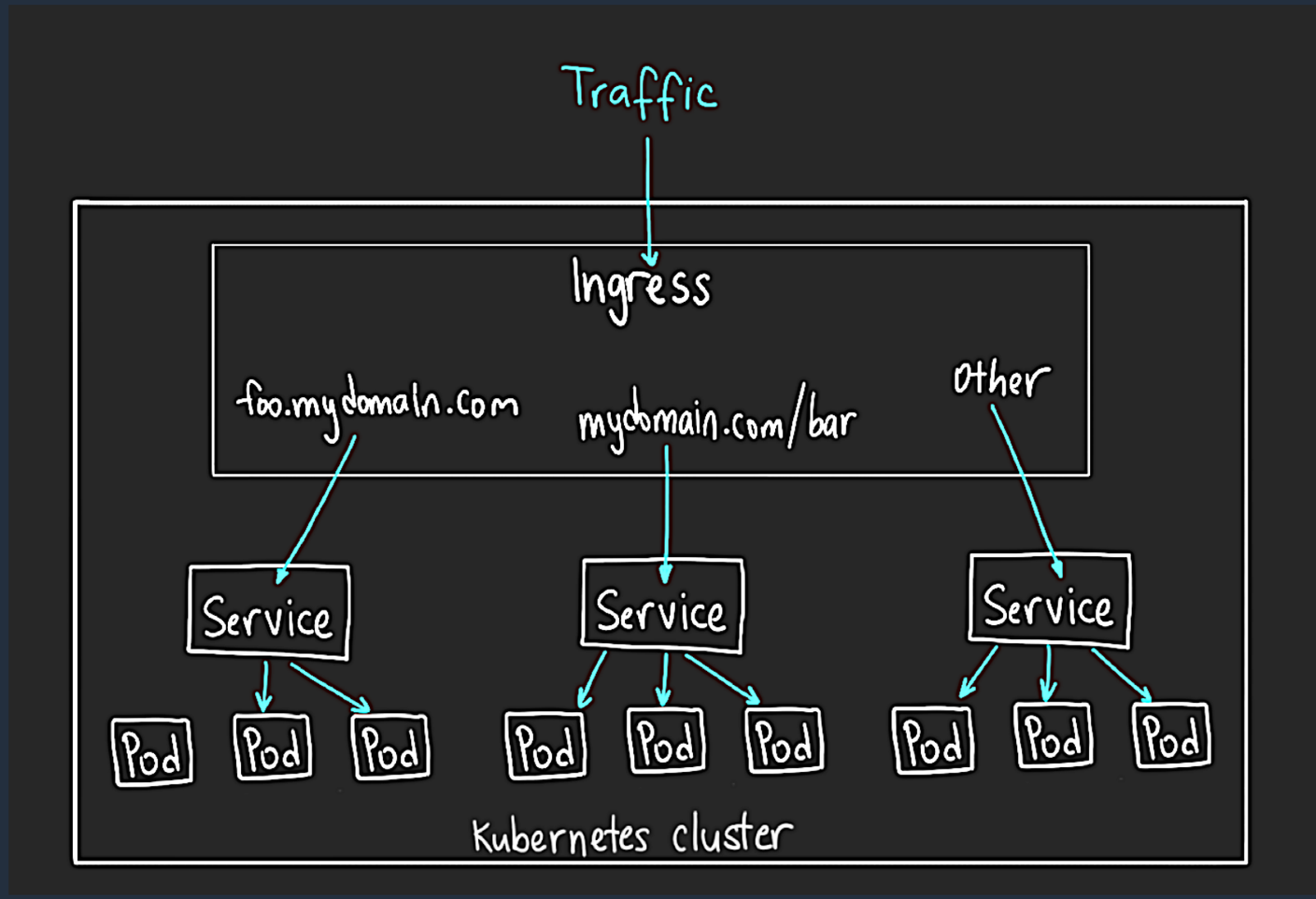
Using annotations to specify additional config (SSL)

```
…
kind: Service
metadata:
 annotations:
    service.beta.kubernetes.io/aws-load-balancer-ssl-cert: "arn:aws:acm:...."
    service.beta.kubernetes.io/aws-load-balancer-ssl-ports: "*"
    service.beta.kubernetes.io/aws-load-balancer-backend-protocol: "http”
spec:
    …
```


aws

Ingress Object

- Exposes HTTP/HTTPS routes to services within the cluster
- Implemented using an Ingress Controller.
- Many implementations: ALB, Nginx, F5, HAProxy etc



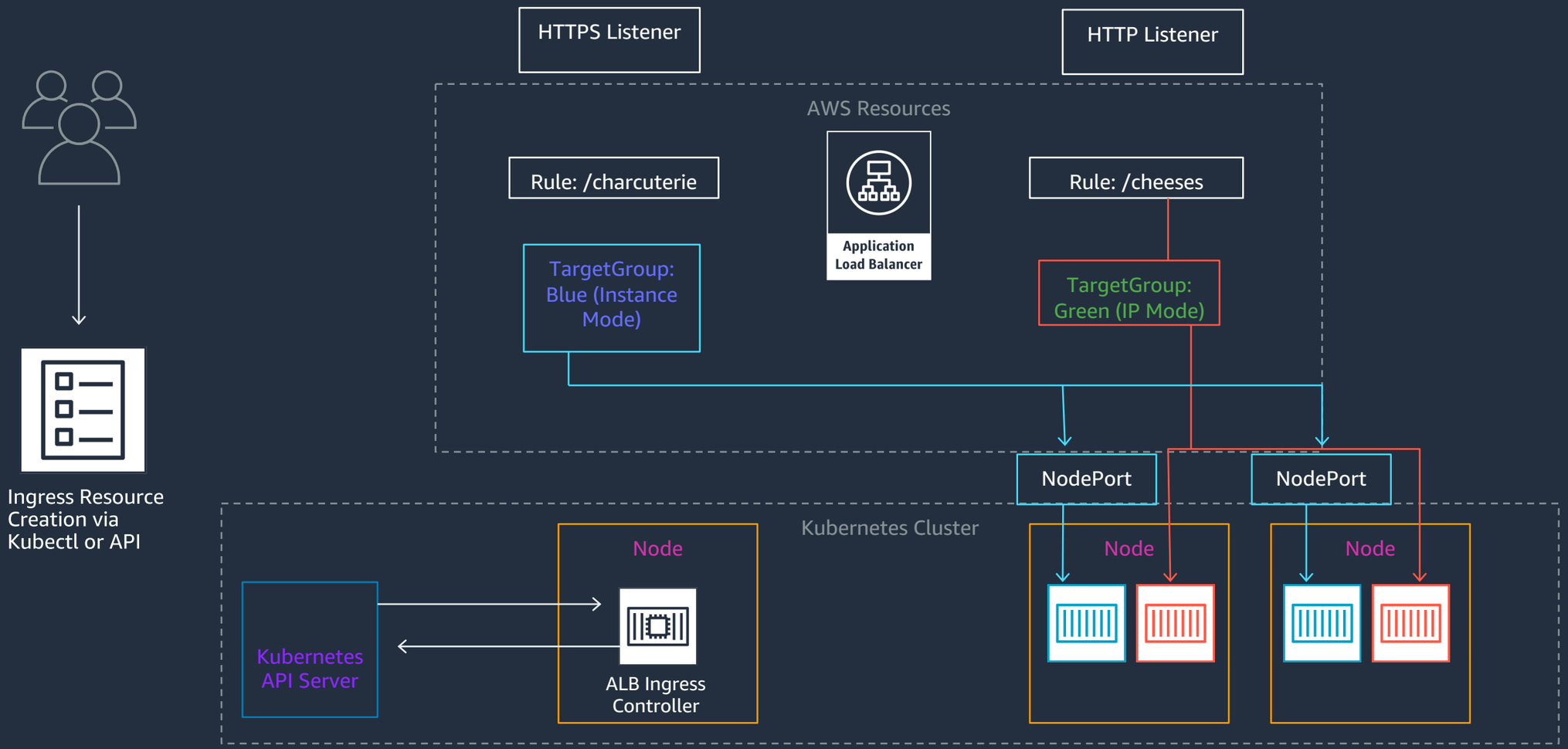
aws

Ingress in Kubernetes

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: echoserver
  namespace: echoserver
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/subnets:  subnet-0a611a4986dd598ce ,subnet-024888df2852fa4ae
    alb.ingress.kubernetes.io/tags: Environment=dev,Team=test
spec:
  rules:
    - host: echoserver.example.com
      http:
        paths:
          - path: /
            backend:
              serviceName: echoserver
              servicePort: 80
```


aws

ALB Ingress Controller



https://aws.amazon.com/blogs/opensource/kubernetes-ingress-aws-alb-ingress-controller/

AWS Load Balancer Support

Classic and Network Load Balancer:

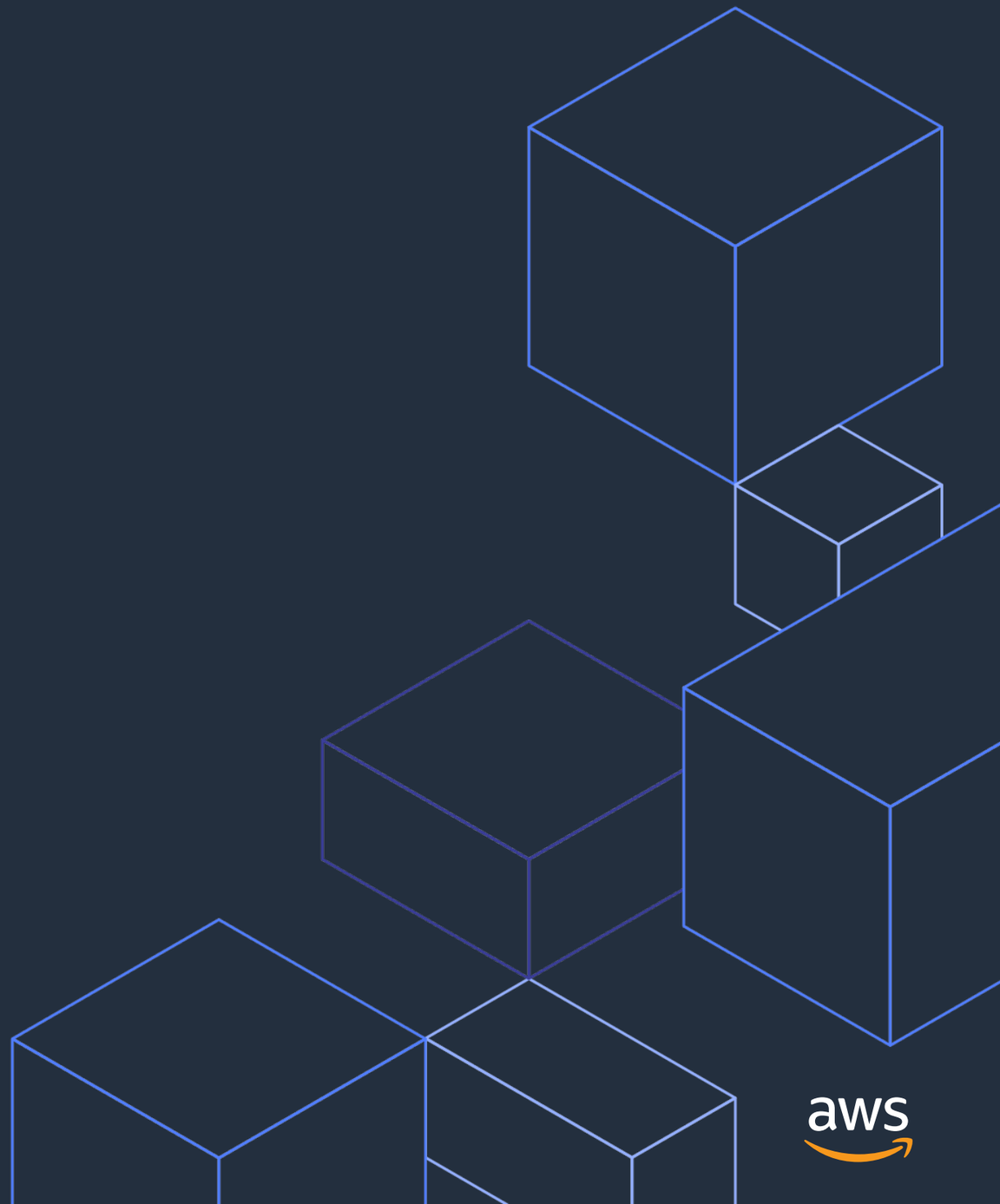
Fully Integrated out of the box.
Deployed using simple Kubernetes manifests

Application Load Balancer:

Integrated using the "ALB Ingress Controller" extension
https://github.com/kubernetes-sigs/aws-alb-ingress-controller

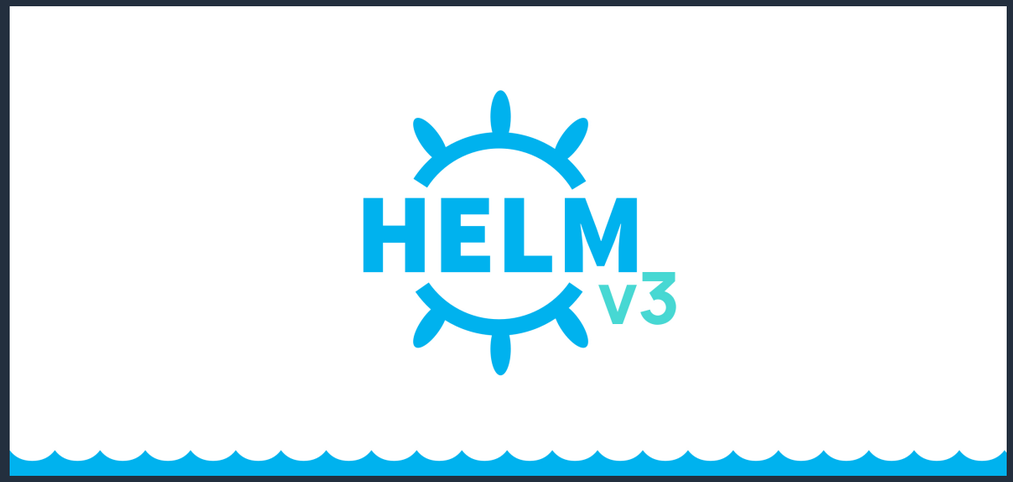
aws

Operations



What is Helm?

- Helm is *the* package manager for Kubernetes
- Inspired by tools like Homebrew, apt, npm
- Maintained by the CNCF
- Consists of two components
    - **Helm** – client component
    - **Tiller** – in-cluster server component



aws

Helm - Why should I care?

- Easier to manage all objects required for applications (Services, Pods, etc.)
- Easier to manage all the version / configurations of applications too
- Reduce the leaning curve associated with deploying production-grade
- Take the pain out of updates with in-place upgrades
- Reduce the complexity by providing sane defaults and exposing parameters in a consistent way

- For example, mysql helm package would create below

  All required Service accounts, secrets, service, configMaps, pvc, deployment, etc required for running mysql pods in the cluster

aws

Basic Helm Terminology

- **Charts** – a Helm package that contains all the resource definitions necessary to run an application, tool or service in a Kubernetes cluster

- **Repository** – the place where charts can be collected and shared

- **Release** – an instance of a chart running in a kubernetes cluster

- **Values** – provide a way to override template defaults with your own information

aws

Health checks

- Liveness probes
  - Don’t use unless you fully understand the consequences
  - Don’t use the same settings as the readiness probe
  - Don’t have external dependencies, e.g. a database connection
  - Avoid using exec probes
- Readiness probes
  - Use with microservices that provide an HTTP endpoint
  - Use the traffic port for the readiness probe
- Used together with Kubernetes services to determine the set of healthy endpoints

Liveness example

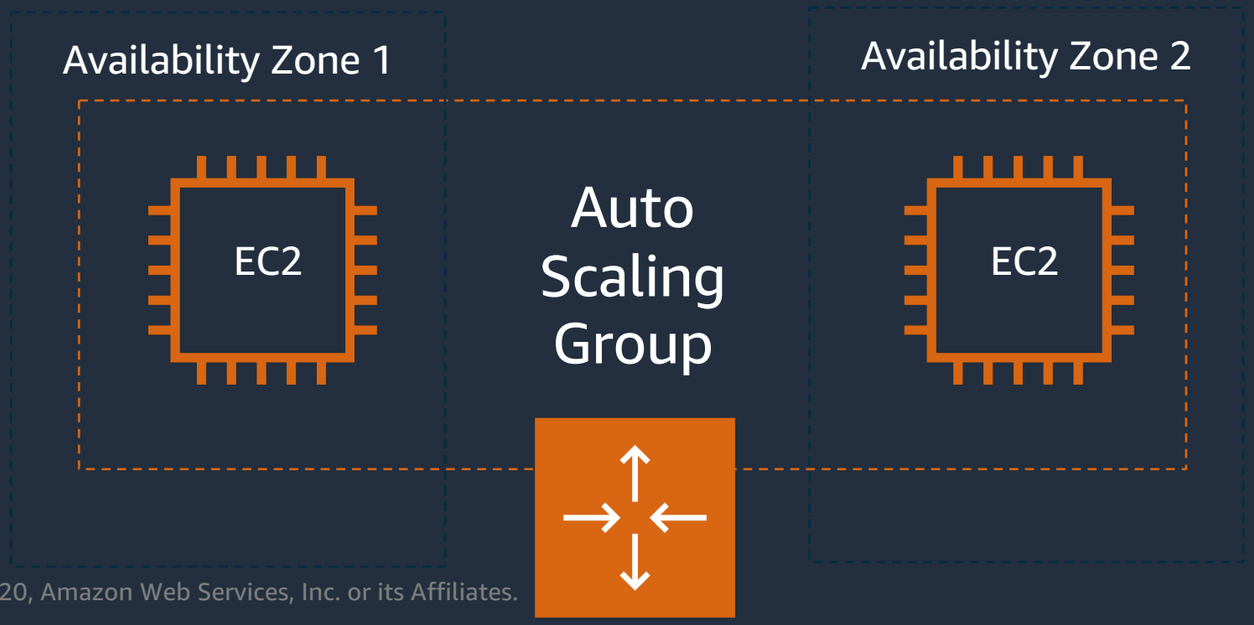
```
spec:
  containers:
  - image: my-app:1.0
  livenessProbe:
    httpGet:
      path: /healthz
      port: 8080
  initialDelaySeconds: 30
```

Readiness examples

```
spec:
  containers:
  - image: my-app:1.0
  readinessProbe:
    exec:
      Command: [“stat”, “/var/run/file”]
```

Data Plane Scaling

- Integration called Cluster Autoscaler for AWS
- Deployed as daemonset in kube-system namespace
- Scale up/down unschedulable / under-utilized
- Create policy with access to `SetDesiredCapacity`




```
...
resources:
    limits:
        memory: 200Mi
    requests:
        memory: 100Mi
...
```

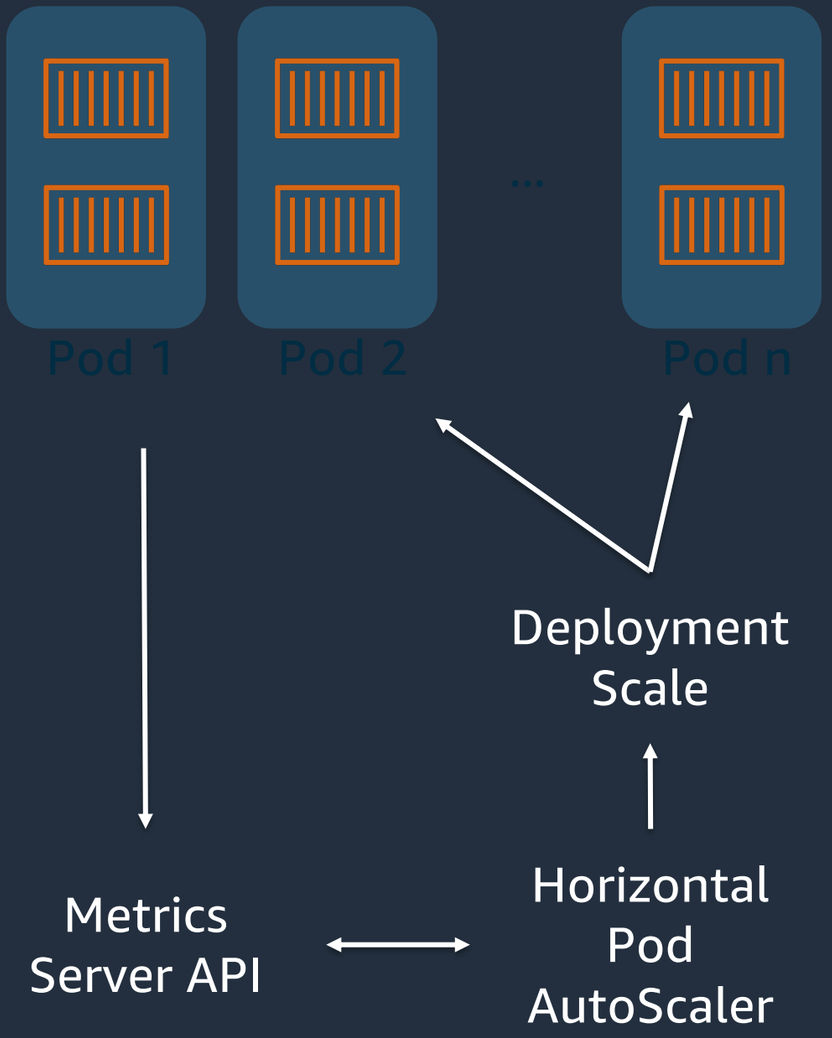

aws

Horizontal Pod AutoScaler (HPA)

- Scaling # pods based on CPU metrics
- Custom metrics supported such as
    - Inbound Number Connections

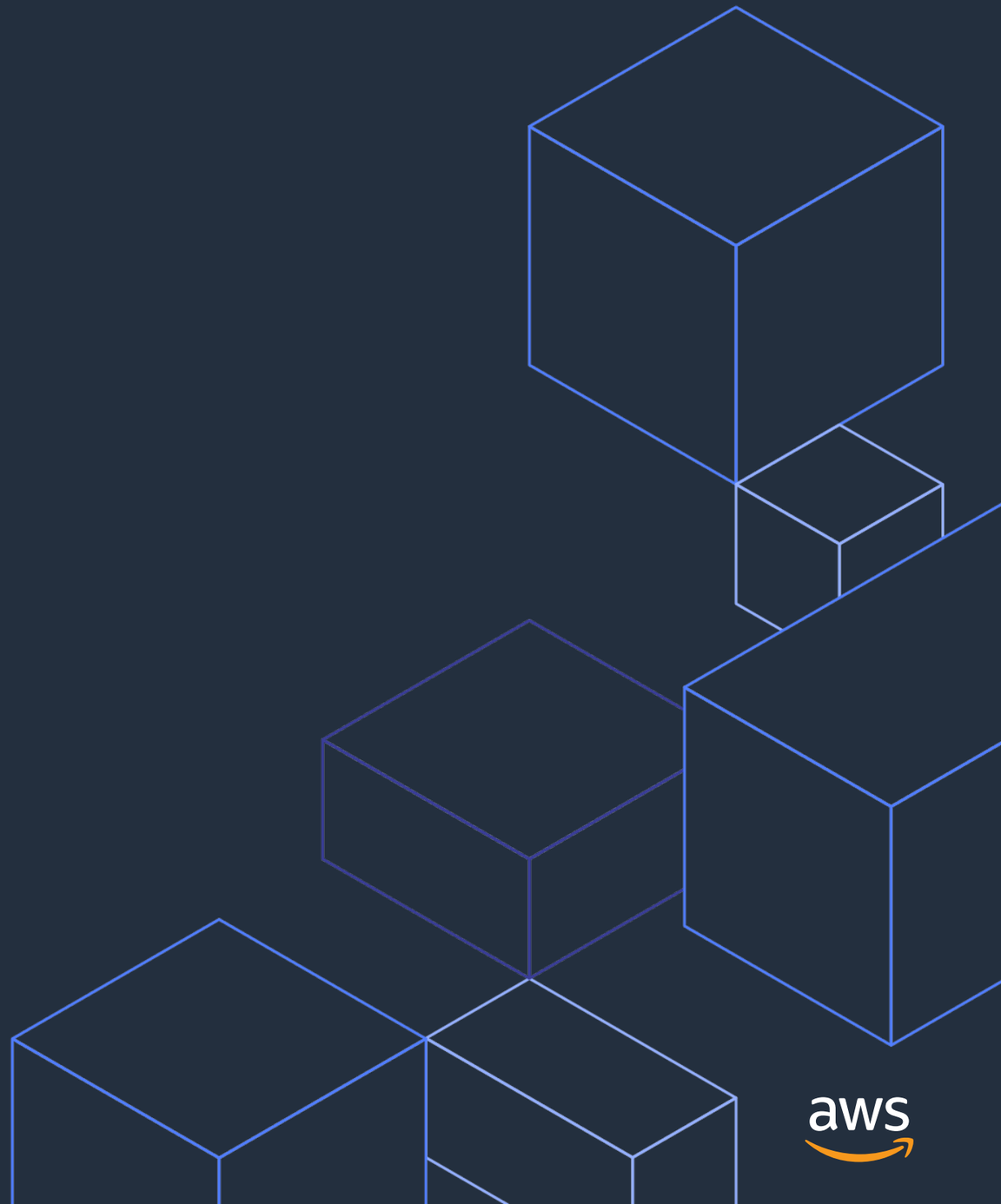
```
kubectl run php-apache --
image=k8s.gcr.io/hpa-example --
requests=cpu=200m --expose --
port=80
```

```
kubectl autoscale deployment php-
apache --cpu-percent=50 --min=1 -
-max=10
```

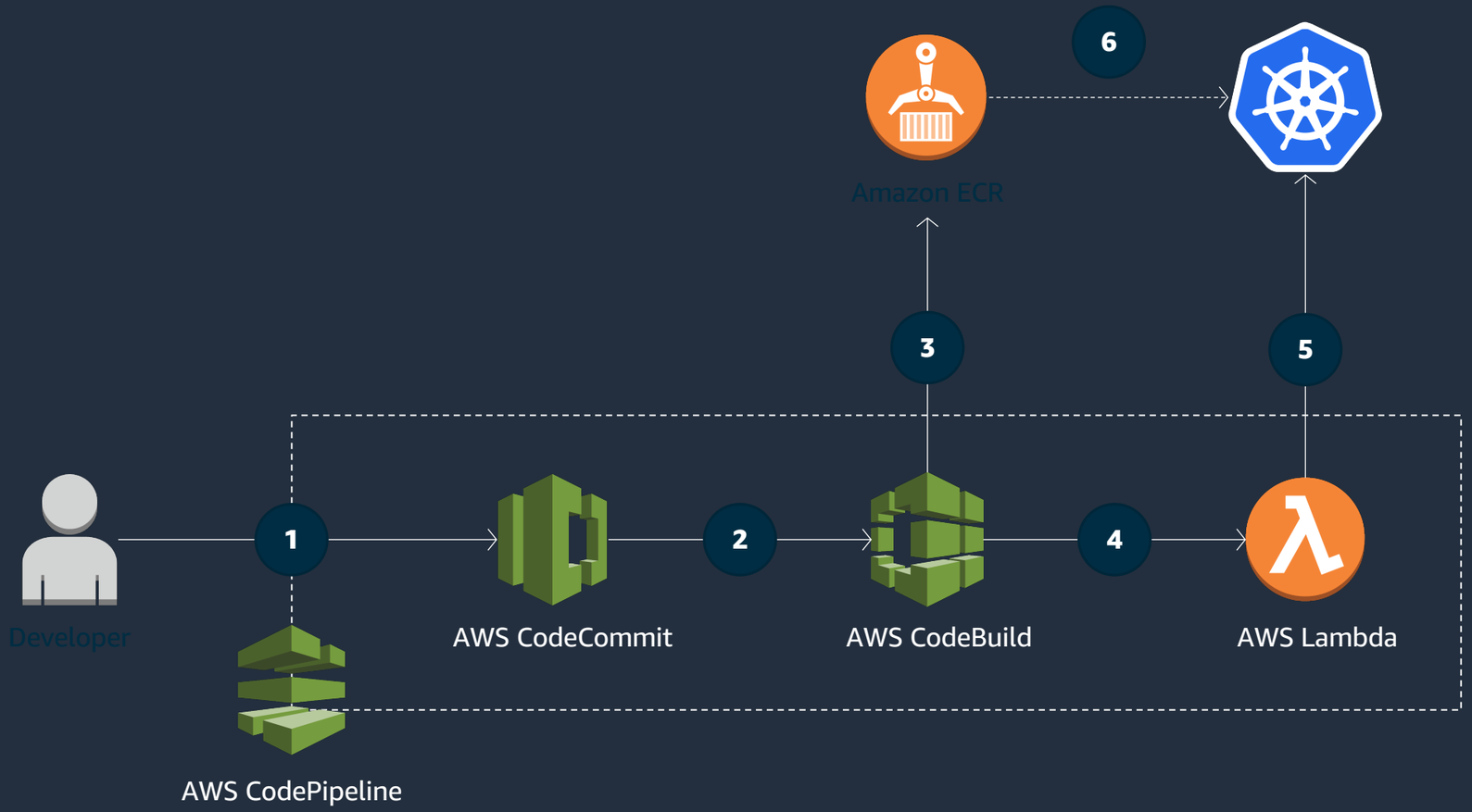


aws

DevOps



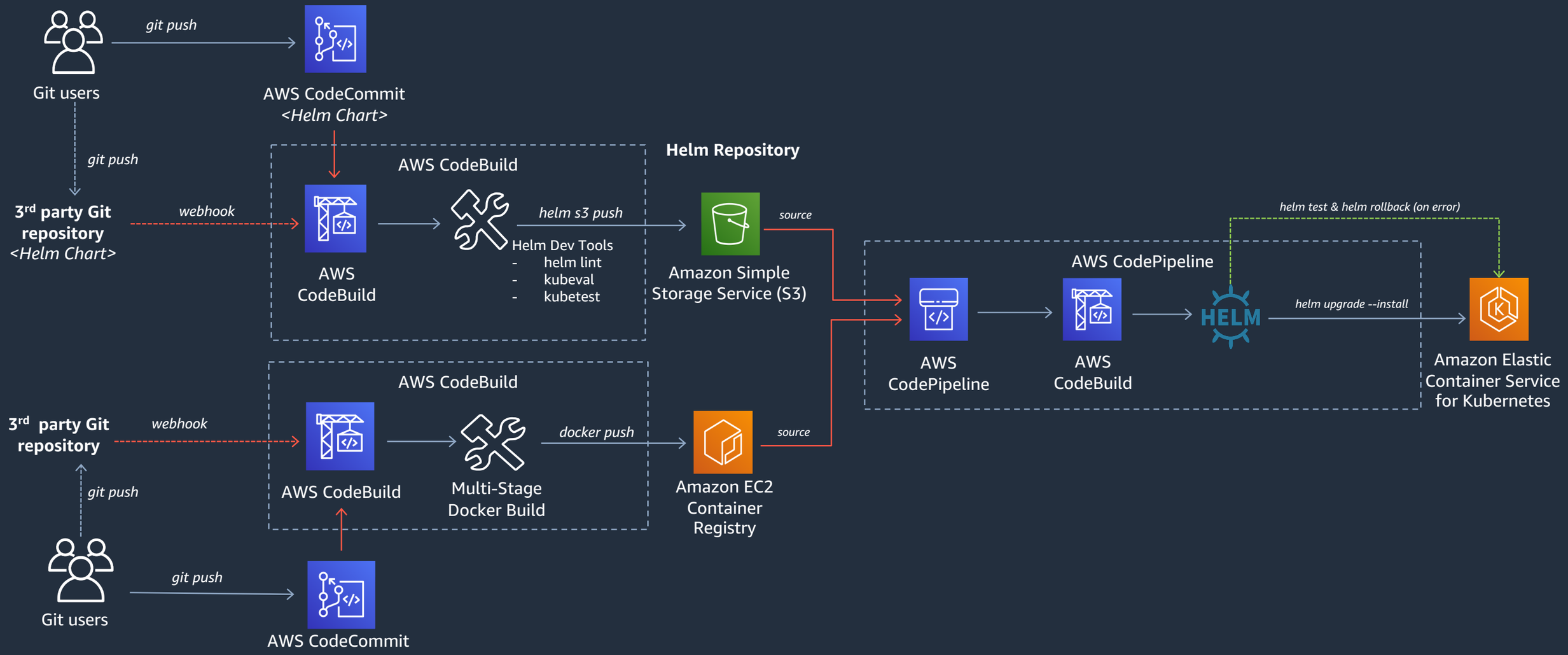
Kubernetes continuous deployment



1. Developers continuously integrate changes into a main branch hosted within a repo
2. Triggers an execution of the pipeline when a new version is found, builds a new image with build id
3. Pushes the newly built image tagged with build id to ECR repo
4. Invokes a Lambda function to trigger application deployment
5. Leverages Kubernetes Python SDK to update a deployment
6. Fetches new container image and performs a rolling update of deployment

aws

EKS Continuous Deployment pipeline



GitOps...?

GitOps is a paradigm or a set of practices that empowers developers to perform tasks which typically fall under the purview of IT operations. **GitOps** requires us to describe and observe systems with declarative specifications that eventually form the basis of continuous everything.

aws

GitOps vs Traditional CI/CD

In a traditional CI/CD pipeline, CD is an implementation extension powered by the continuous integration tooling to promote build artifacts to production. In the GitOps pipeline model, any change to production must be committed in source control (preferable via a pull request) prior to being applied on the cluster. If the entire production state is under version control and described in a single Git repository, when disaster strikes, the whole infrastructure can be quickly restored without rerunning the CI pipelines.

- Weaveworks Flux
- Intuit ArgoCD
- Jenkins-X
- Many more..

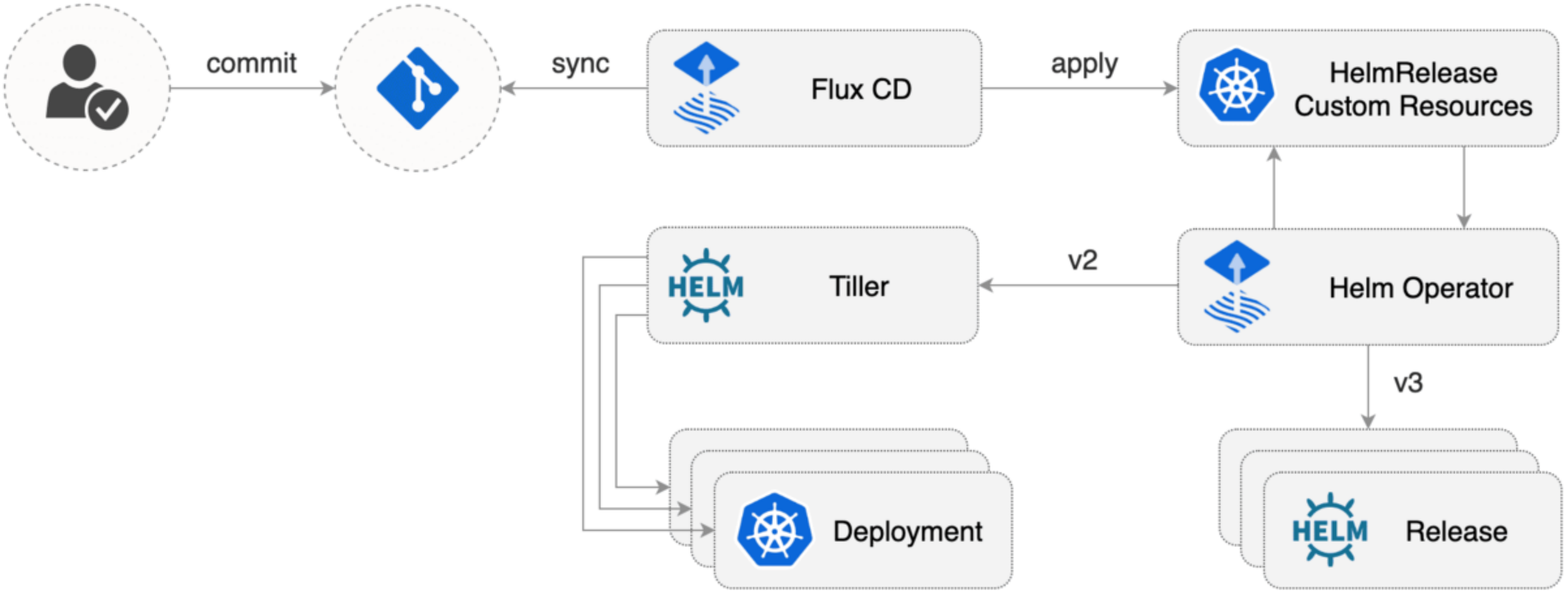
aws

GitOps Benefits

- Stored history of environment changes
- Easy rollback to a previous state
- Secure deployments
- Lightweight approval process
- Modular architecture
- Tool-independent architecture
- Reuse existing knowledge
- Compare different environments
- Backups come out of the box
- Testing your changes like app code
- Highly available deployment infrastructure

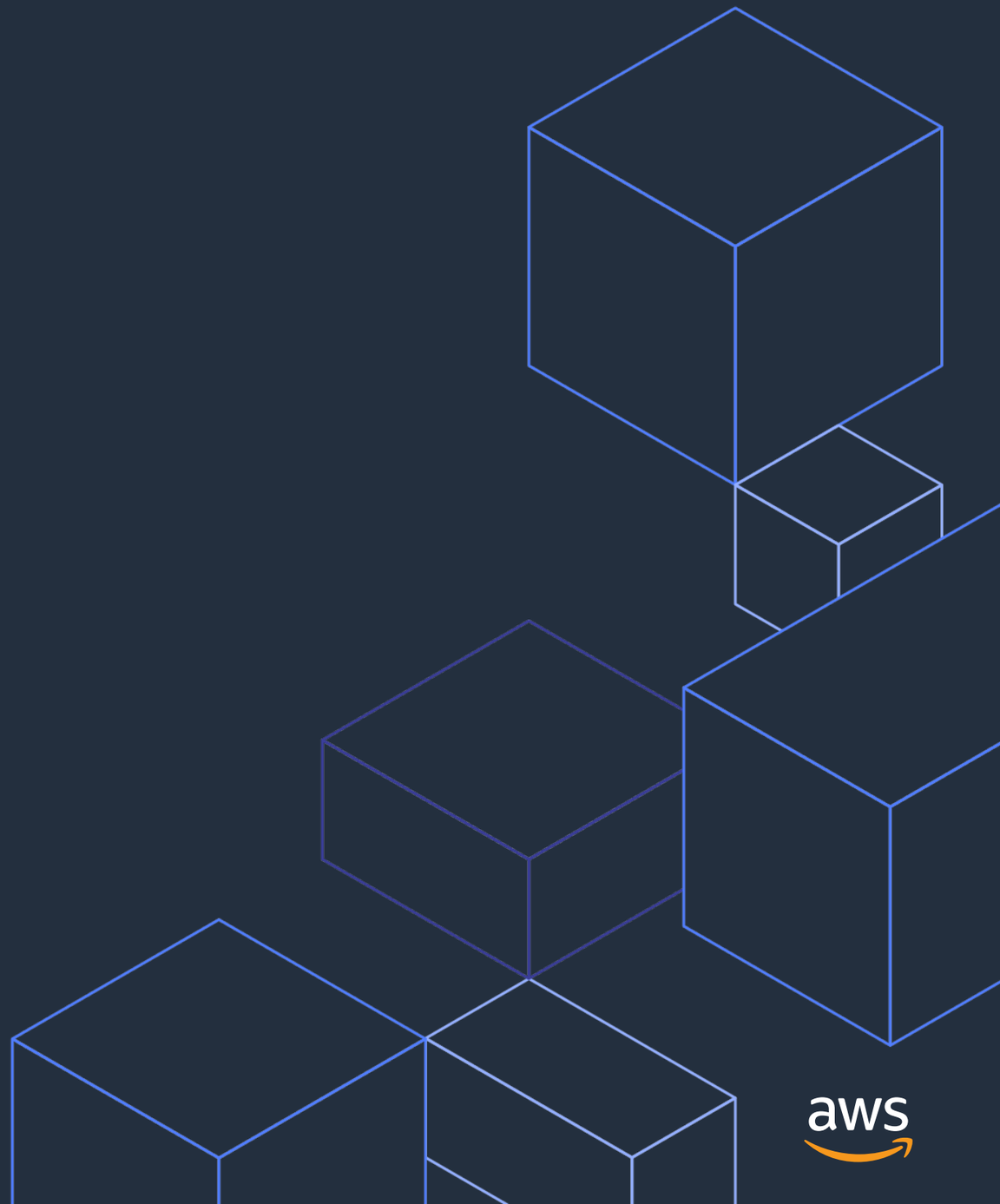
aws

GitOps

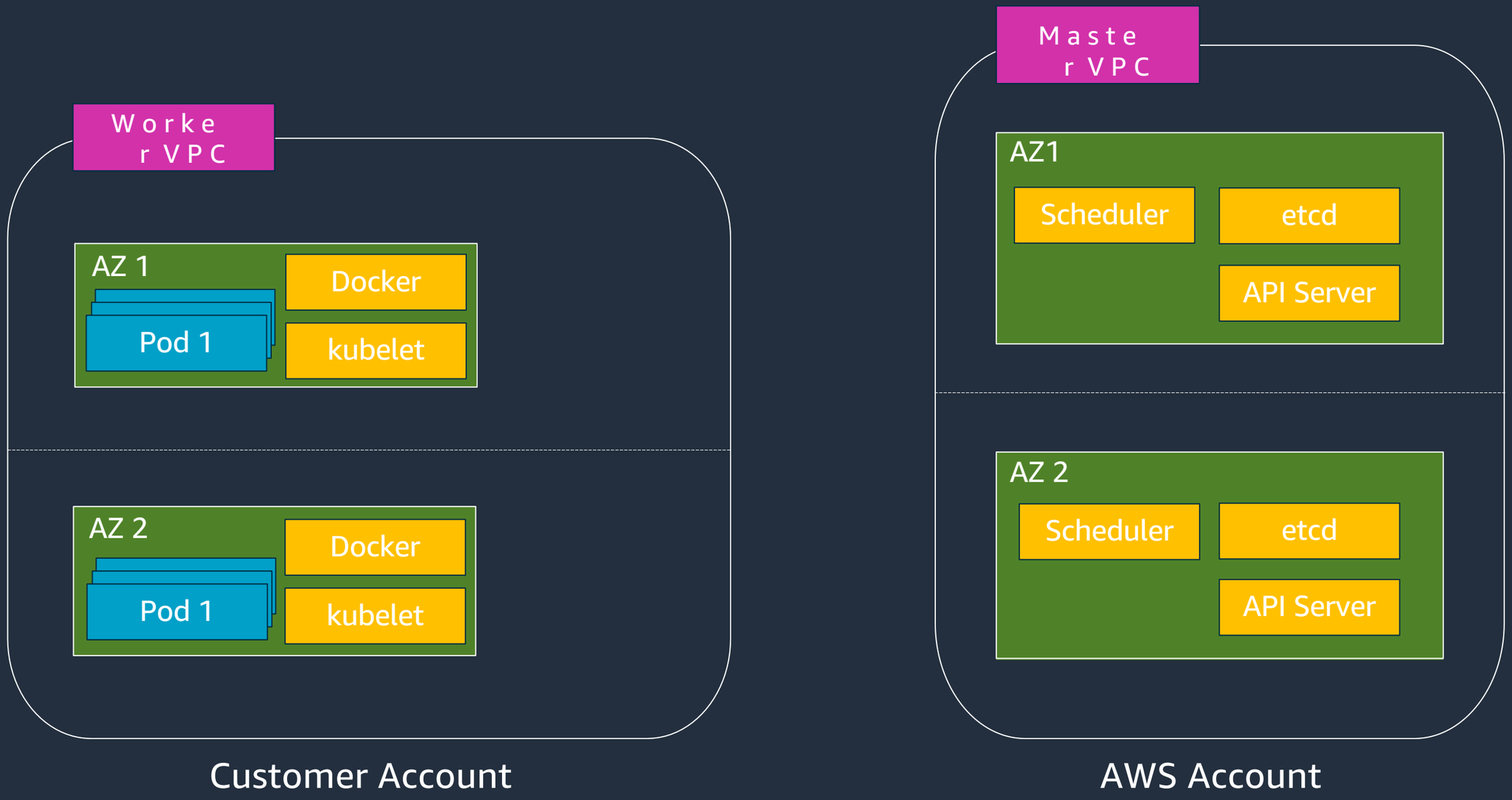


aws

Networking

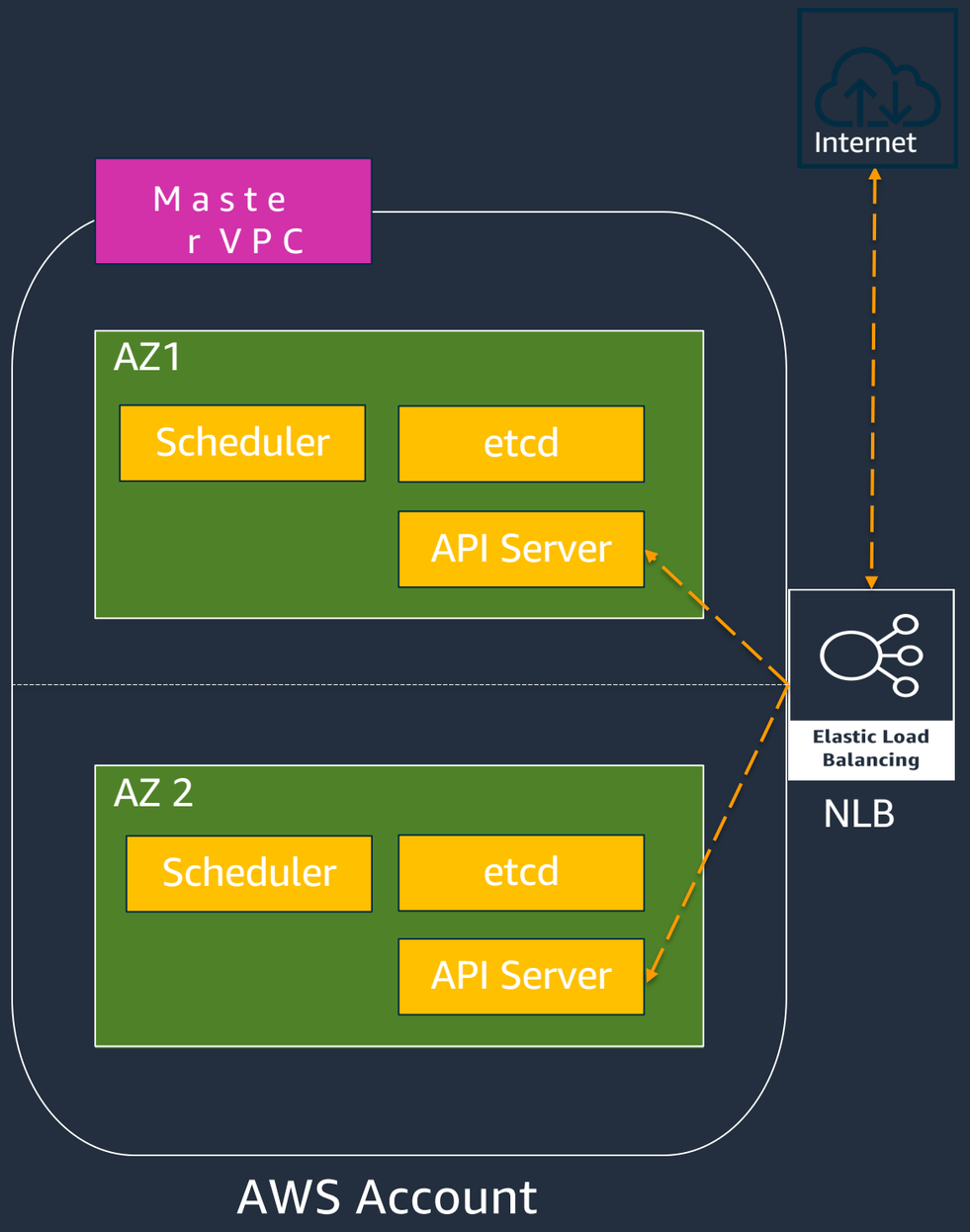


EKS Architecture

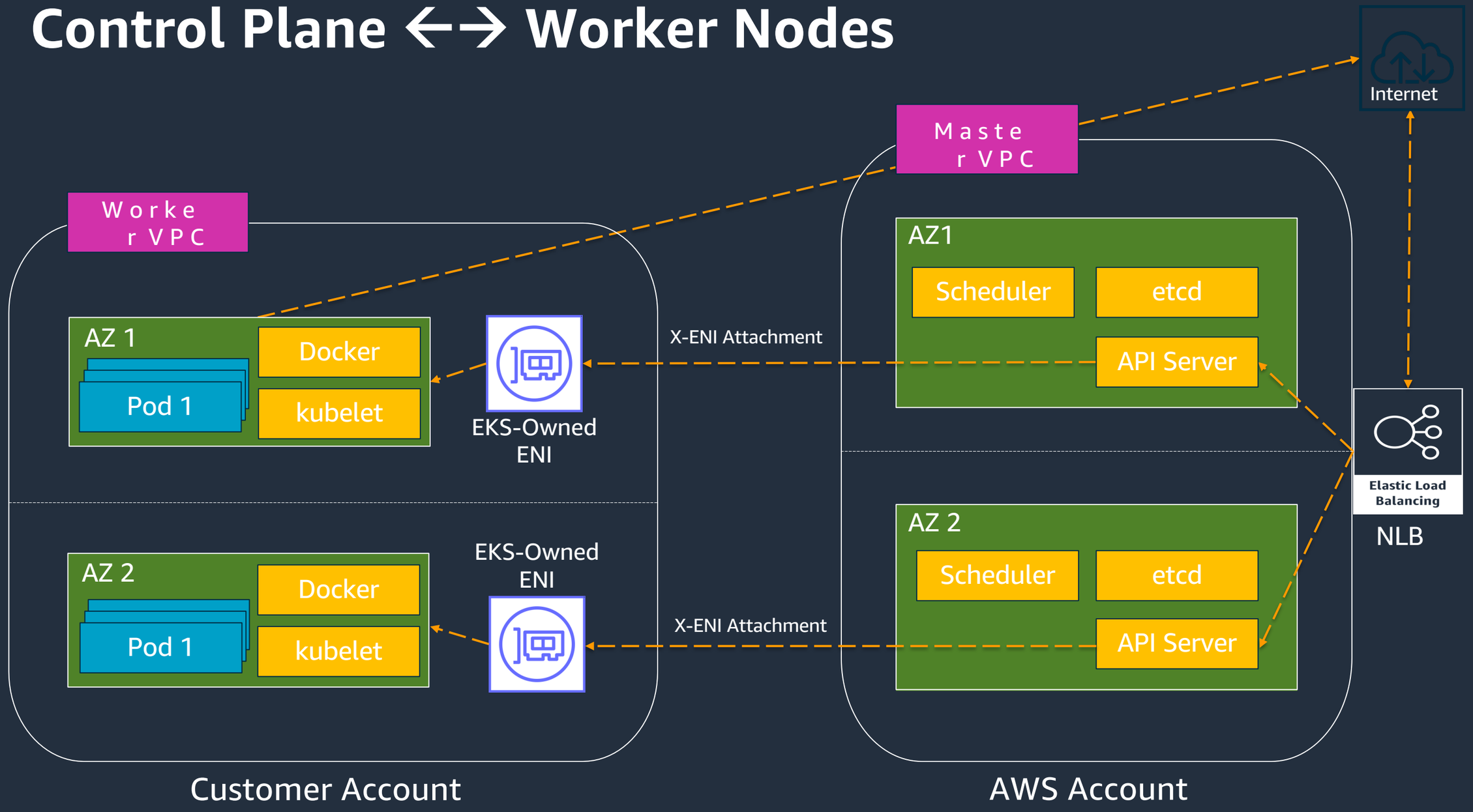


aws

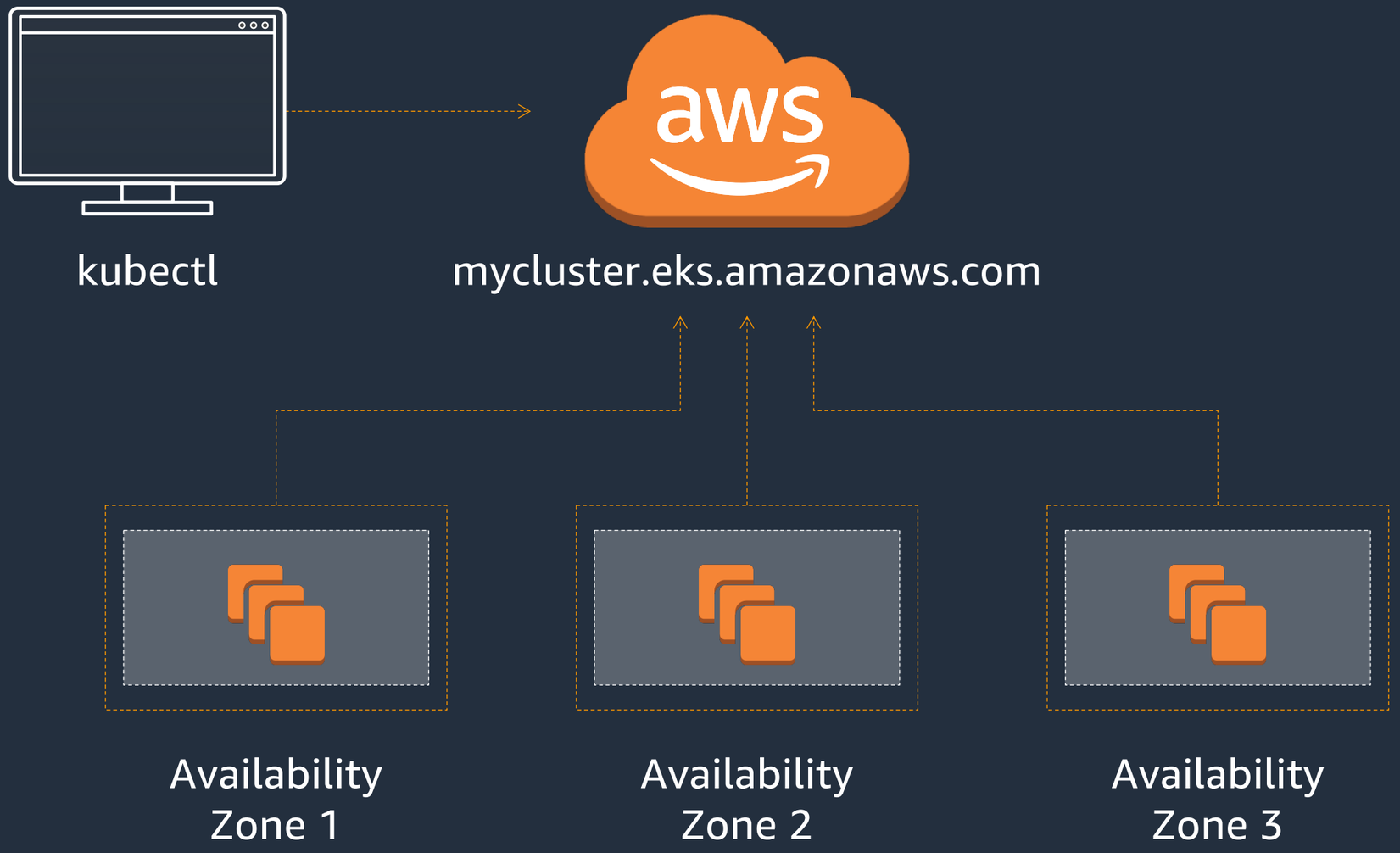
Control Plane Networking



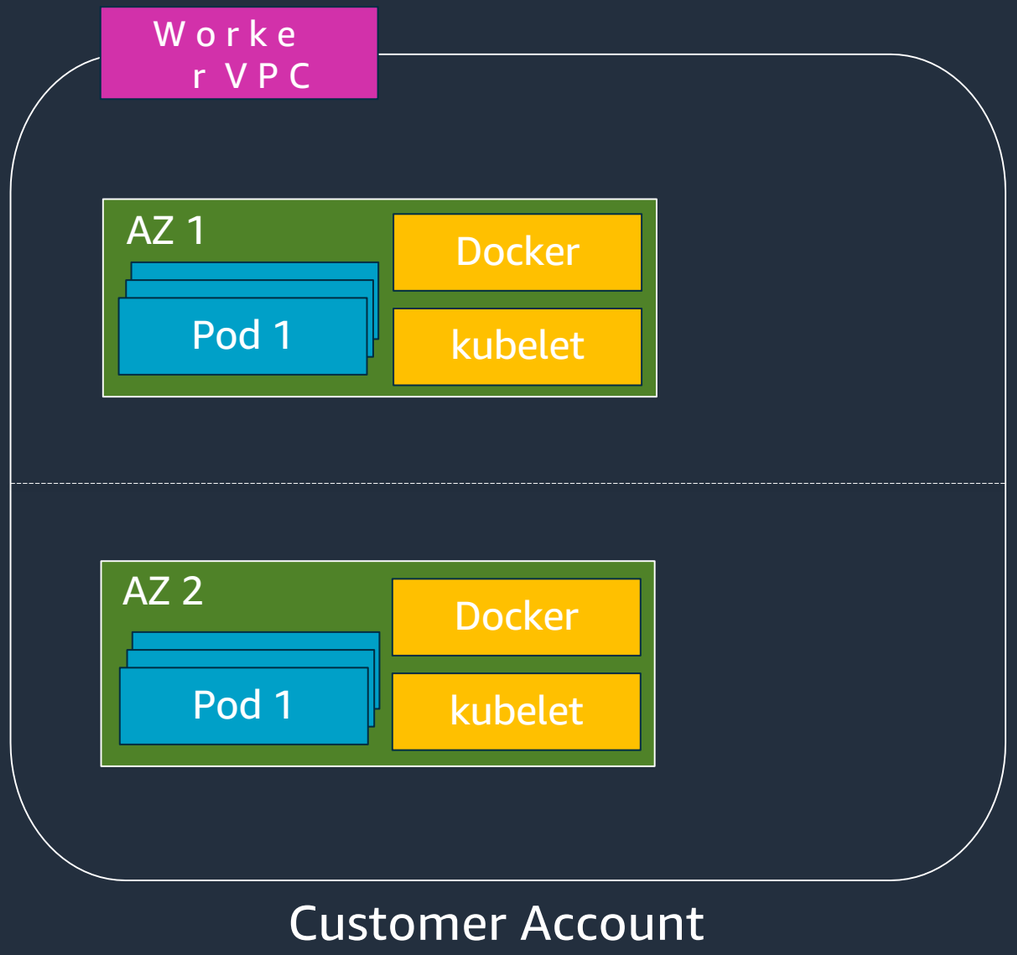
Control Plane ←→ Worker Nodes



Kubernetes Endpoint Private Access



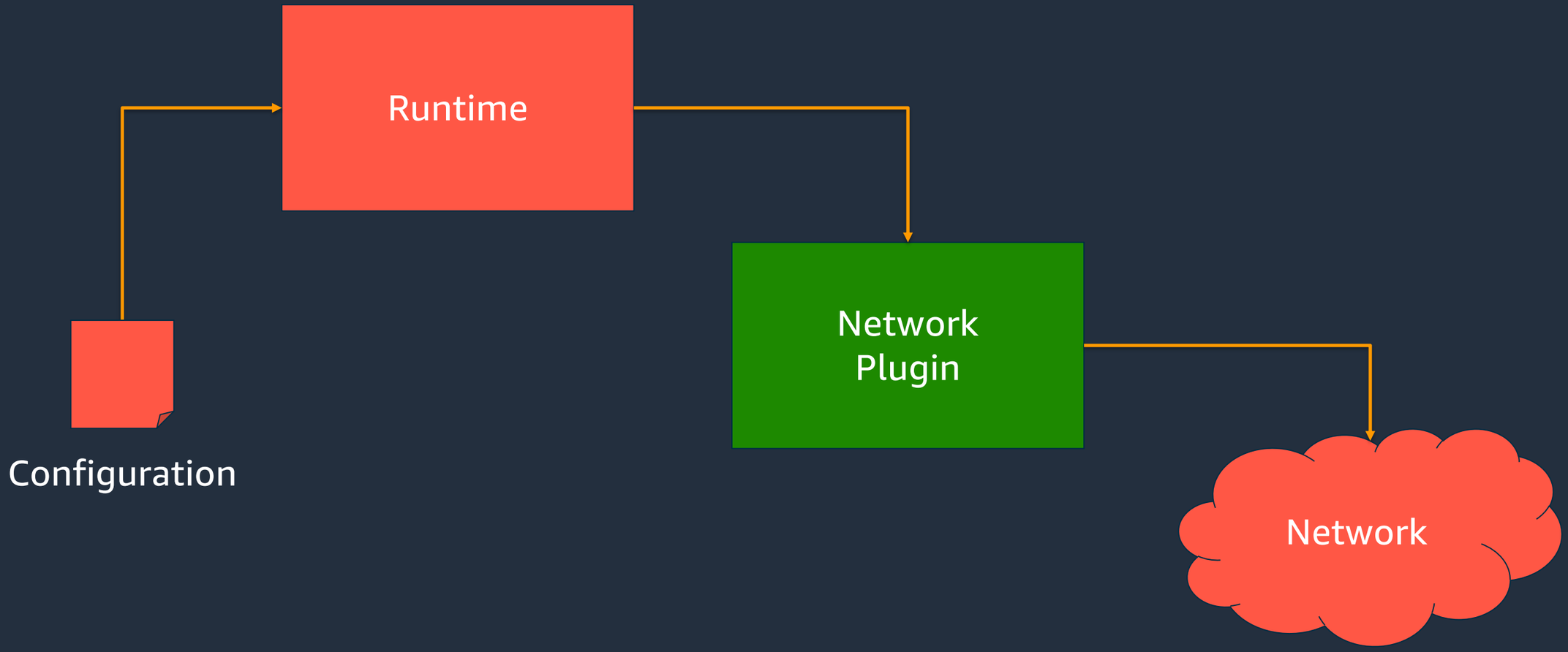
Kubernetes Network Requirements



- All containers can communicate with all other containers without NAT
- All nodes can communicate with all containers (and vice-versa) without NAT
- The IP address that a container sees itself as is the same IP address that others see it as

aws

Container Network Interface (CNI)



aws

Amazon VPC CNI Plugin Goals

1. Simplify networking options for customers

2. Support high throughput, high availability, low latency and minimal jitter

3. Allow customers to reuse AWS VPC networking and security best practices such as use of:
    - VPC flow logs for troubleshooting and compliance auditing
    - VPC routing polices for traffic engineering
    - Security groups for isolation and regulatory requirements

4. Setup Pod networking within seconds

5. Support cluster scale to a minimum of 5000+

aws

Amazon VPC CNI Plugin

Native VPC networking with CNI plugin

Pods have the same VPC address inside the pod as on the VPC

Simple, secure networking

Open source and on Github

https://github.com/aws/amazon-vpc-cni-k8s

EC2 and VPC Considerations

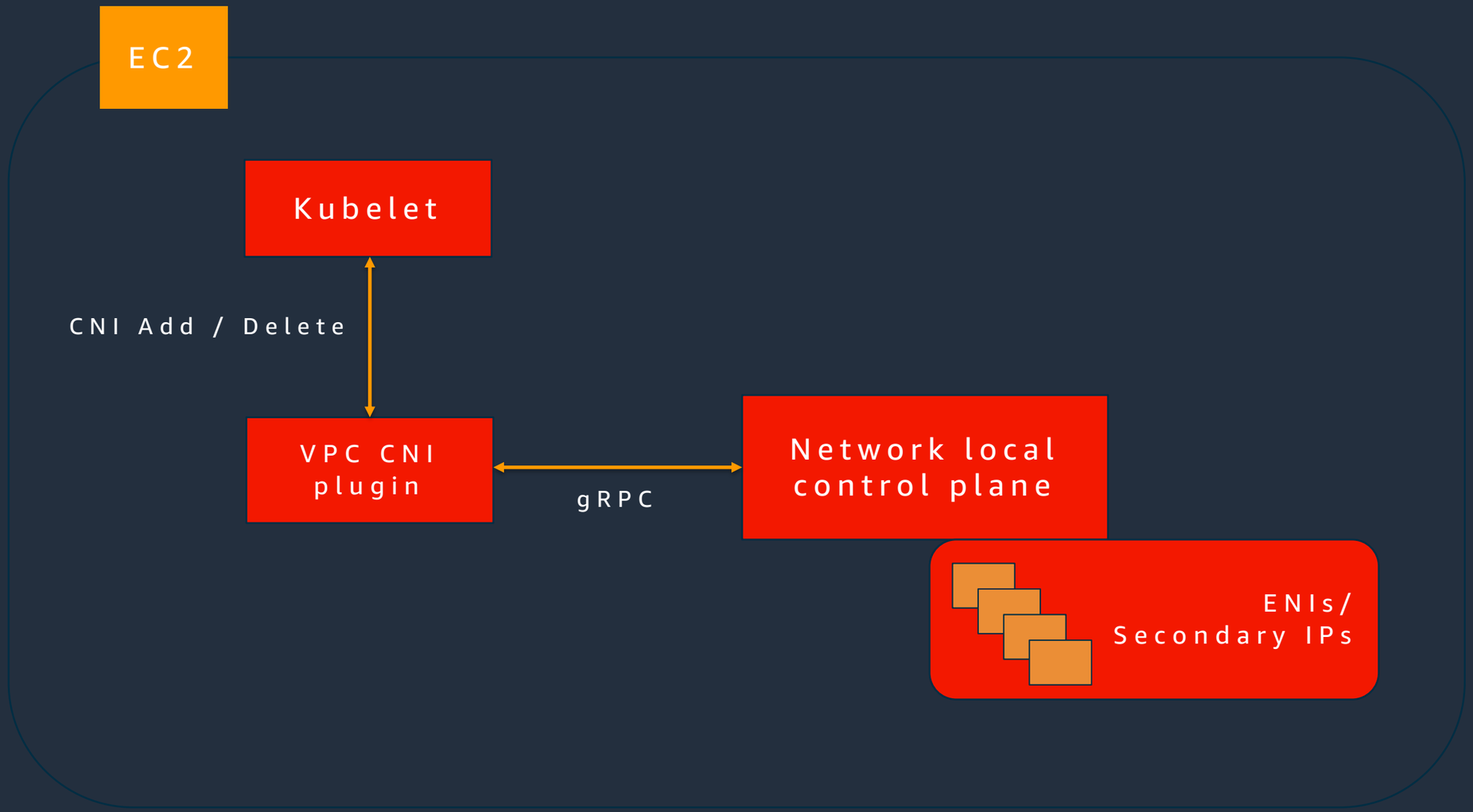
- Use separate VPC for each EKS cluster
- EKS requires subnets in at least two AZ’s
- VPC and Subnet CIDR
- Pod density calculation
  - Max Pods = min((N * M - N), subnet's free IP)
  - N = # of ENIs, M = # of IP per ENI

aws

Amazon VPC CNI Deep Dive

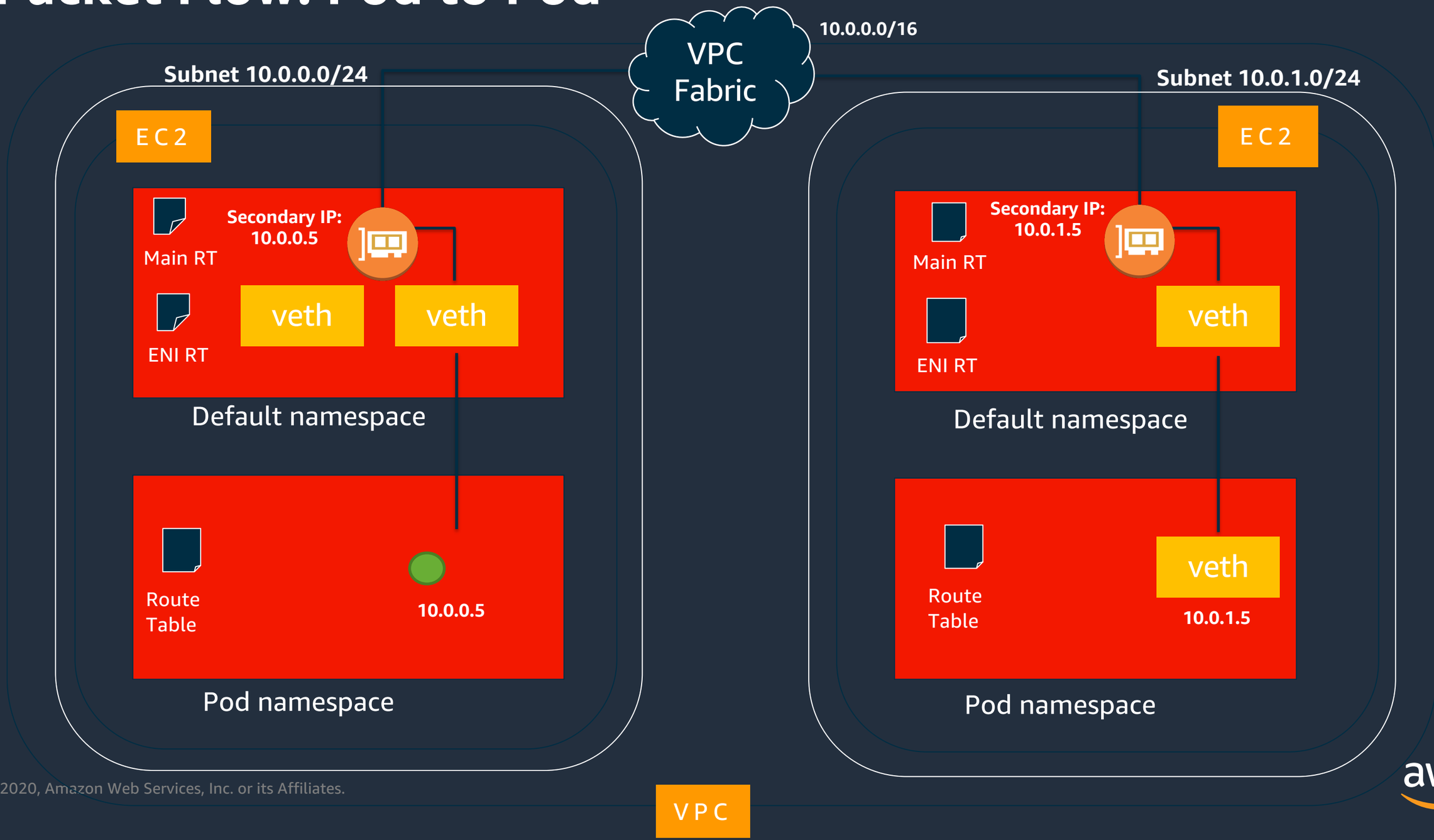
aws

Amazon VPC CNI Plugin Architecture

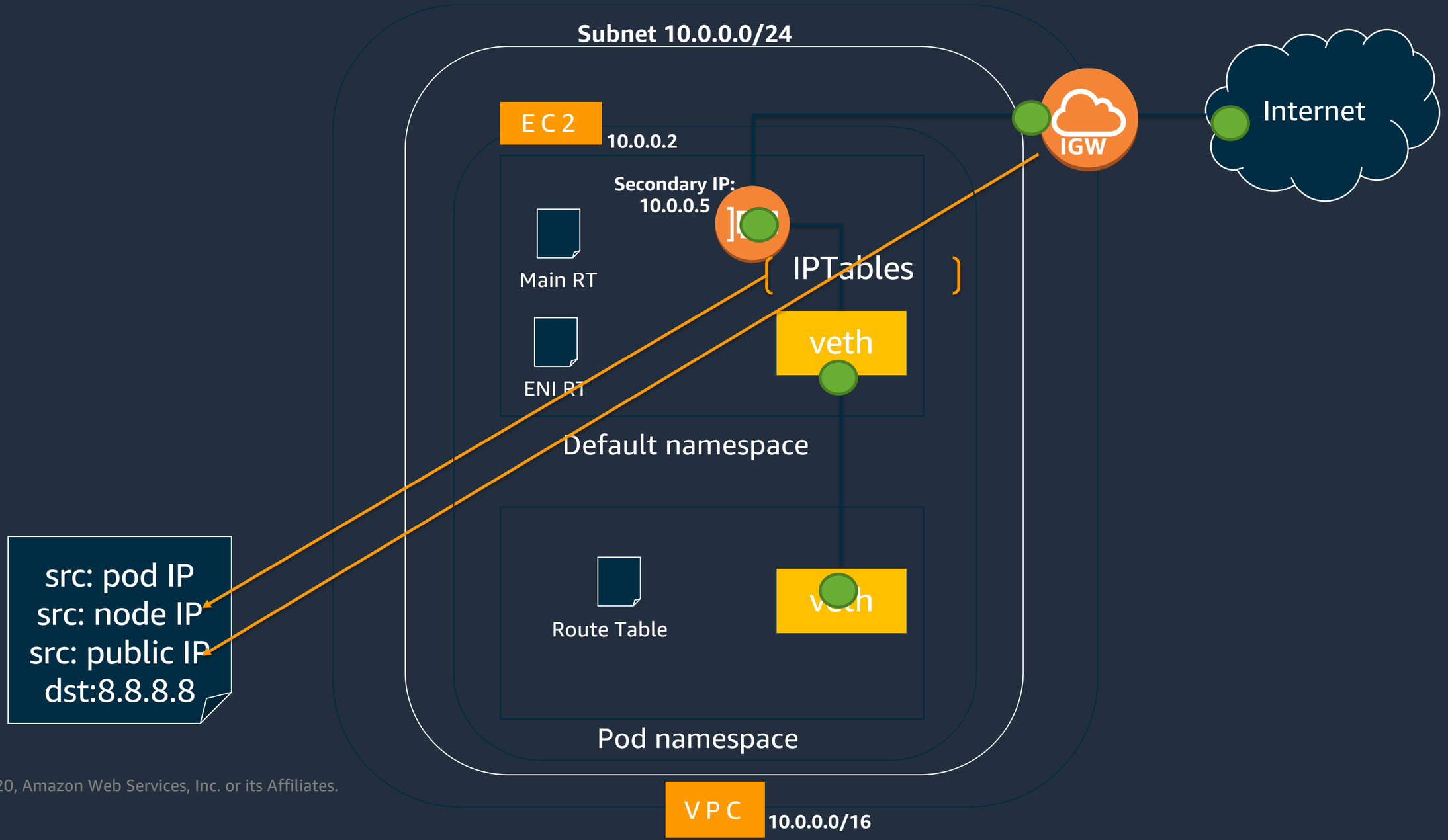


aws

Packet Flow: Pod to Pod



Packet Flow: Pod to Internet



aws

Amazon VPC CNI plugin – Understanding IP Allocation

Primary CIDR range

- RFC 1918 addresses ➔ 10/8, 172.16/12, 192.168/16
- Publicly routable CIDR block (since May 2019)

Used in EKS for:

- Pods
- X-account ENIs for (masters → workers) communication (exec, logs, proxy etc.)
- Internal Kubernetes services network (10.100/16 or 172.20/16)

Secondary CIDR

- non-RFC 1918 address blocks (100.64.0.0/10 and 198.19.0.0/16)

Used in EKS for Pods only

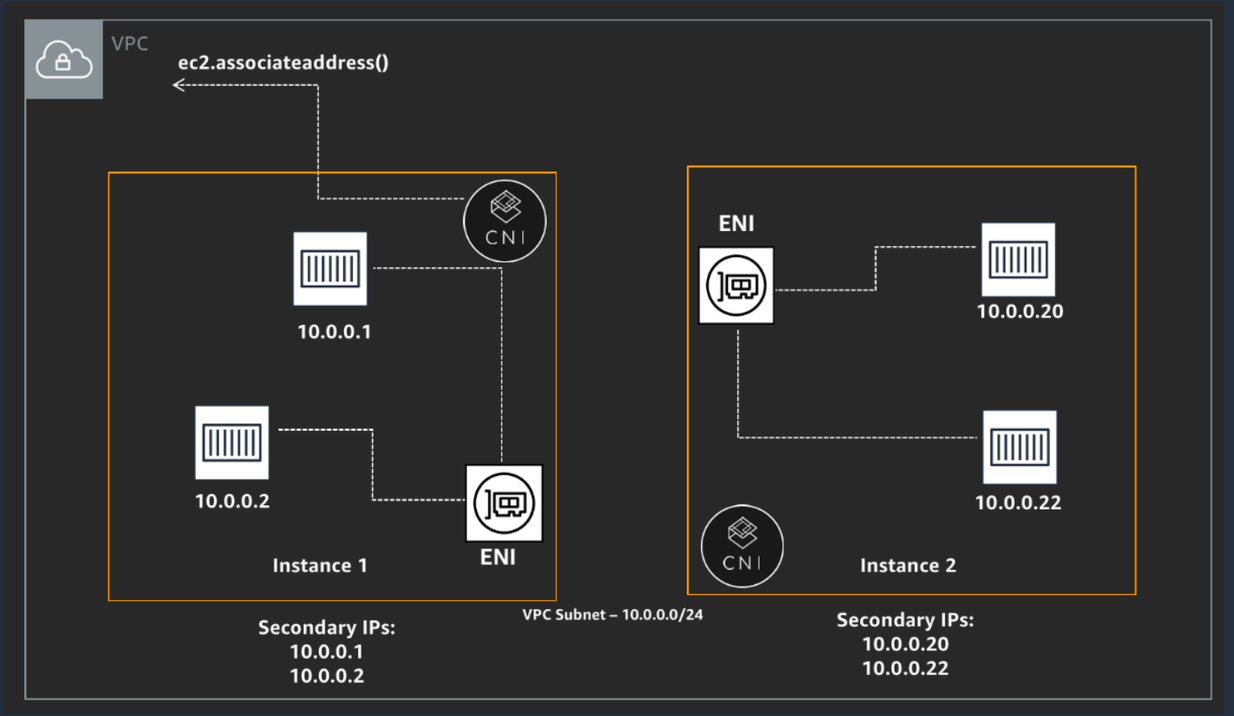
How?

ENIConfig

aws

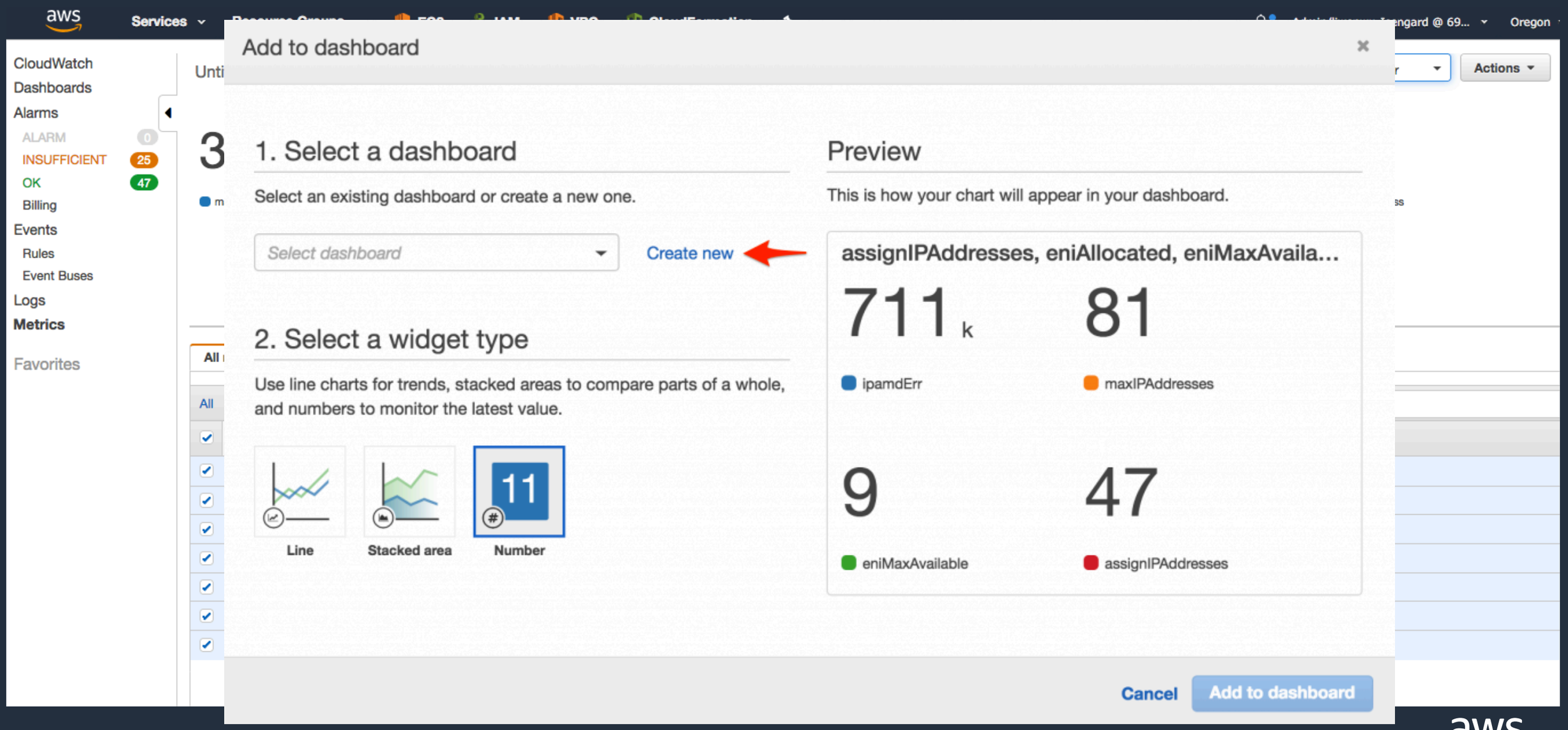
Amazon VPC CNI plugin - Configurability

- Custom Network Configs
- SNAT / External SNAT
- Configurable warm pool



aws

Introspecting the VPC CNI Plugin - CNI Metrics Helper



aws

Calico Network Policy

Kubernetes Network Policies enforce network security rules

Calico is the leading implementation of the network policy API

Open source, active development (>100 contributors)

Commercial support available from Tigera

Q&A

Thank you!