Overview of EKS Security

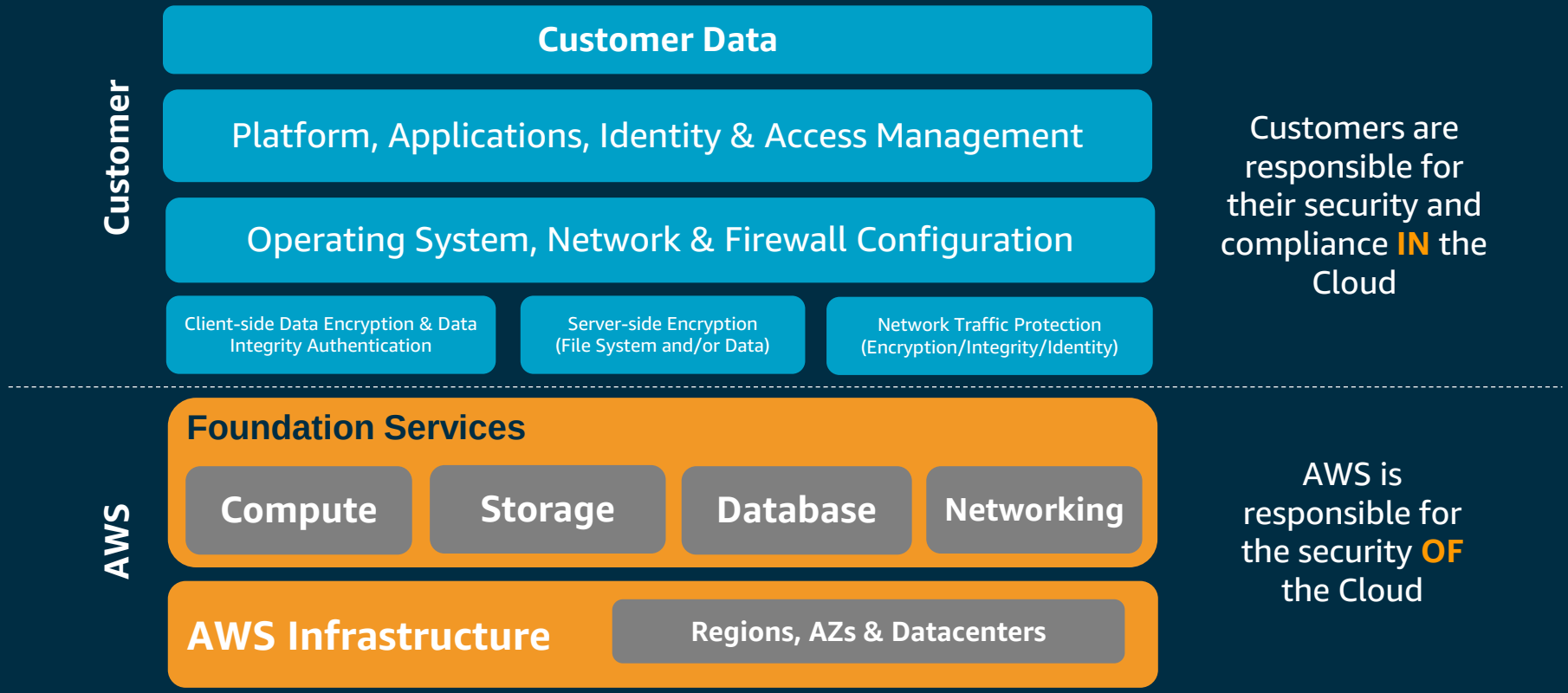
Manel Mendoza Flores
AWS Solution Architect.

Agenda

- AWS Shared Responsibility Model
- Container Security Best Practices
- Amazon EKS Security Best Practices

EKS Shared Responsibility Model

Responsibility for security and compliance is shared

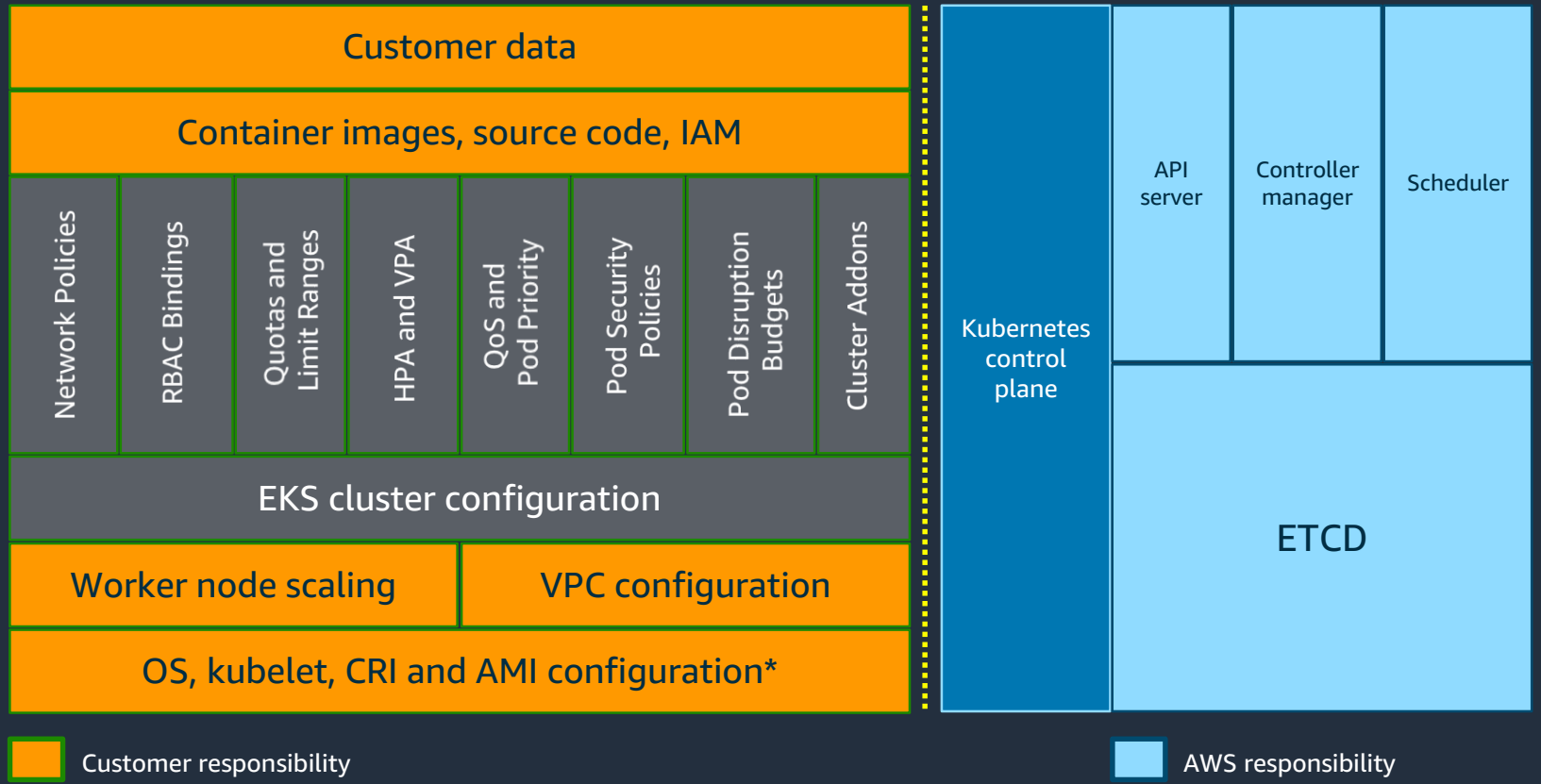


Customers are responsible for their security and compliance **IN** the Cloud

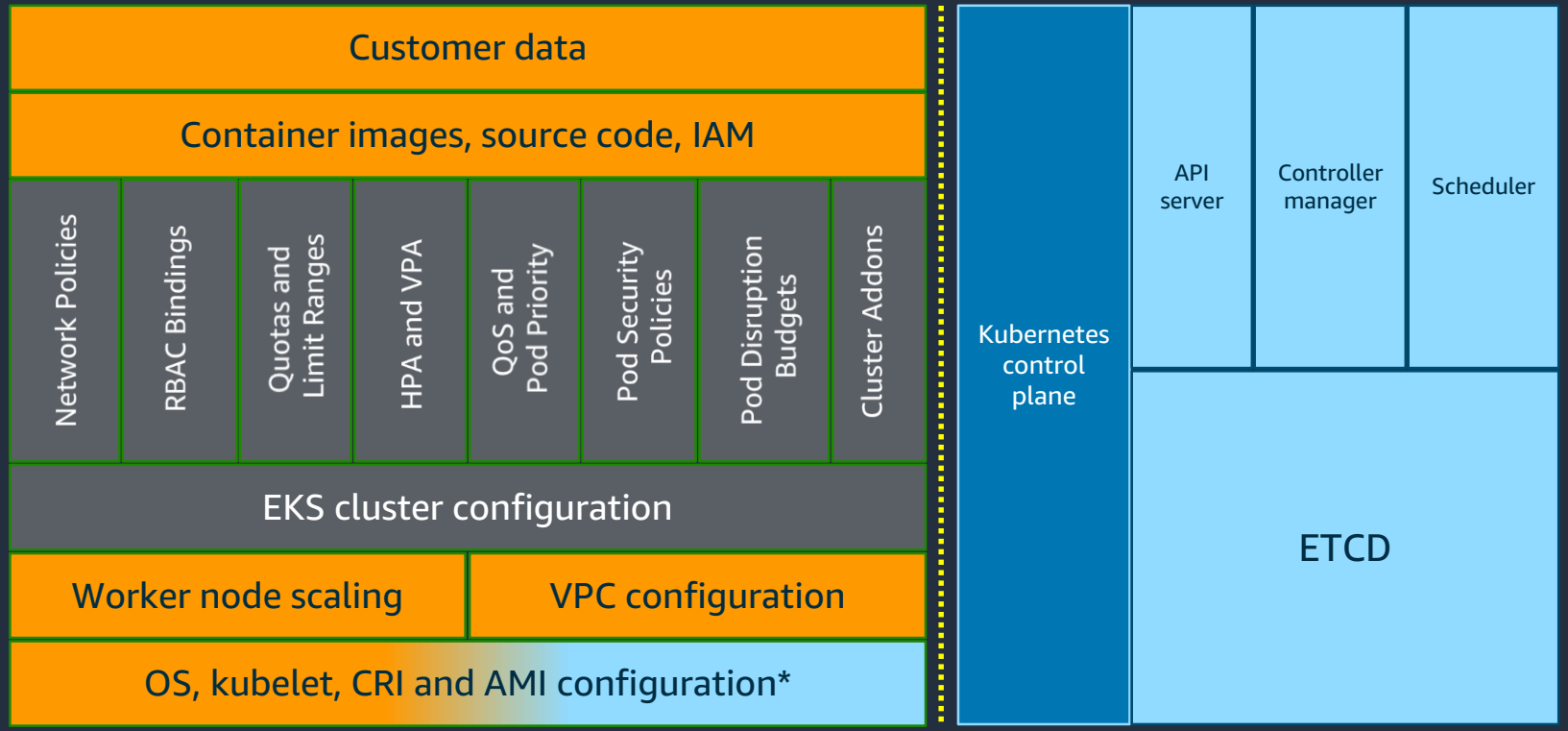
AWS is responsible for the security **OF** the Cloud

aws

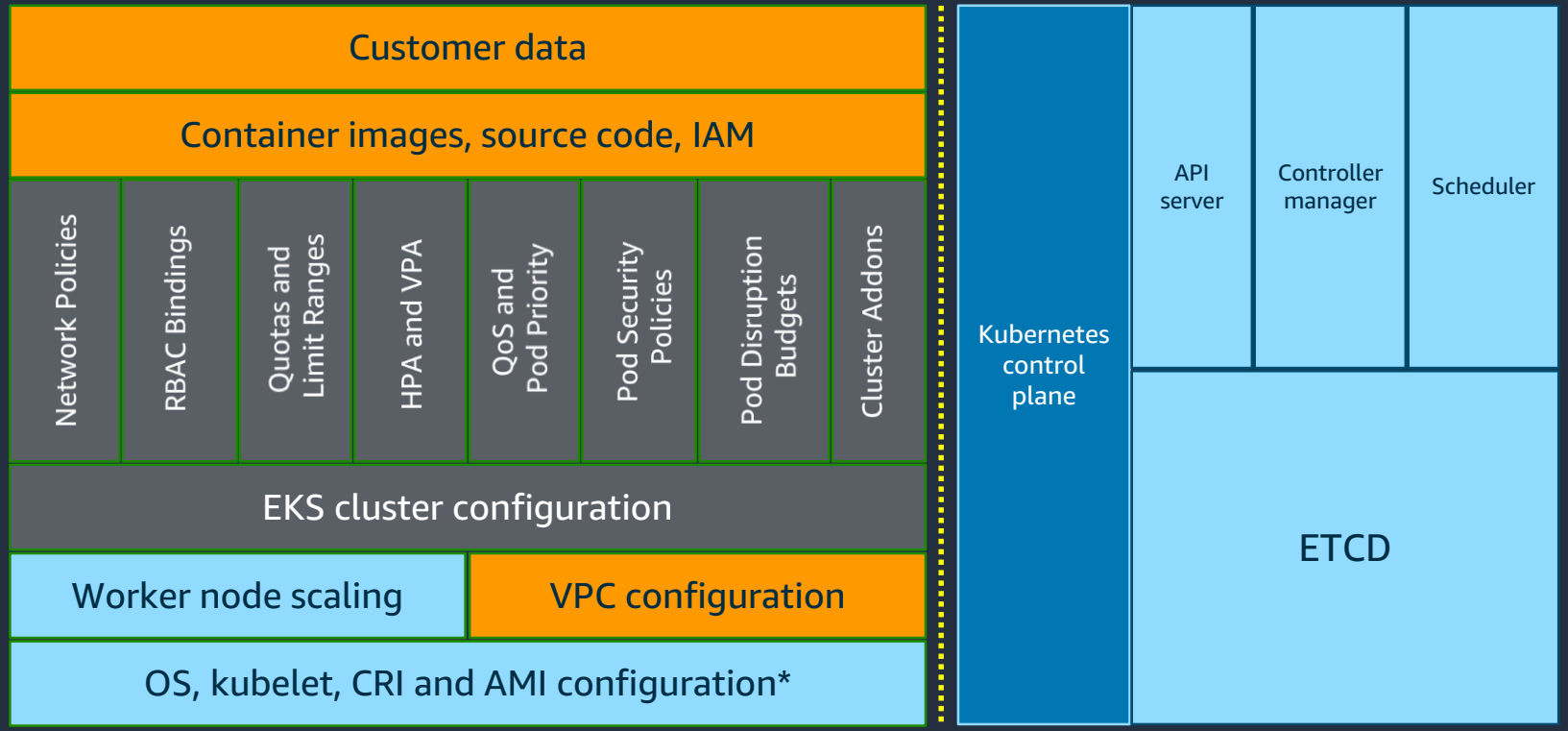
EKS with Self-Managed Workers



EKS with Managed Node Groups



EKS with Fargate

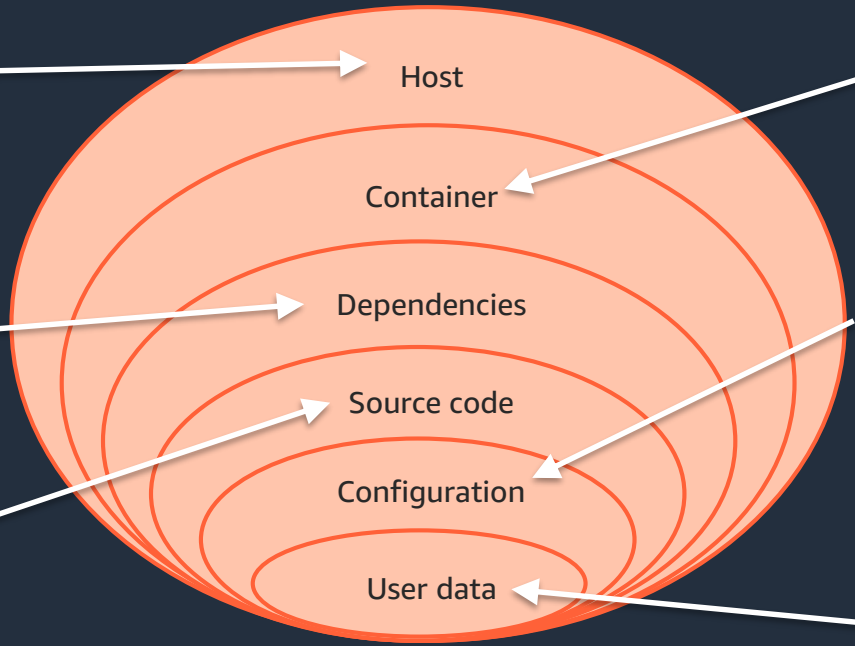


Amazon EKS Security Best Practices

Container Security

Defense in Depth

- Full-blown distribution Amazon Linux 2 (AL2) vs. 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**: huge surface, many languages

- Sensitive configuration (passwords, API keys, etc.)
- **Gotchas**: commits-to-source, non-separated access (dev. has cleartext password)

- Sanitizing user input
- Static code analysis
- **Gotcha**: log-leaking

- Business core data
- Personally identifiable information (PII)
- **Gotchas**: leaks, General Data Protection Regulation (GDPR (in Europe))

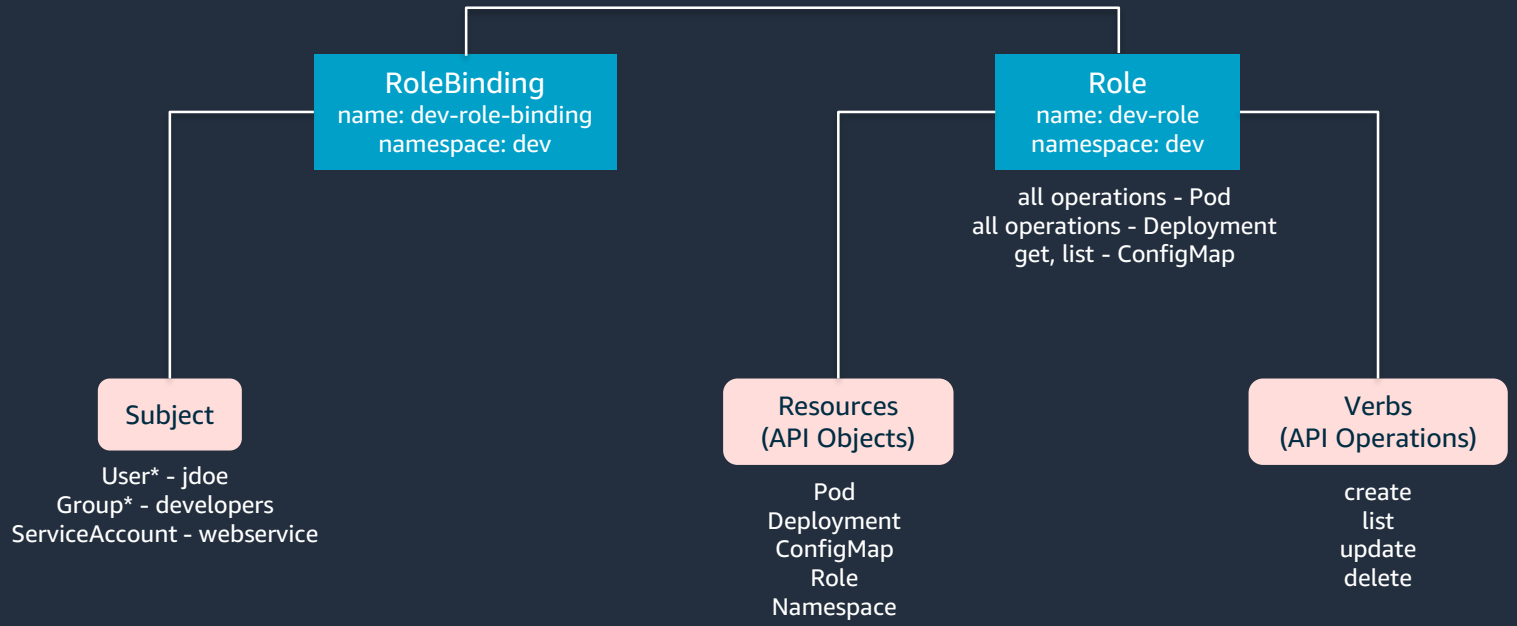
Amazon Elastic Container Registry (ECR)

- Fully-managed, OCI Compliant, container registry
  - Tag immutability
  - Image scanning (uses open-source Clair)
  - Lifecycle management of images using policies
  - Accessible over VPC interface endpoints
  - Access control using IAM policies
  - Cross-region replication

IAM & Kubernetes RBAC Integration

Kubernetss RBAC - Role & RoleBinding

- Access granted by Role and RoleBinding is limited to a namespace



Managing Authentication via IAM

User authenticates to AWS

User assumes an IAM role based on access

AWS IAM Role is mapped to Kubernetes RBAC users/roles

Managing Authentication via IAM

User authenticates to AWS

User assumes an IAM role based on access

AWS IAM Role is mapped to Kubernetes RBAC users/roles

Integration of Kubernetes RBAC & AWS IAM

- Securely authenticate users via well-known IAM methods
- Map IAM users and roles to Kubernetes subjects using aws-auth ConfigMap
- Manage access control within EKS cluster using Kubernetes RBAC

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: aws-auth
  namespace: kube-system
data:
  mapRoles: |
    - rolearn: arn:aws:iam::123456789012:role/EKS-WorkerNode-InstanceRole
      username: system:node:{{EC2PrivateDNSName}}
      groups: ['system:bootstrappers', 'system:nodes']
    - rolearn: arn:aws:iam::123456789012:role/EKS-Developer-Role
      groups: [developers]
```

IAM Role

Kubernetes Group

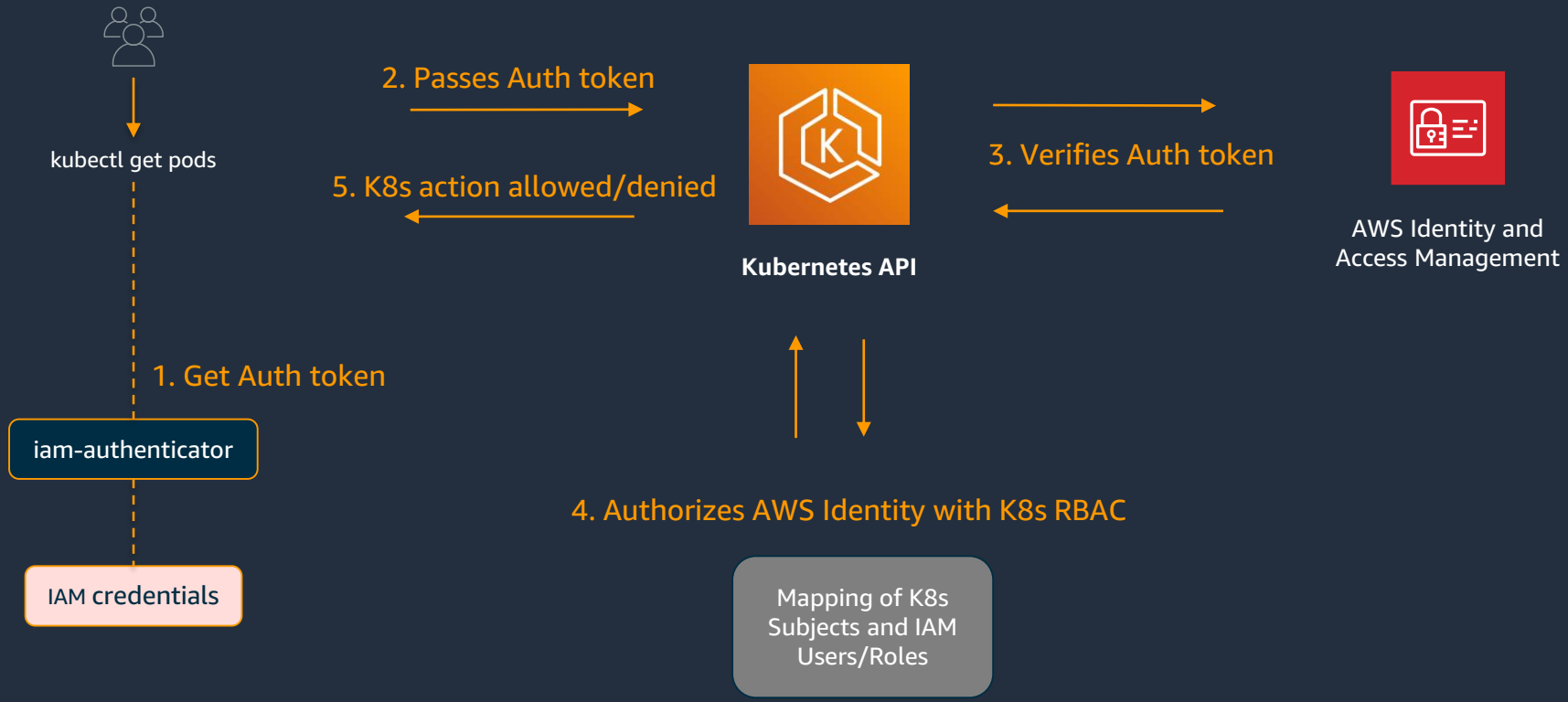
Managing Authorization via Kubernetes RBAC

```yaml
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: developers-role-binding
  namespace: dev
roleRef:
  kind: Role
  name: developers-role
  apiGroup: rbac.authorization.k8s.io
subjects:
  - kind: Group
    name: developers
    apiGroup: rbac.authorization.k8s.io
```

```yaml
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: Role
metadata:
  name: developers-role
  namespace: dev
rules:
  - apiGroups:
      - ""
      - apps
      - batch
      - extensions
      - rbac.authorization.k8s.io
    resources: ["*"]
    verbs:
      - get
      - list
      - watch
      - create
      - update
      - patch
      - delete
```

- Kubernetes Users/Groups are mapped to Kubernetes Roles using RoleBindings

IAM Authentication to EKS



Best Practices with IAM Roles for EKS

- Maintain clear separation between IAM roles used for different tasks in an EKS cluster
  - Role for EKS cluster creation
  - EKS cluster role (passed to EKS service at the time of cluster creation)
  - Roles for Kubernetes service accounts
  - Roles for authenticating clients access to EKS cluster
- Employ principle of least privileges

Logging

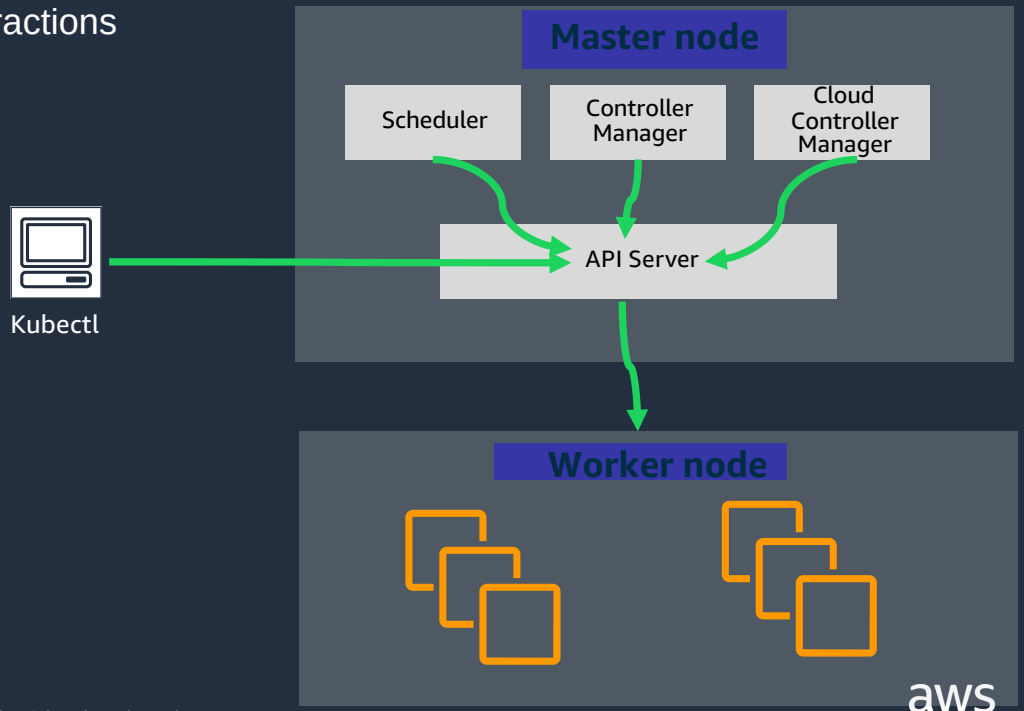
Kubernetes control plane API and audit logs

Kubernetes control plane API – HTTP API to **query** and **manipulate** the state of API objects in Kubernetes

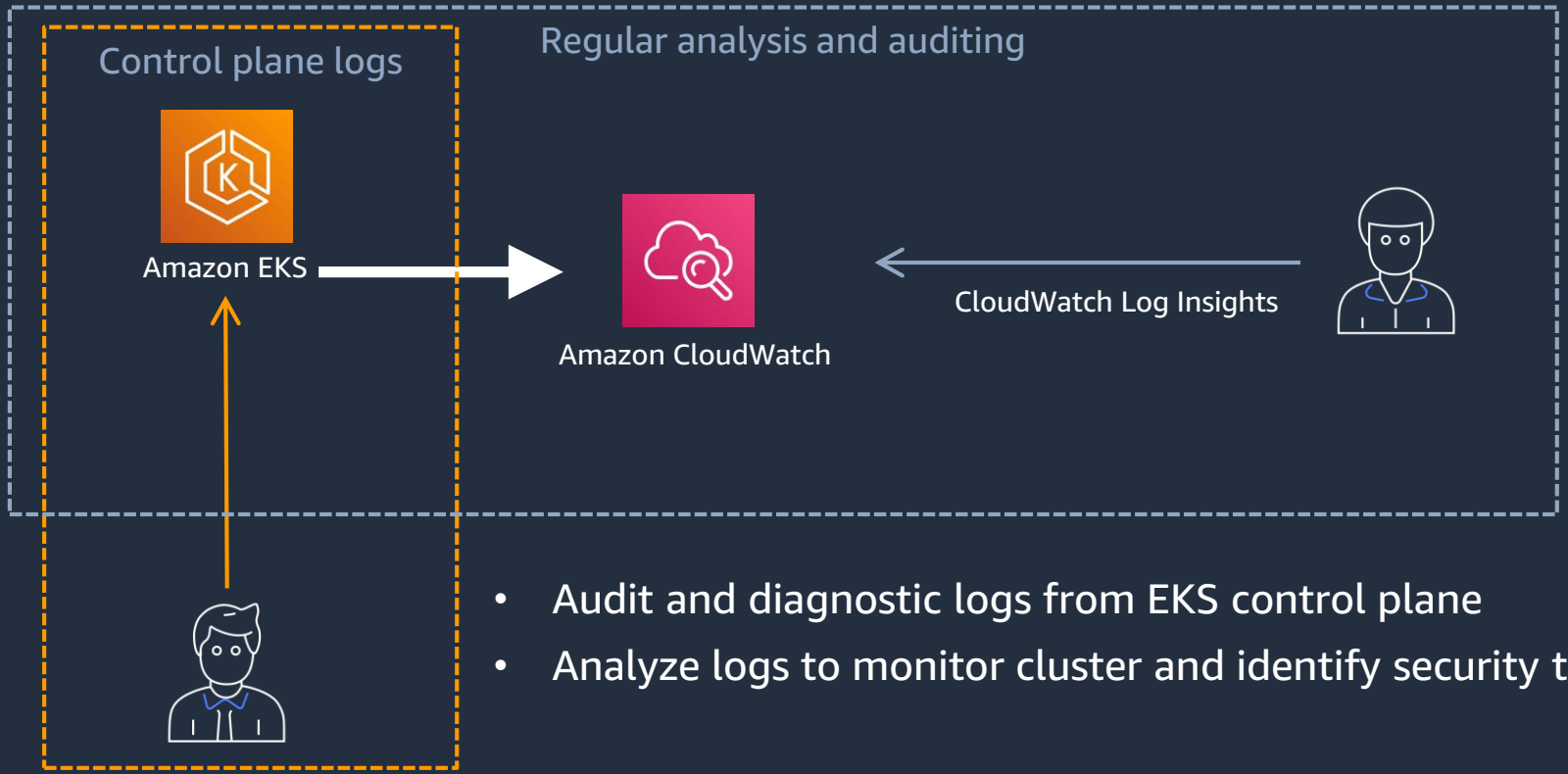
- Pods, namespaces, ConfigMaps, events

Audit logs provide information on API interactions

- What happened?
- When did it happen?
- Who initiated it?
- On what did it happen?
  - Endpoints, pods, ConfigMap, and so on
- Where was it observed?
- From where was it initiated?
- To where was it going?

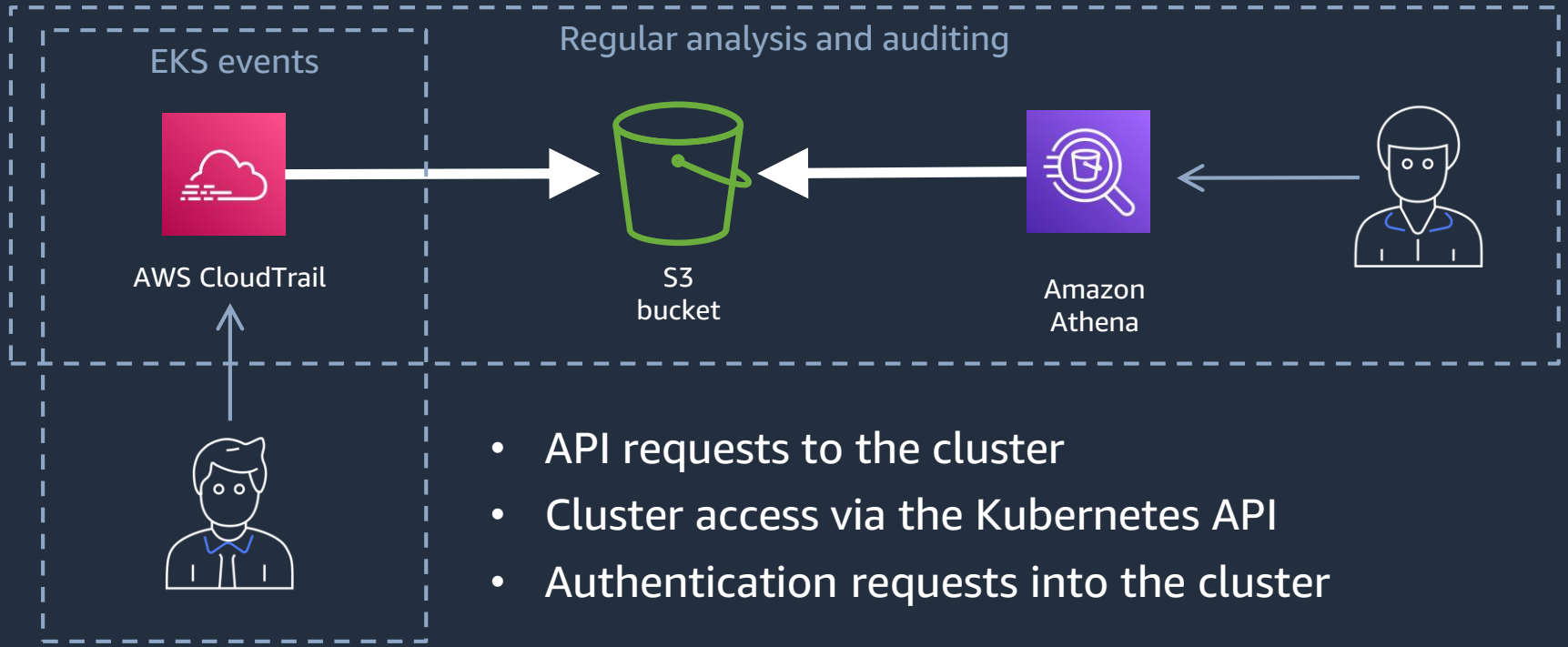


Analyze Control Plane Logs



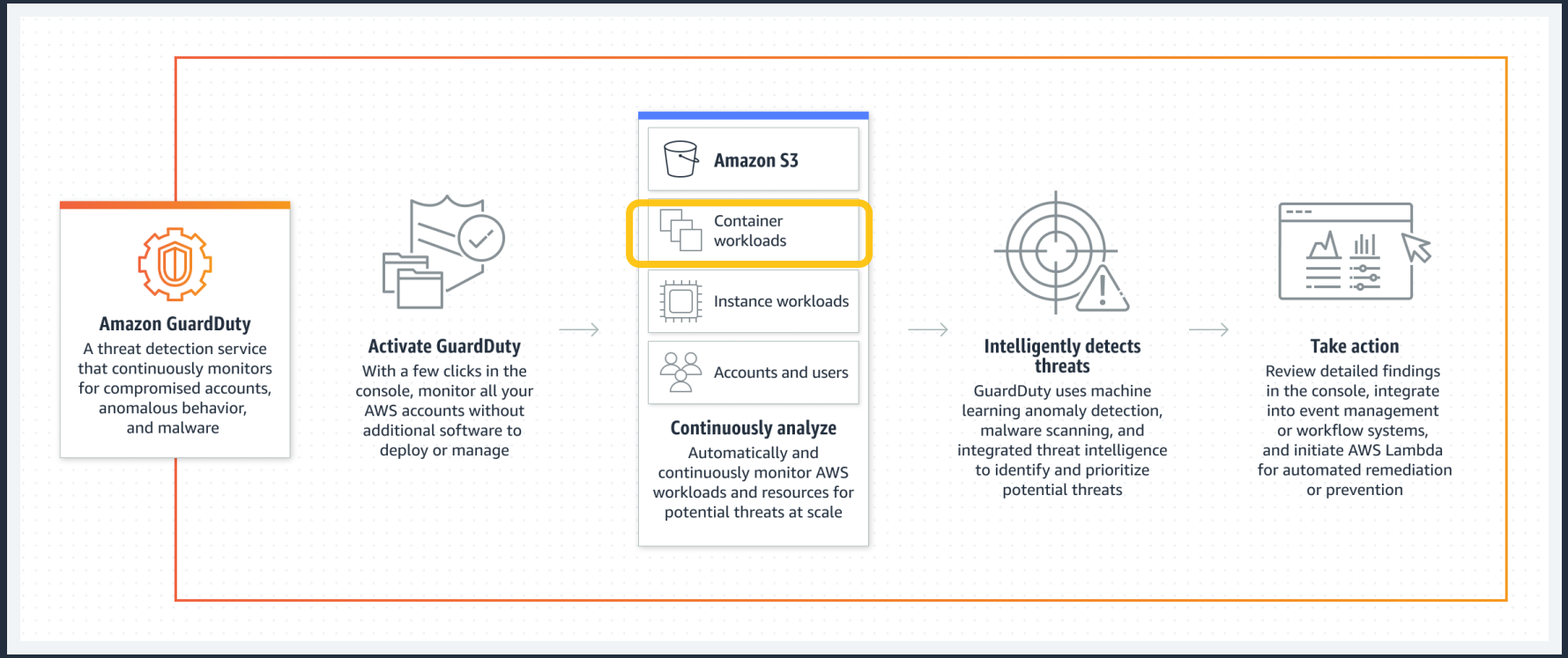
- Audit and diagnostic logs from EKS control plane
- Analyze logs to monitor cluster and identify security threats

Audit EKS API calls with CloudTrail



- API requests to the cluster
- Cluster access via the Kubernetes API
- Authentication requests into the cluster

Amazon GuardDuty



GuardDuty Kubernetes finding types

Policy

- Exposed dashboard
- Admin access to default service account
- Anonymous access granted

Malicious access

- Data discovery, exfiltration, or modification from:
  - Tor
  - Successful anonymous access
  - Malicious IPs

Suspicious behavior

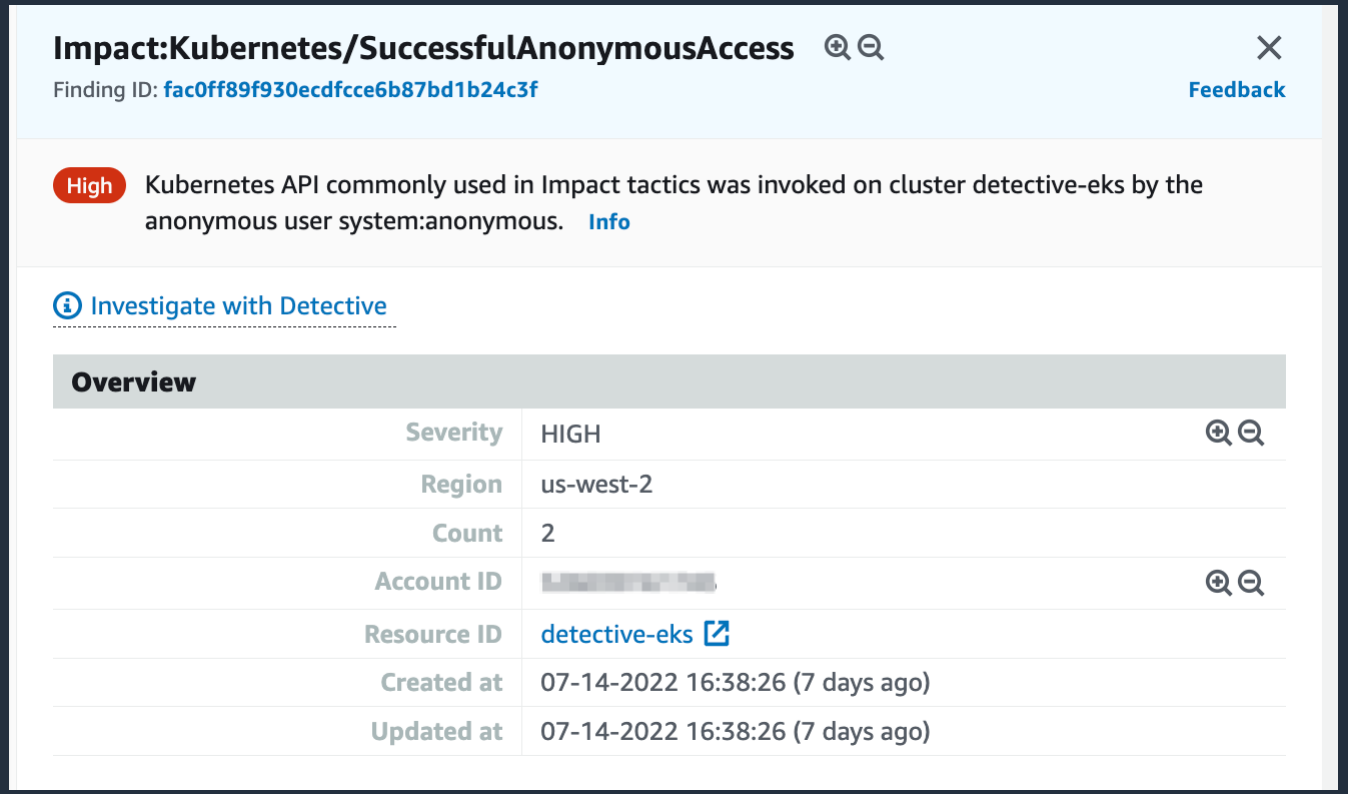
- Execution in Kubernetes system pod
- Container with sensitive mount
- Privilege container

GuardDuty immediately begins to analyze Kubernetes data sources from your Amazon EKS clusters and monitors them for malicious and suspicious activity

GuardDuty aligns findings using the MITRE ATT&CK framework

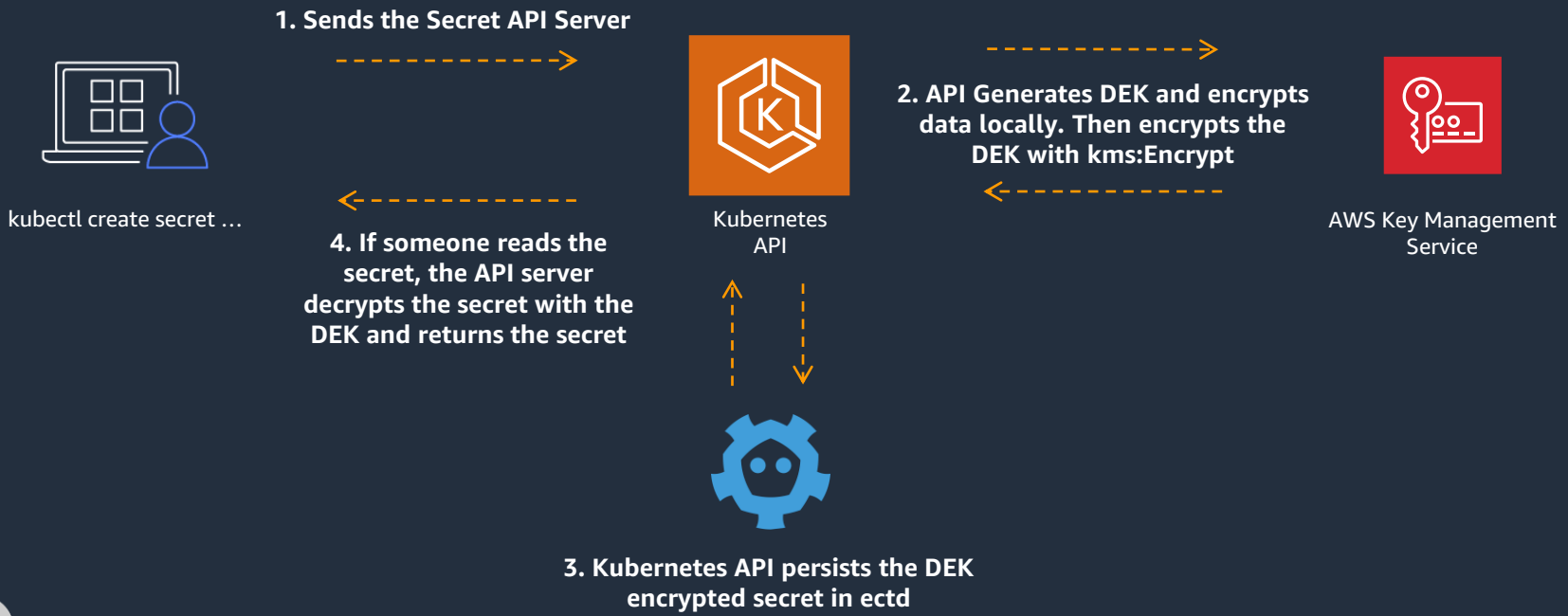
Credential access Defense evasion Discovery Impact
Privilege escalation Policy Execution Persistence

GuardDuty Kubernetes findings detail



Managing Secrets

Envelope Encryption for Kubernetes Secrets



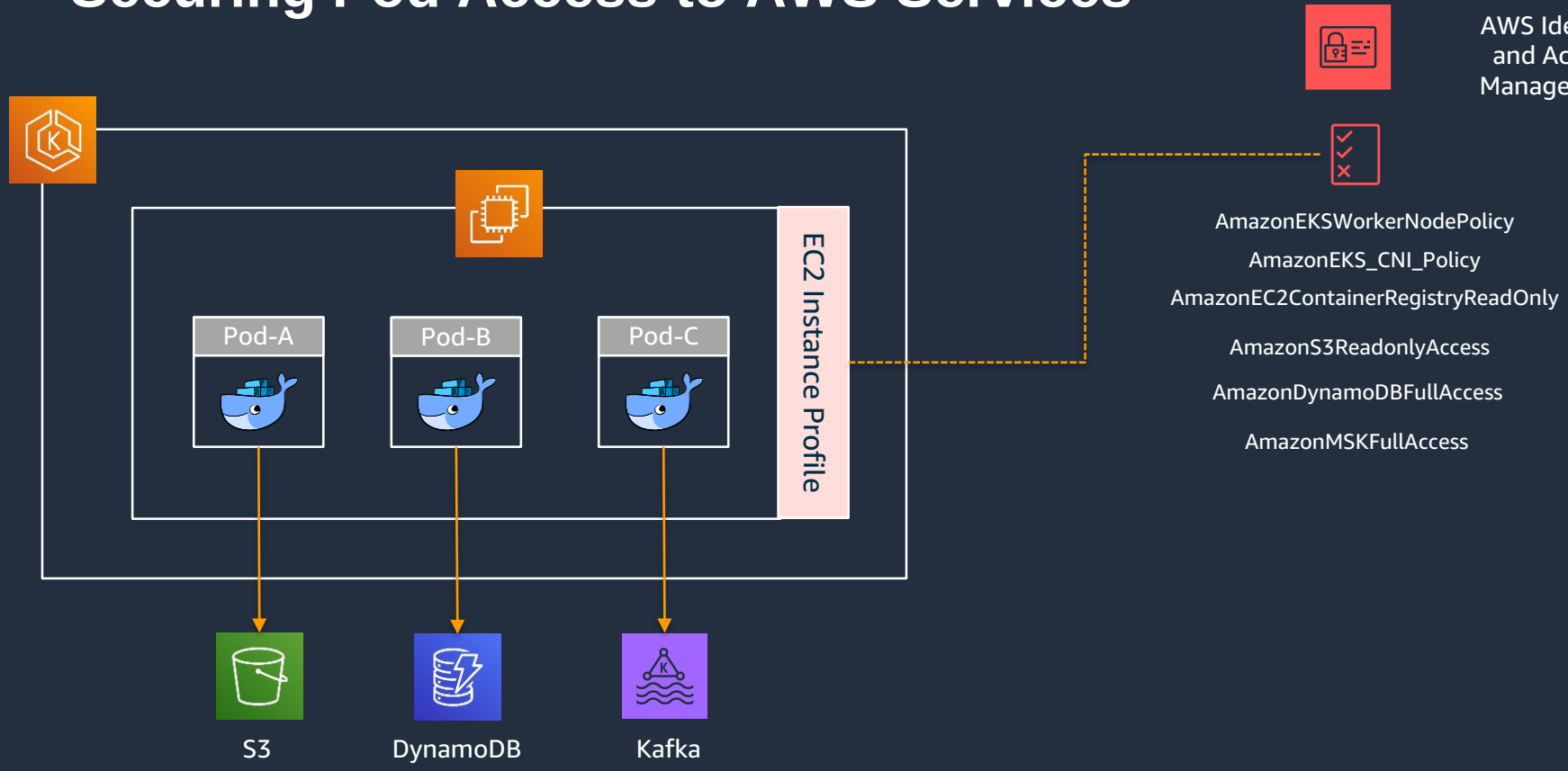
kubernetes-sigs/aws-encryption-provider

AWS Secrets Manager and Configuration Provider (ASCP)

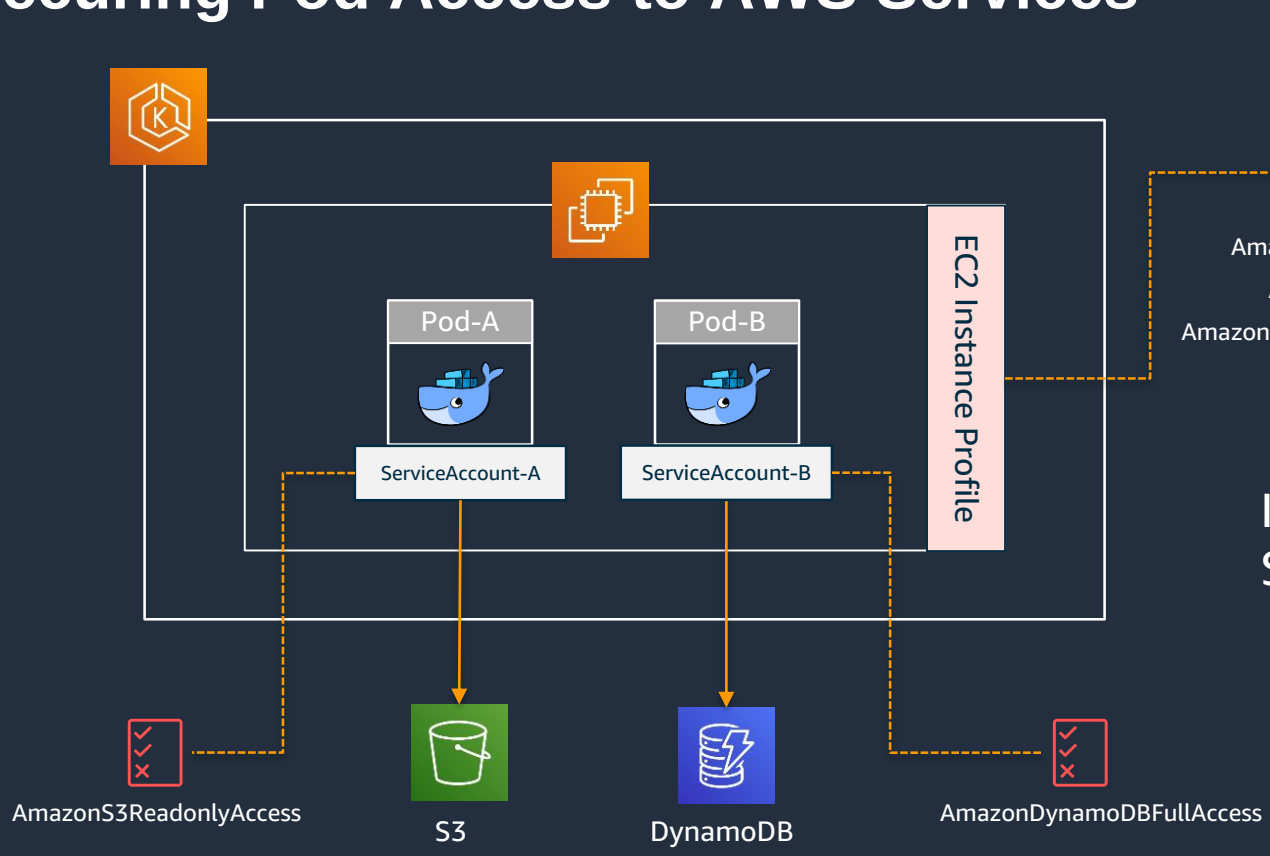
- ASCP is a plugin for industry standard Kubernetes Secrets Store CSI Driver
- Securely store and manage secrets in Secrets Manager or SSM Parameter Store
- Make secrets accessible to Pods running on EKS
  - Mounted into the Pod file system as volume
  - Exposed as Kubernetes Secret resource
- Limit and restrict secrets access to specific Pods with IAM policies using IRSA

Roles Assignments Service Accounts

Securing Pod Access to AWS Services



Securing Pod Access to AWS Services



IRSA (IAM roles for Service accounts)

Network Security

Securing Pod to Pod Communications

Security groups for pods

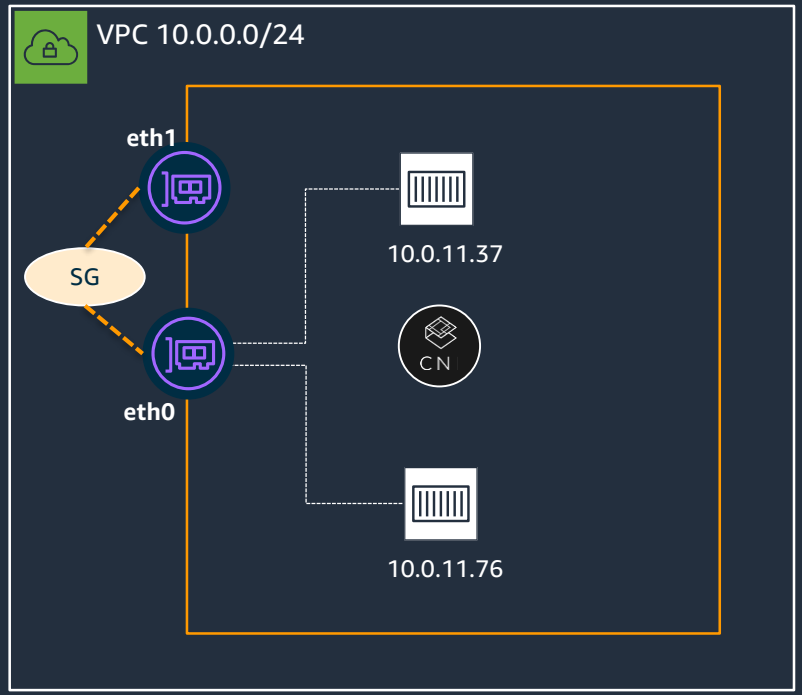
- Integrate EC2 security groups with Kubernetes pods
- Re-use operational knowledge, and tooling around existing security groups

Network policies

- Implement network segmentation and tenant isolation
- Kubernetes native approach for traffic control

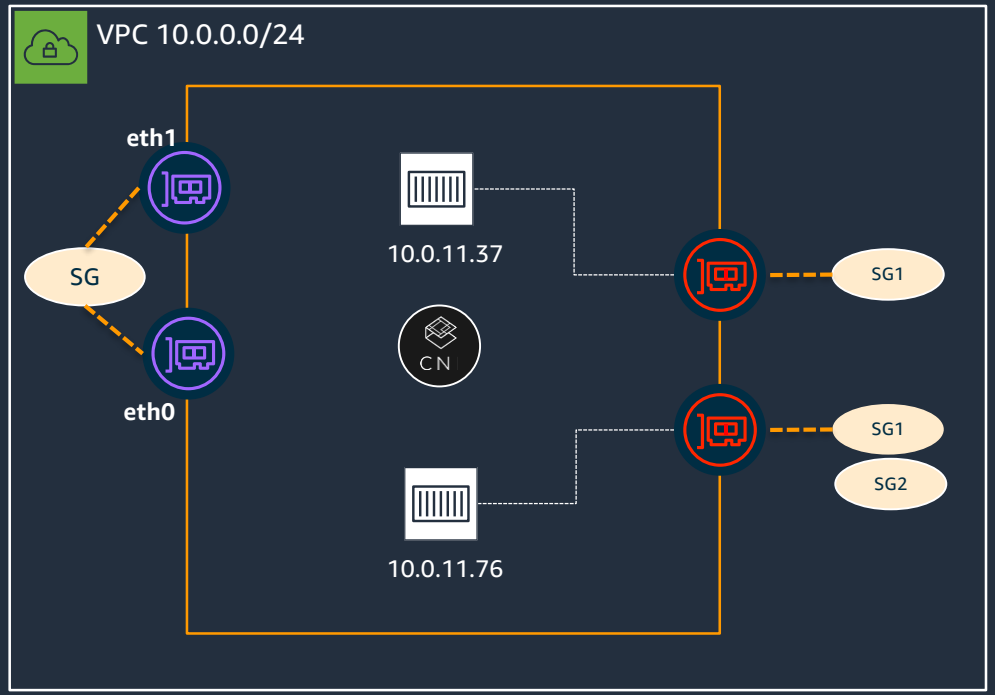
Securing Pods with EC2 Security Groups

- Worker node and all its Pods share the same security group(s) attached to the primary ENI



Securing Pods with EC2 Security Groups

- Worker node and all its Pods share the same security group(s) attached to the primary ENI
- Assign each Pod to a dedicated ENI with its own separate set of security groups(s)

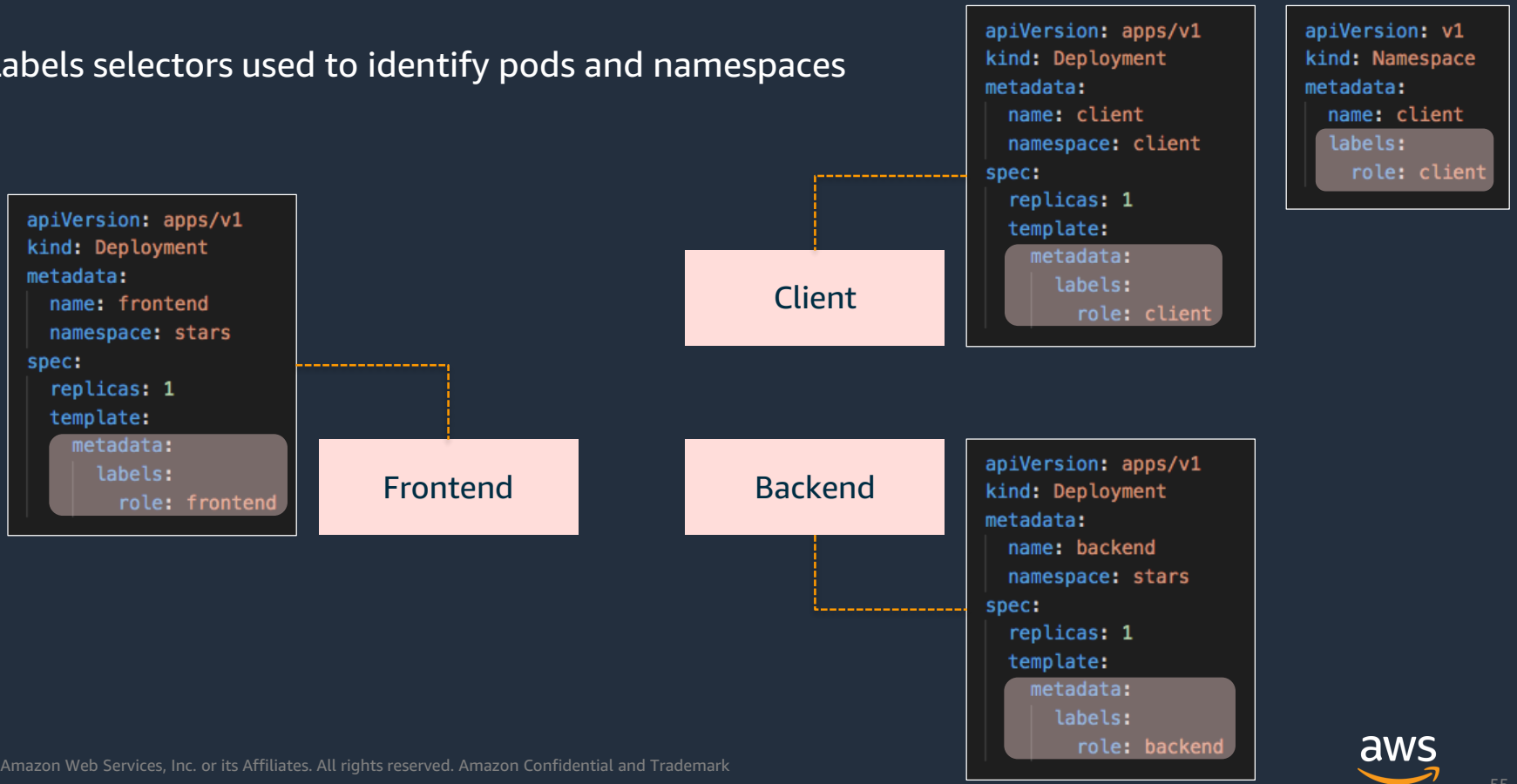


What is a Network Policy?

- NetworkPolicy specifies how pods communicate with each other and other network endpoints.
- Network policies define egress and ingress rules in order to secure network traffic
- Network policies are implemented by the CNI plugin
  - AWS VPC CNI plugin does not have a policy engine
  - EKS clusters need a policy engine such as Calico to support network policies
- Does not support controlling access to AWS resources outside the cluster

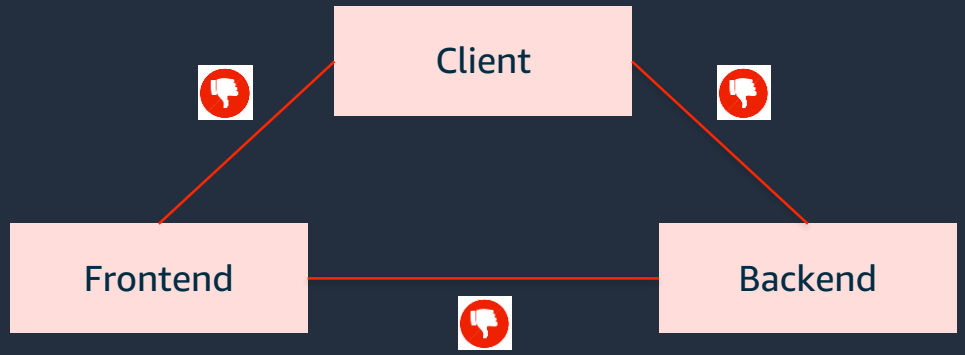
Network Policy Manifest

- Labels selectors used to identify pods and namespaces



Network Policy: Deny All Rule

- NetworkPolicy to isolate all services from each other

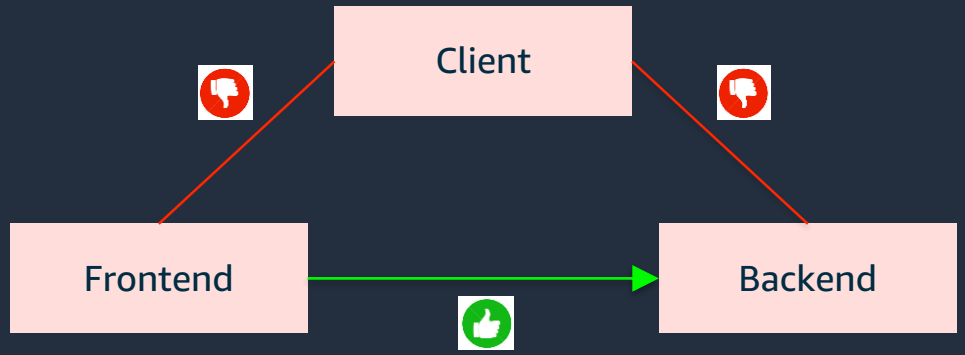



```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
  namespace: stars
spec:
  podSelector:
    matchLabels: {}
```

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

Network Policy: Ingress Rule

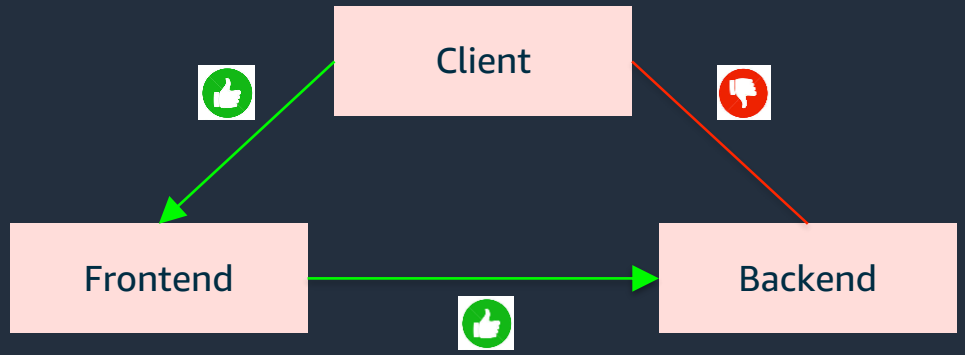
- NetworkPolicy to allow traffic to ingress into Backend pod from Frontend pod
- No effect on ingress traffic for Frontend and Client pods




```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  namespace: stars
  name: backend-policy
spec:
  podSelector:
    matchLabels:
      role: backend
  ingress:
    - from:
        - podSelector:
            matchLabels:
              role: frontend
      ports:
        - protocol: TCP
          port: 6379
```

Network Policy: Ingress Rule

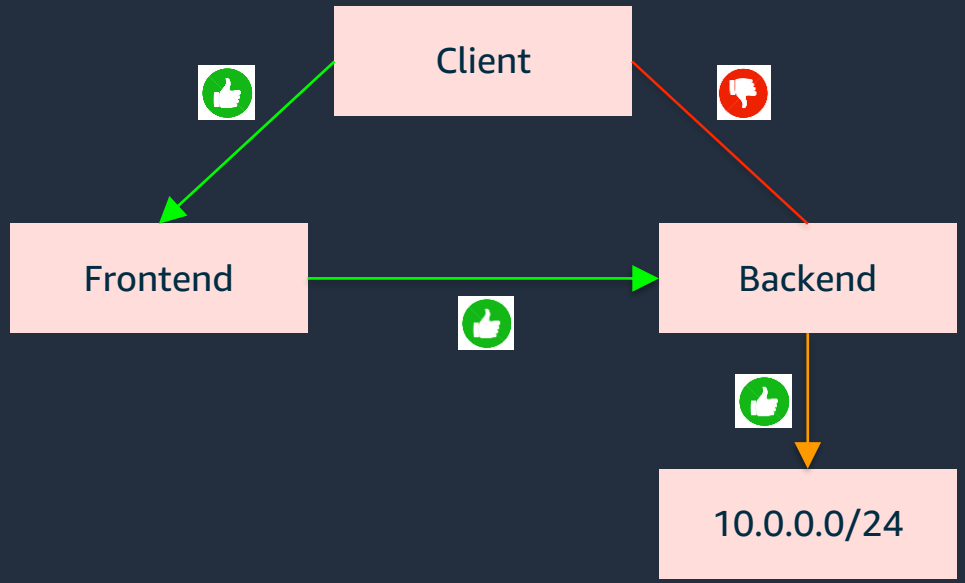
- NetworkPolicy to allow traffic to ingress into Frontend pod from client namespace




```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  namespace: stars
  name: frontend-policy
spec:
  podSelector:
    matchLabels:
      role: frontend
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              role: client
      ports:
        - protocol: TCP
          port: 80
```

Network Policy: Egress Rule

- NetworkPolicy to allow traffic to egress to specific CIDR ranges




```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  namespace: stars
  name: backend-policy
spec:
  podSelector:
    matchLabels:
      role: backend
  ingress:
    - from:
        - podSelector:
            matchLabels:
              role: frontend
      ports:
        - protocol: TCP
          port: 6379
  egress:
    - to:
        - ipBlock:
            cidr: 10.0.0.0/24
```

Summary

- Apply security in depth.
- K8s users management integrated with AWS IAM, to improve identity lifecycle.
- Uses IAM roles for service accounts applied to pods.
- Enable logging for all clusters.
- Enable Guarddutty for EKS and ECR containers scanning.
- Use AWS secret manager to improve
- Apply network security policies to isolate the namespaces / workloads.

Labs

- RBAC
- IRSA (IAM Role For Service Account)
- Network policies
- Security Groups 4 pods

Thank You