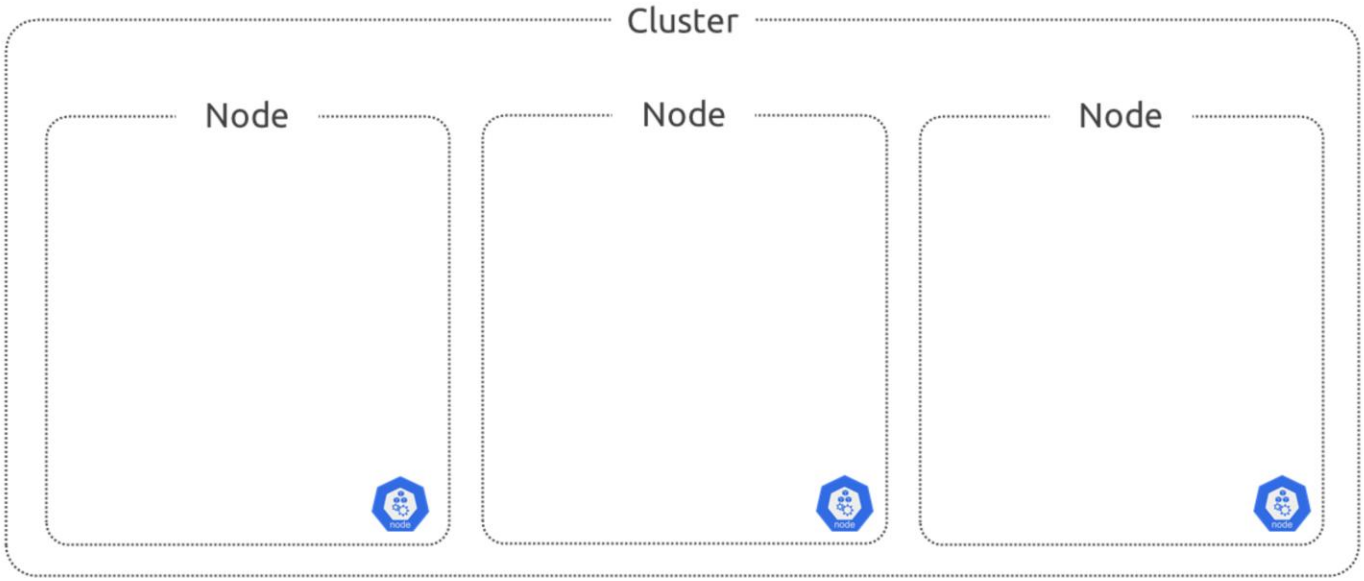

We'll now look at the security best practices to be followed on the controller manager and scheduler

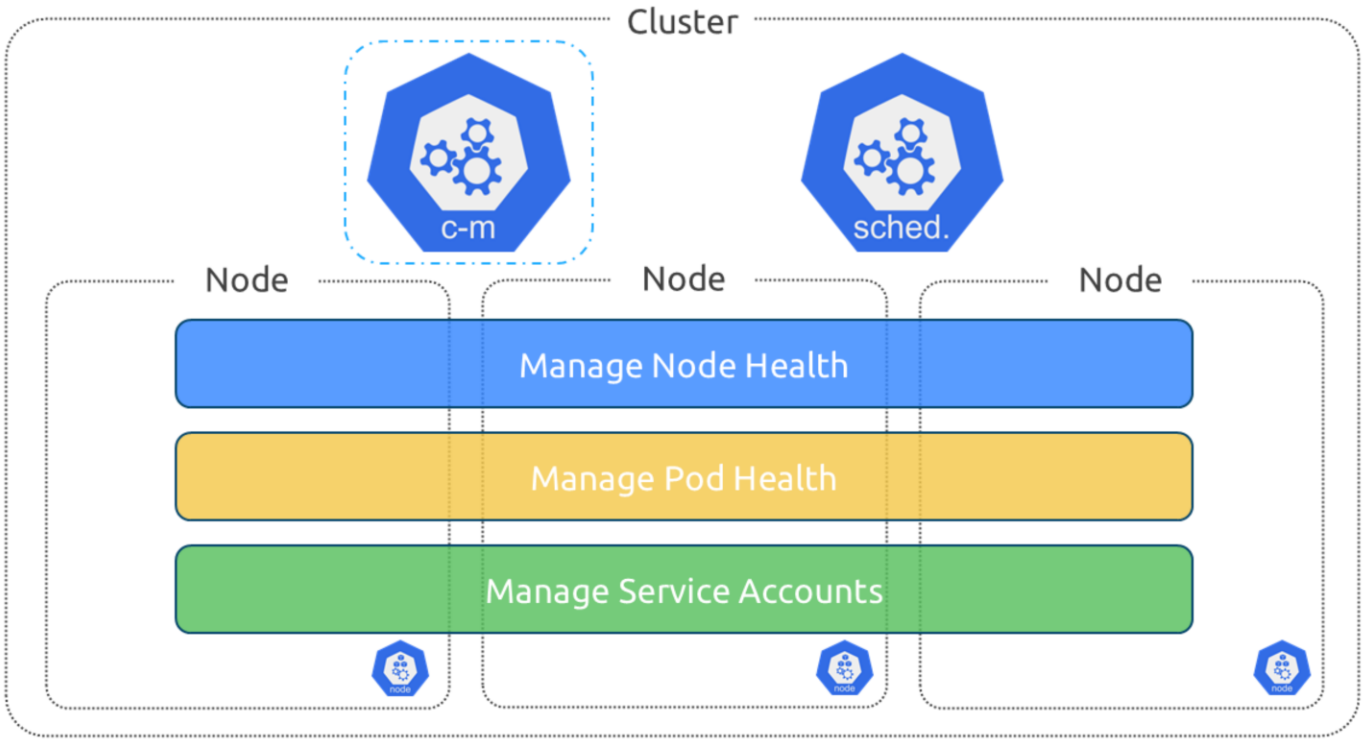
# Kubernetes Controller Manager and Scheduler






Here is a cluster with multiple nodes.

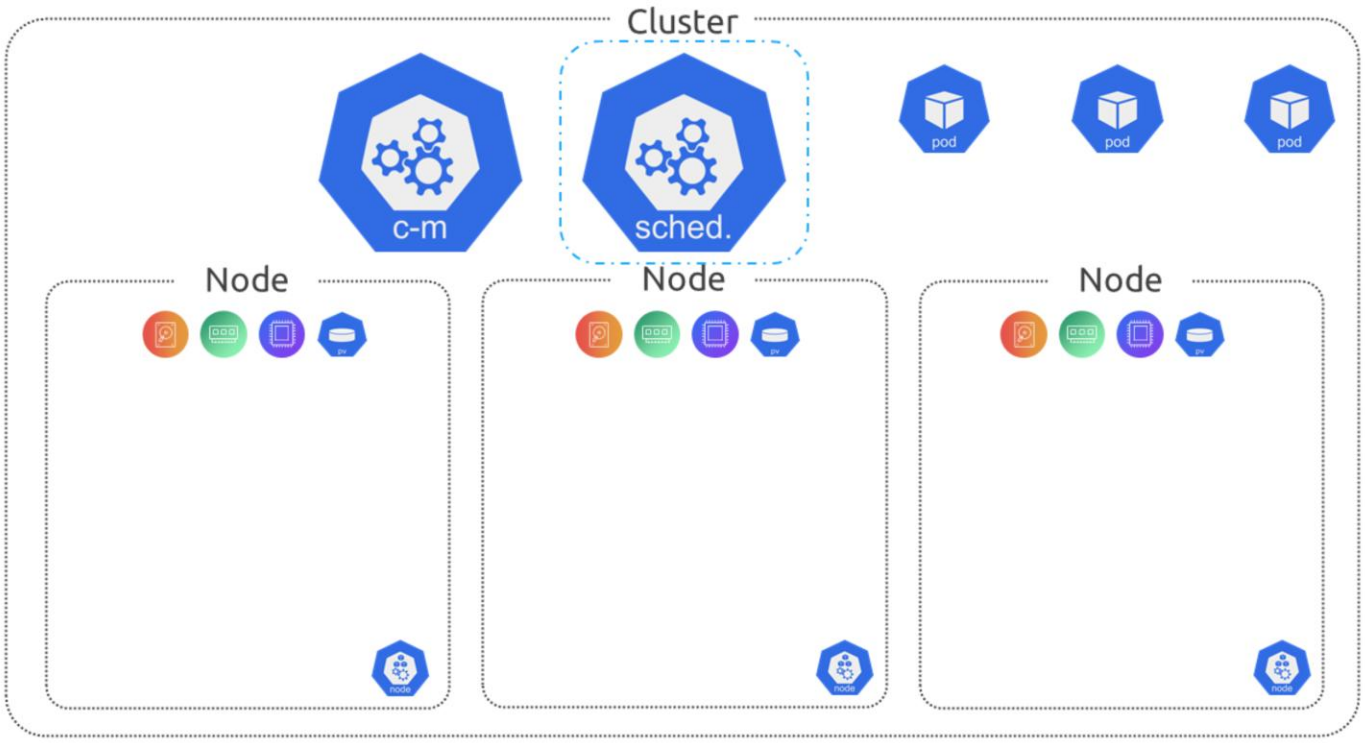
# Kubernetes Controller Manager and Scheduler






The Kubernetes Controller Manager and Scheduler are crucial for keeping your cluster in the desired state. The Controller Manager handles tasks like ensuring nodes are healthy, maintaining the right number of pod replicas, and managing service accounts, with controllers like the Replication Controller, Endpoints Controller, Namespace Controller, and ServiceAccounts Controller.

# Kubernetes Controller Manager and Scheduler

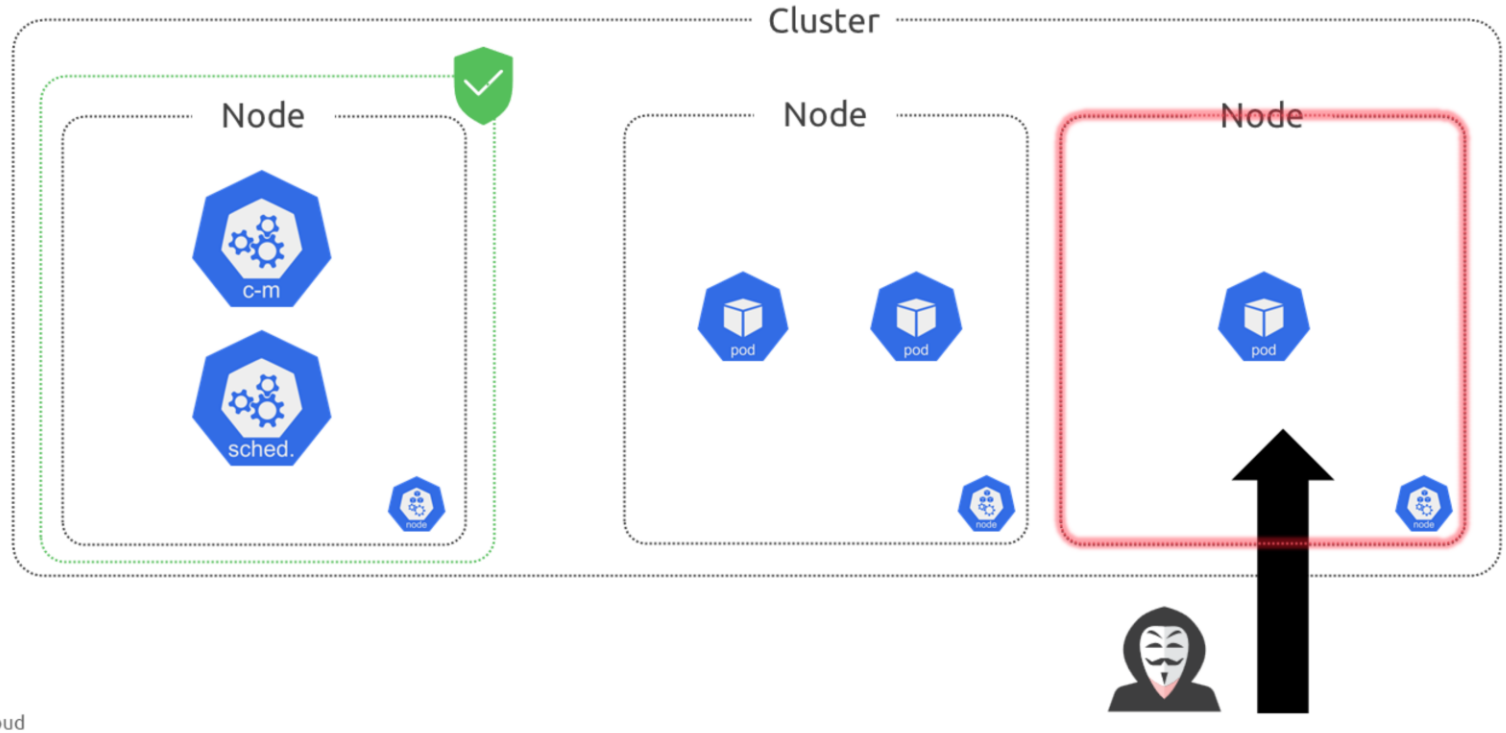





The Scheduler manages where and when pods run within your cluster, looking at available resources and deciding the best node for each pod.

Now how do we protect the Controller Manager and Scheduler? You protect them by isolating them.

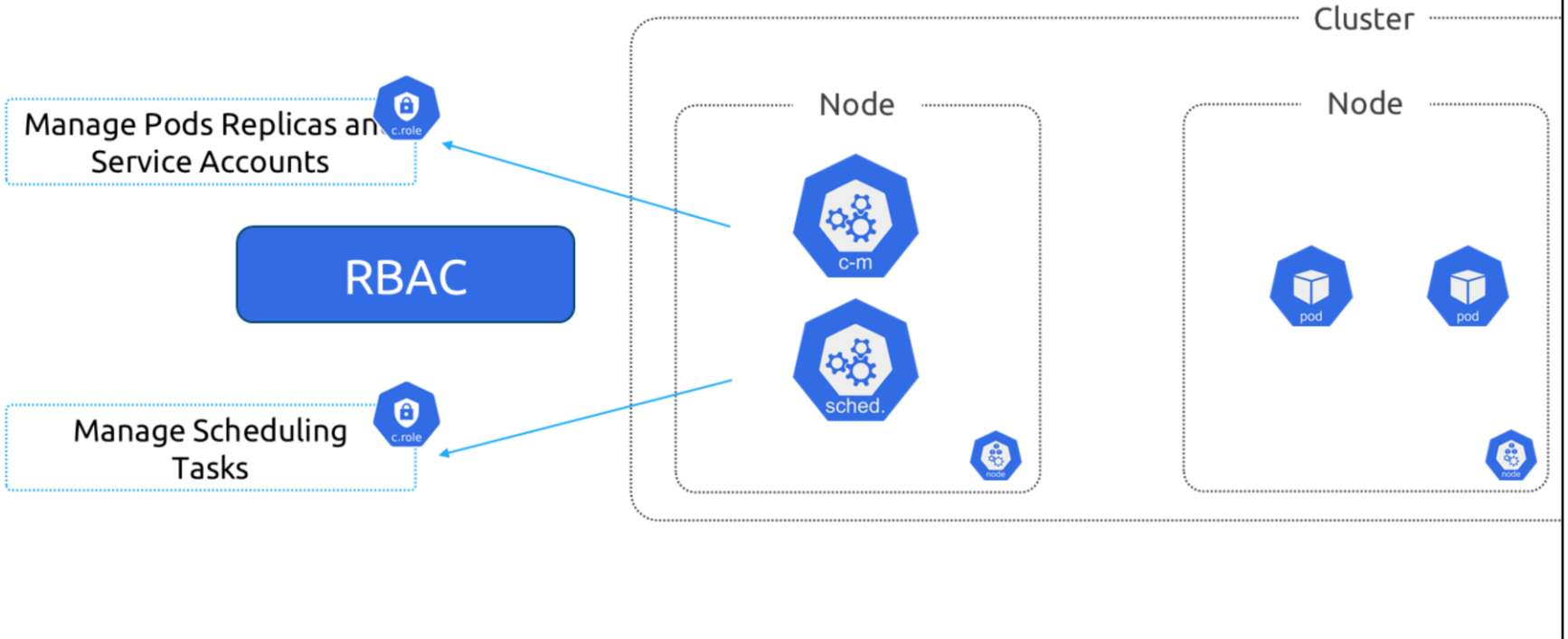
# Kubernetes Controller Manager and Scheduler






You run them on dedicated nodes separate from where the applications are running. This way, if a security breach occurs in one of the application nodes, it doesn't directly impact these critical components. For instance, during a security incident, attackers might gain access to an application pod. If the Controller Manager and Scheduler are isolated on separate nodes, they remain protected, reducing the risk of a full cluster compromise.

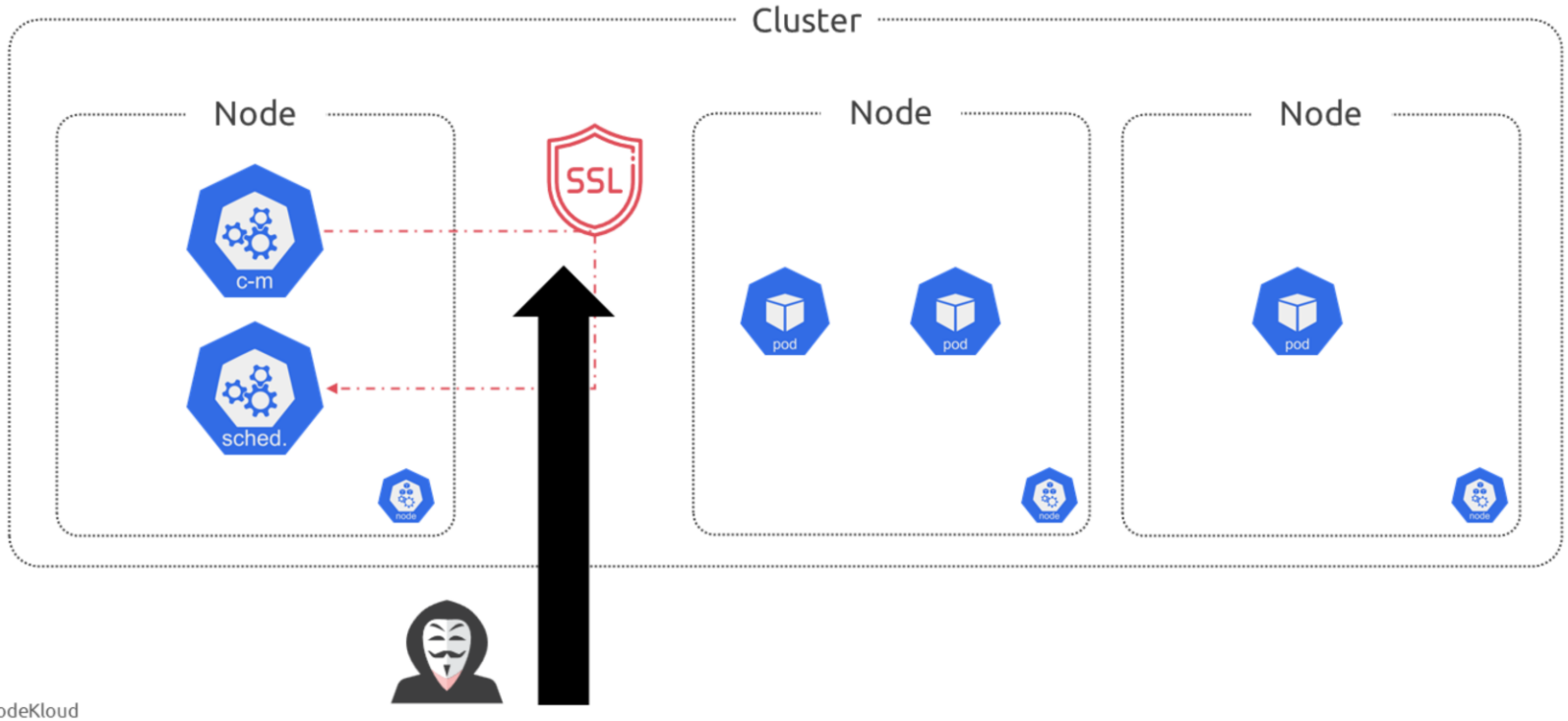
# Kubernetes Controller Manager and Scheduler






In another scenario, you set up Role-Based Access Control (RBAC) to limit what the Controller Manager and Scheduler can do. Suppose your Controller Manager only needs to manage pod replicas and service accounts, and the Scheduler only needs to manage scheduling tasks. You configure RBAC rules to give them permissions only for those tasks. This way, even if someone tries to exploit these components, they won't be able to perform unauthorized actions, like accessing secrets or altering network policies. For example, if a malicious actor gains limited access, RBAC ensures they can't escalate privileges to cause more damage.

# Kubernetes Controller Manager and Scheduler






Communication security is also crucial. Let's say your Controller Manager and Scheduler need to communicate with the API server and etcd. By using TLS (Transport Layer Security), you ensure all data transferred between these components is encrypted. This prevents attackers from intercepting sensitive information. Imagine if an attacker tries to eavesdrop on the communication between these components and the API server. With TLS, all data is encrypted, making it unreadable to the attacker. Regularly renewing TLS certificates ensures this secure communication remains intact over time.

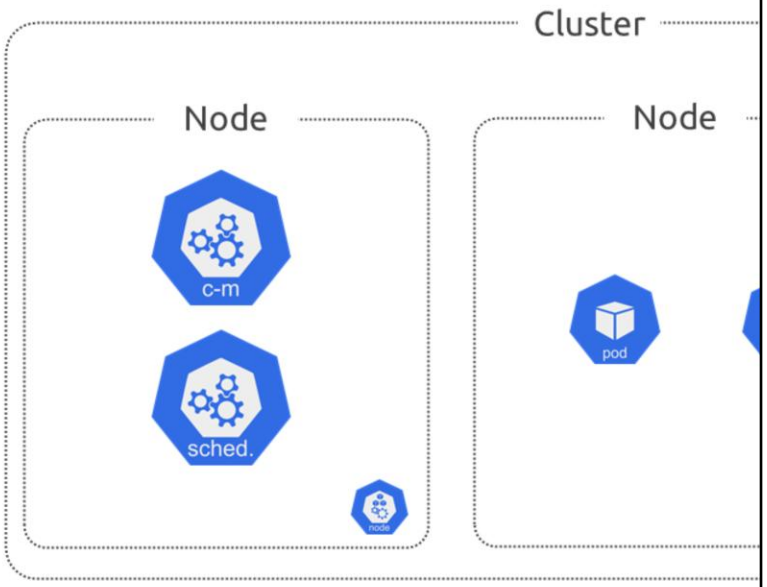
We will be talking more about RBAC and TLS in future lessons, diving deeper into how these security measures work and how you can implement them effectively.

# Kubernetes Controller Manager and Scheduler

Prometheus

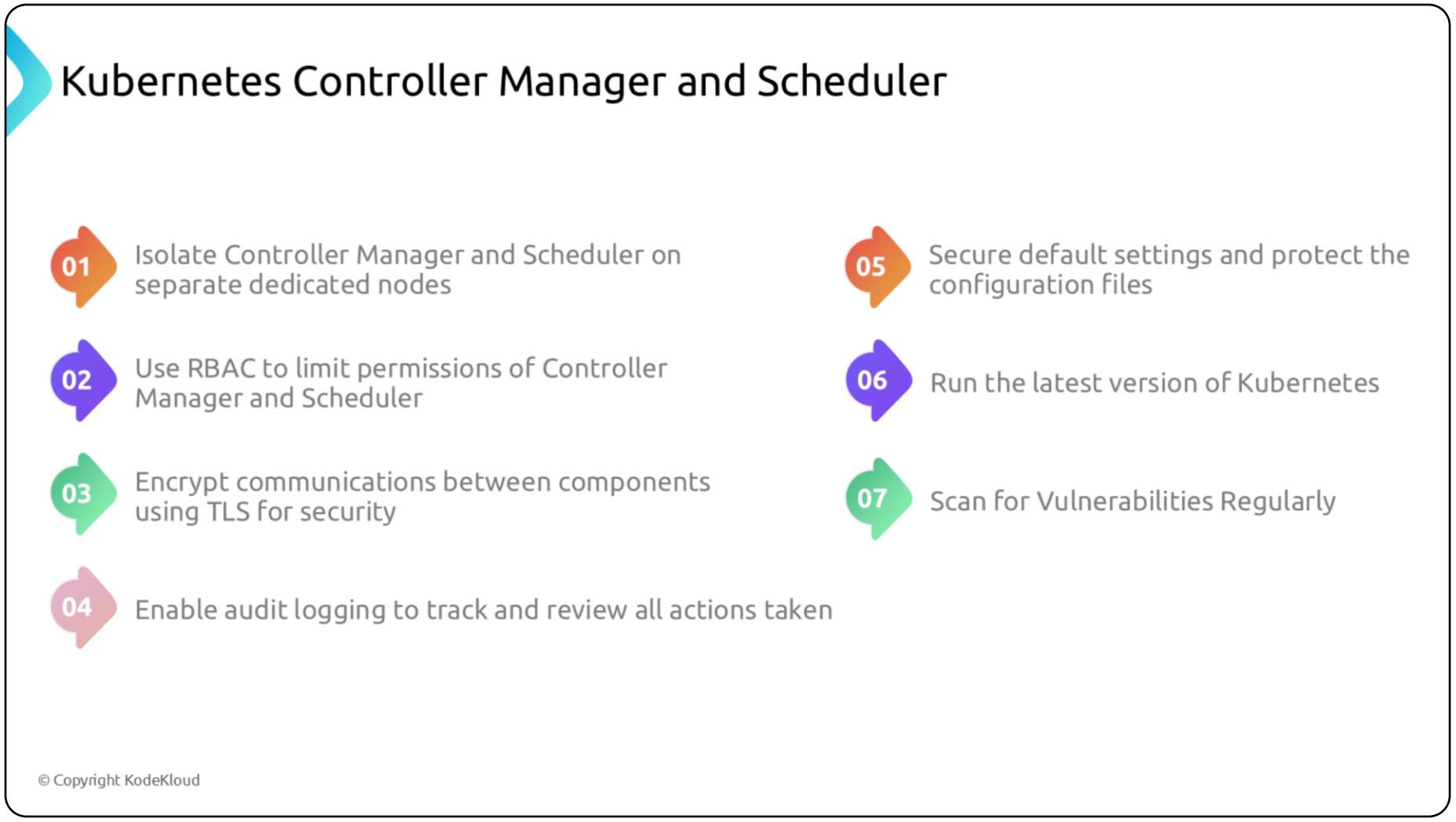
Grafana

```
Event Type: PATCH
Timestamp: 2024-11-09T12:34:56Z
Description: ReplicaSet controller updated deployment "my-deployment" in namespace
"default" to adjust replicas to 3. Request by user "system:serviceaccount:kube-
system:replicaset-controller" from IP 10.10.10.10. Status: 200 OK.

Event Type: GET
Timestamp: 2024-11-09T12:35:12Z
Description: Kube Controller Manager retrieved config map "kube-root-ca.crt" in
namespace "kube-system". Request by user "system:serviceaccount:kube-system:kube-
controller-manager" from IP 10.10.10.11. Status: 200 OK.

Event Type: PATCH
Timestamp: 2024-11-09T12:35:45Z
Description: Horizontal Pod Autoscaler adjusted settings for "my-app-autoscaler" in
namespace "default" to min replicas 2, max replicas 10, target CPU utilization 80%.
Request by user "system:serviceaccount:kube-system:horizontal-pod-autoscaler" from
IP 10.10.10.12. Status: 200 OK.
```






To keep track of what happens in your cluster, you enable audit logging for the Controller Manager and Scheduler. This records all actions taken, providing a detailed log that you can review. For example, if unusual activity occurs, you can refer to the audit logs to understand what happened and who performed the actions. Tools like Prometheus and Grafana help monitor these activities, and setting up alerts ensures you are notified of any suspicious behavior, allowing you to respond quickly.




So we discussed about Isolating Controller Manager and Scheduler on separate dedicated nodes.

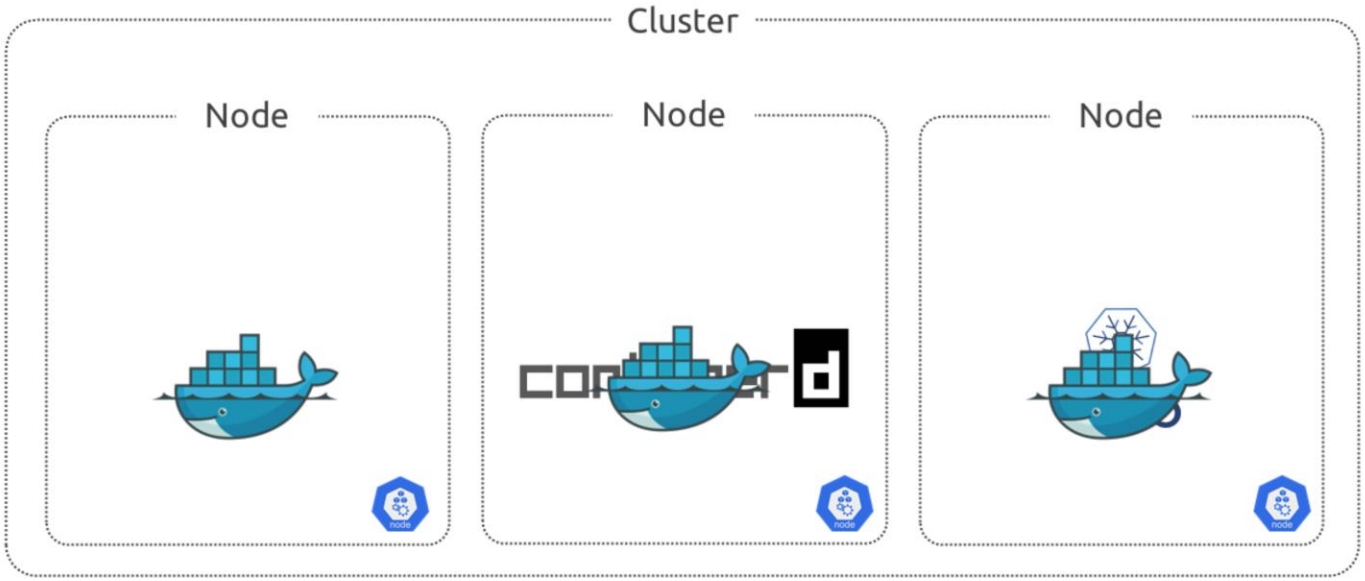
Apart from that you also make sure to use secure default settings and protect the configuration files of the Controller Manager and Scheduler. Always running the latest version of Kubernetes helps you stay up to date with security patches. For instance, if a new vulnerability is discovered, updating Kubernetes ensures your components are protected against it.

Regularly scanning for vulnerabilities helps you identify and fix potential security issues before they can be exploited.

That’s all for the Controller Manager and Scheduler security. In the next section, we will talk about kubelet security.

# Securing
# Container Runtime

# Kubernetes Controller Manager and Scheduler






Each compute node in a Kubernetes cluster has a container runtime engine. Most Kubernetes applications historically used Docker as the container runtime engine. However, Kubernetes now supports other runtime engines such as containerd and CRI-O , which are more suited to Kubernetes' architecture and security needs. Securely configuring your container runtime engine involves following the security best practices issued by the engine you use.

# Container Runtime Security

Early Docker allowed containers to run with root access on the host.



In the early days of Docker, a major vulnerability allowed containers to run with root-level access on the host machine. This was a huge risk because attackers could exploit this access to take control of the host, potentially leading to data breaches and system compromises.

# Container Vulnerabilities

| Vulnerability Name | CVE ID | Affected Systems | Description | More Information |
|---|---|---|---|---|
| Dirty Cow | CVE-2016-5195 | Linux kernel | A flaw in the Linux kernel that allowed containers to gain unauthorized root access, compromising the entire host. | here |
| RunC Container Breakout | CVE-2019-5736 | Docker, Kubernetes, etc. | Allowed attackers to overwrite the host runC binary and gain root access. | Link |
| Docker Container Escape | CVE-2014-3499 | Docker | Allowed a malicious container to access sensitive files on the host. | Link |
| containerd DoS Vulnerability | CVE-2020-15257 | containerd | Allowed a malicious container to cause a denial of service (DoS) on the host. | Link |
| CRI-O Container Escape | CVE-2021-20291 | CRI-O | Allowed a container escape, potentially giving attackers access to the host. | Link |



The "Dirty Cow" vulnerability (CVE-2016-5195) is another notable example that we saw an example of earlier in this course. It was a flaw in the Linux kernel that allowed containers to gain unauthorized root access, compromising the entire host. The RunC container breakout (CVE-2019-5736) affected Docker, Kubernetes, and other container systems. It allowed attackers to overwrite the host runC binary and gain root access.

Another example is the Docker container escape vulnerability (CVE-2014-3499), where a malicious container could access

sensitive files on the host.

More recently, containerd had a vulnerability (CVE-2020-15257) that allowed a malicious container to cause a denial of service (DoS) on the host.

CRI-O also had a bug (CVE-2021-20291) allowing a container escape, potentially giving attackers access to the host.

# Updating and Patching Regularly


Keep your container runtime updated regularly.



Keeping your container runtime updated is one of the simplest yet most effective security measures. Regular updates include patches for known vulnerabilities.

# Updating and Patching Regularly

Update containerd on Ubuntu

```
sudo apt-get update
sudo apt-get install containerd
```


Use package managers to update your runtime.



For example, if you are using containerd on Ubuntu, you can update it with:

sudo apt-get update

```
sudo apt-get install containerd
```

For other systems or runtimes, refer to the documentation provided by your container runtime or Kubernetes distribution.

Regularly checking for and applying updates ensures that any newly discovered vulnerabilities are patched before they can be exploited. This is crucial because many attacks exploit known vulnerabilities that have been fixed in newer versions.

# Running Containers With Least Privileges


Specify non-root user and group in your pod configuration.



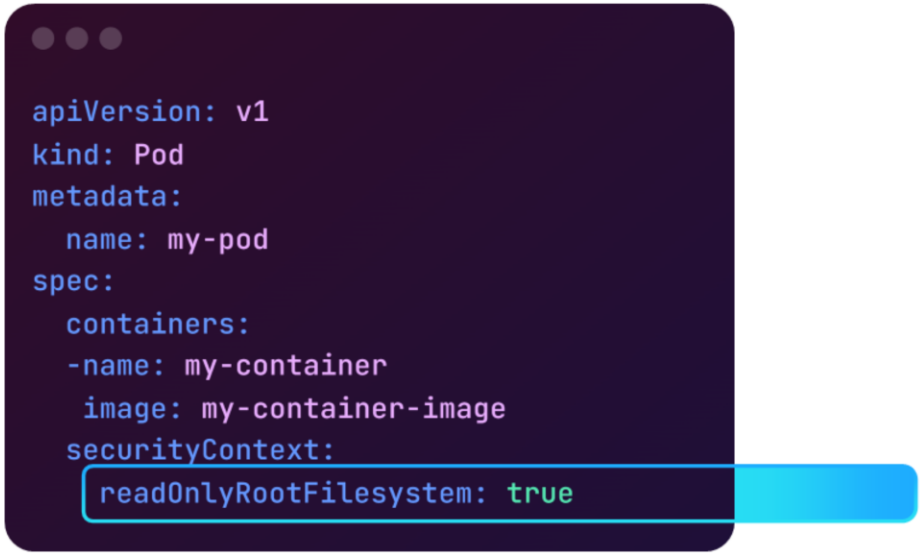
Running containers with the least privileges necessary helps to minimize the risk if a container is compromised. Avoid running containers as the root user. Instead, run them with a non-root user whenever possible. This limits the potential damage an attacker can cause.

In Kubernetes, you can set the runAsUser and runAsGroup fields in your pod specification to specify that containers should

run as a non-root user and group. For example, the shown configuration ensures that the container runs with user ID 1000 and group ID 3000:

This practice reduces the risk of privilege escalation and limits the impact of a potential security breach.

# Using Read-Only Filesystems




Set the readOnlyRootFilesystem field in your pod configuration.



Using read-only filesystems can prevent unauthorized modifications to the container's filesystem. This reduces the attack surface and helps ensure that your containers remain secure.

In Kubernetes, you can set the readOnlyRootFilesystem field in your pod specification to make the container's filesystem read-only:

Making filesystems read-only helps to protect against attacks that involve modifying the container's files. This is particularly important for containers that do not need to write to disk during normal operation.

# Limiting Resource Usage


Set CPU and memory limits in your pod configuration.



Setting resource limits on CPU and memory usage prevents any single container from consuming all the host’s resources. This helps maintain the performance and availability of your system, even if one container behaves unexpectedly.

In this example,

By limiting resources, you prevent denial-of-service (DoS) attacks where a malicious or misconfigured container exhausts the host’s resources, causing disruption to other services.

# Using Security Profiles

SELinux

```
apiVersion: v1
kind: Pod
metadata:
  name: selinux-demo
spec:
  containers:
  - name: nginx
    image: nginx
    securityContext:
      seLinuxOptions:
        user: "system_u"
        role: "system_r"
        type: "spc_t"
        level: "s0:c123,c456"
```

Configure SELinux

AppArmor using Kubernetes Annotations

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  annotations:
    container.apparmor.security.beta.Kubernetes.io/my-
container: localhost/my-apparmor-profile
spec:
  containers:
  -name: my-container
   image: my-container-image
```

Apply AppArmor profiles using Kubernetes annotations.



Implementing security profiles like SELinux and AppArmor adds an additional layer of security by enforcing mandatory access controls on containers. These profiles restrict what containers can do, reducing the risk of exploitation.

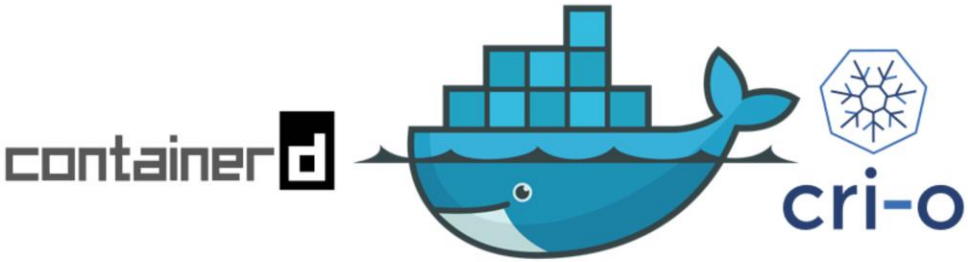
SELinux (Security-Enhanced Linux) is a security module in the Linux kernel that provides a mechanism for supporting access control security policies. SELinux policies define how processes and users can access resources on a system.

AppArmor (Application Armor) is another security module for the Linux kernel, providing a different approach to mandatory access control. AppArmor uses profiles to restrict the capabilities of individual programs, preventing them from performing unauthorized actions.

For SELinux, ensure it is enabled and configured correctly on your nodes. For AppArmor, you can apply profiles using Kubernetes annotations:

Security profiles help enforce good security practices by limiting the actions containers can perform. This can help prevent containers from performing actions that could compromise the host.

# Transitioning to Supported Runtimes




Transition to supported container runtimes like containerd and CRI-O.



Also, With support for Docker deprecated in Kubernetes,

# Transitioning to Supported Runtimes


Transition to supported container runtimes like containerd and CRI-O.

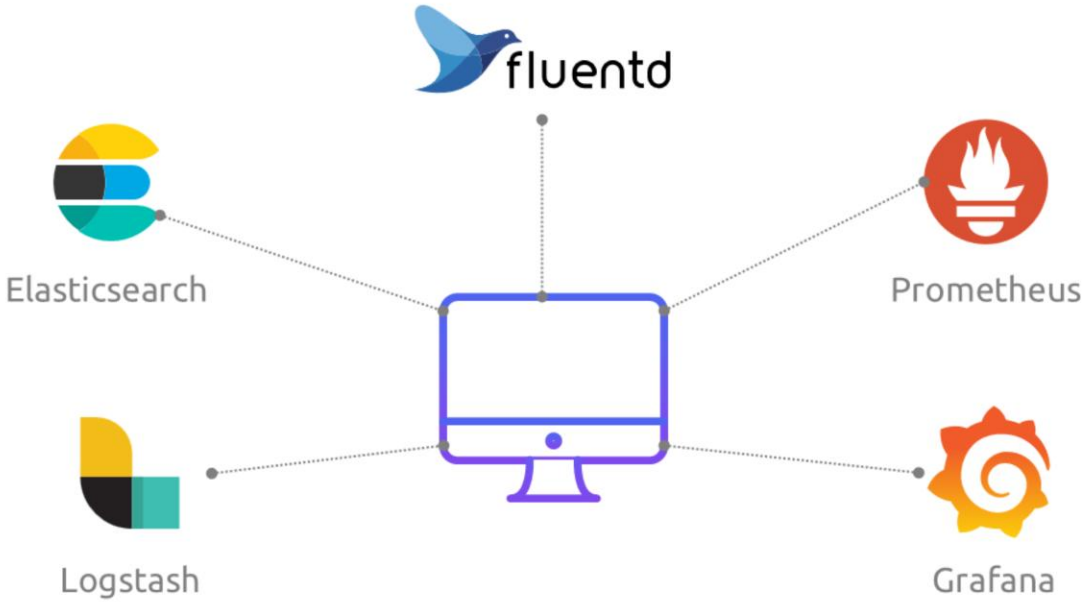


it's important to transition to supported container runtimes like containerd or CRI-O. These runtimes are optimized for Kubernetes, providing better performance and security.

Ensure your Kubernetes nodes are configured to use these supported runtimes by updating your cluster configuration and node settings. Transitioning to these runtimes ensures compatibility with future Kubernetes releases and leverages their

security features.

# Transitioning to Supported Runtimes

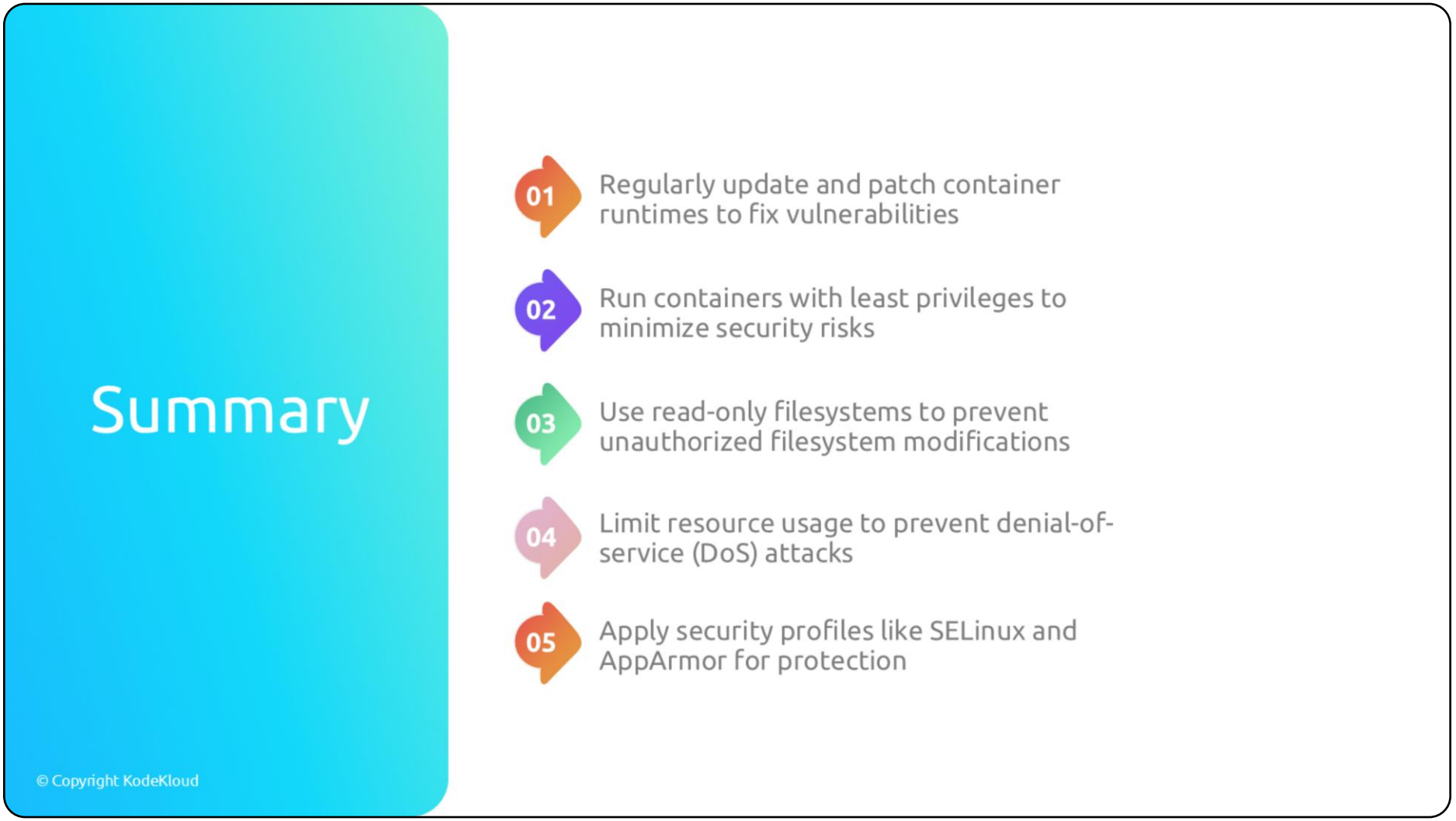



Use tools like Fluentd, Logstash, and Prometheus for centralized logging and monitoring.



Finally, implementing monitoring and logging with tools like Fluentd, Logstash, or Elasticsearch for centralized logging, and Prometheus or Grafana for runtime behavior monitoring helps detect and respond to security incidents promptly.

Enabling audit logs provides a detailed record of actions, aiding in forensic analysis and understanding the impact of security incidents. Regularly review and update your security practices to stay ahead of potential threats and ensure the ongoing security of your container runtime.




So to secure container runtimes we spoke about regulary…

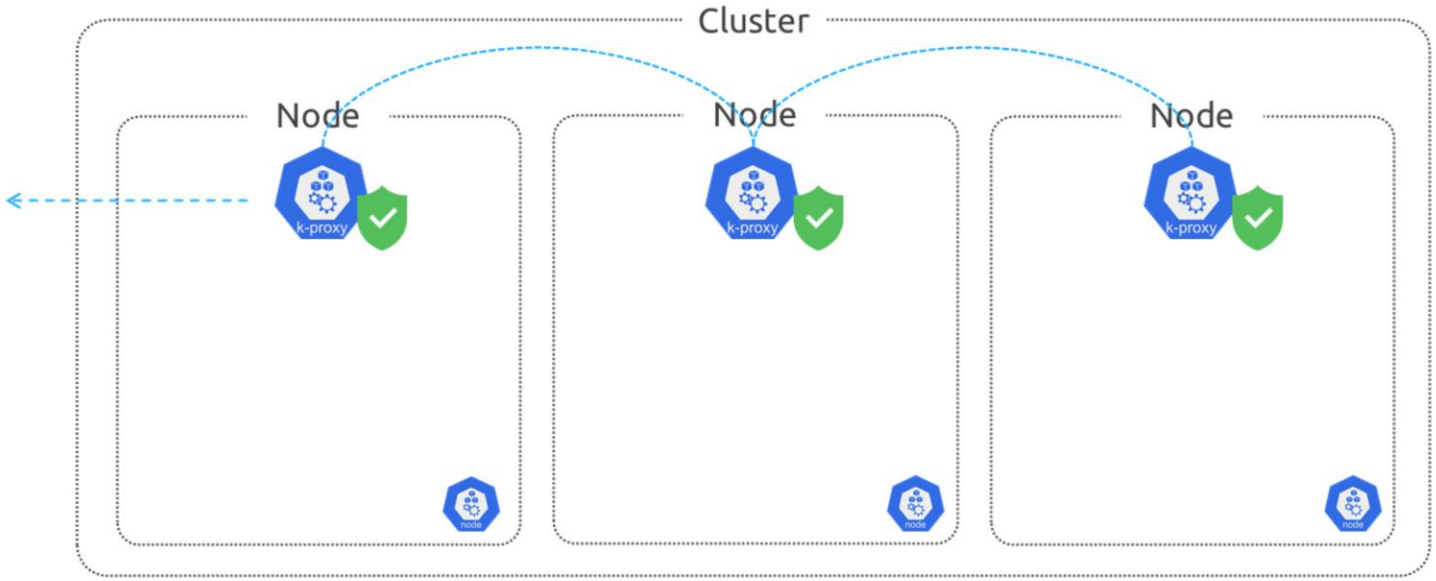
# Summary

06 Transition to supported runtimes like containerd or CRI-O

07 Implement monitoring and logging for runtime behavior detection

# Securing Kube-Proxy

# Kubernetes Controller Manager and Scheduler






Kube-proxy is a network proxy that runs on each node in your Kubernetes cluster, maintaining network rules. It ensures that your nodes can communicate with internal and external resources as required and allowed. Securing kube-proxy is essential to protect your Kubernetes environment from vulnerabilities and potential attacks. In this lesson, we'll cover important security measures to safeguard kube-proxy.

# Kubeconfig File


Locate the kubeconfig file by running the following command.



First and most important to ensuring the security of kube-proxy is maintaining the security and integrity of its kubeconfig file. This file contains the configuration needed for kube-proxy to communicate with the Kubernetes API server

To start, you need to locate the kubeconfig file that kube-proxy uses. You can do this by running the command ps -ef | grep kube-proxy. This command lists all the running processes and then filters the results to show only those related to kube-proxy. In the output, look for the --config flag. This flag specifies the path to the kube-proxy config file.




Next, look into the kube-proxy file and identify the kubceconfig file.

# Kubeconfig File


Check the file permissions to ensure they are set to 644 or stricter.



Once you have the location of the kube-proxy config file, the next step is to check its permissions. You can do this by running stat -c %a <kube-proxy config file>. The stat command displays detailed information about the file, and the -c %a option specifically shows the file's permissions in numeric format. Ensuring that the permissions are set to 644 or stricter is crucial because this setting allows only the file owner to write to the file while others can only read it.

# Kubeconfig File


Verify that the file is owned by root to prevent unauthorized access



To further secure the kubeconfig file, you should also check its ownership. Run the command stat -c %U:%G <kube-proxy config file>, where the -c %U:%G option shows the file's owner and group. The output should be root:root, meaning that only the root user has access to the file. This setting prevents unauthorized users from modifying the kubeconfig file. root:root ownership prevents non-root users from modifying the file, which is a good security measure, as kube-proxy does not need write access by other users to perform its function.Unauthorized users attempting to access or alter the file would be restricted by both ownership and permissions, which is ideal.

# Securing Communication

```
joe@ubuntu:!~$ cat /var/lib/kube-proxy/kubeconfig.conf
apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
    server: https://controlplane:6443
  name: default
contexts:
- context:
    cluster: default
    namespace: default
    user: default
  name: default
current-context: default
users:
- name: default
  user:
    tokenFile: /var/run/secrets/kubernetes.io/serviceaccount/token
```


Securing communication between kube-proxy and the API server is crucial.



Securing communication between kube-proxy and the Kubernetes API server is crucial. Ensure that this communication is encrypted using TLS to prevent eavesdropping and tampering. In this case if you open the kubeconfig file used you'll see the ca cert configured to validate the API server's TLS certificate. and the kube-proxy itself uses a service account token to authenticate to kube-api-server.

# Audit Logs

```yaml
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
  # Log all requests from the kube-proxy user
  - level: Metadata
    users: ["system:kube-proxy"]
  # Log all other high-level metadata (optional)
  - level: Metadata
    resources:
      - group: ""
        resources: ["pods", "services", "endpoints"]
  # Default level for everything else
  - level: None
```

Audit Logs

```
tail -f /var/log/audit/audit.log | jq .objectRef.resource
```

Enable auditing in Kubernetes to log all actions performed by kube-proxy.



Audit logs play a significant role in kube-proxy security. Enabling auditing in Kubernetes to log all actions performed by kube-proxy allows you to track changes and identify unauthorized access. Regularly reviewing these logs helps in detecting and addressing suspicious activities promptly.

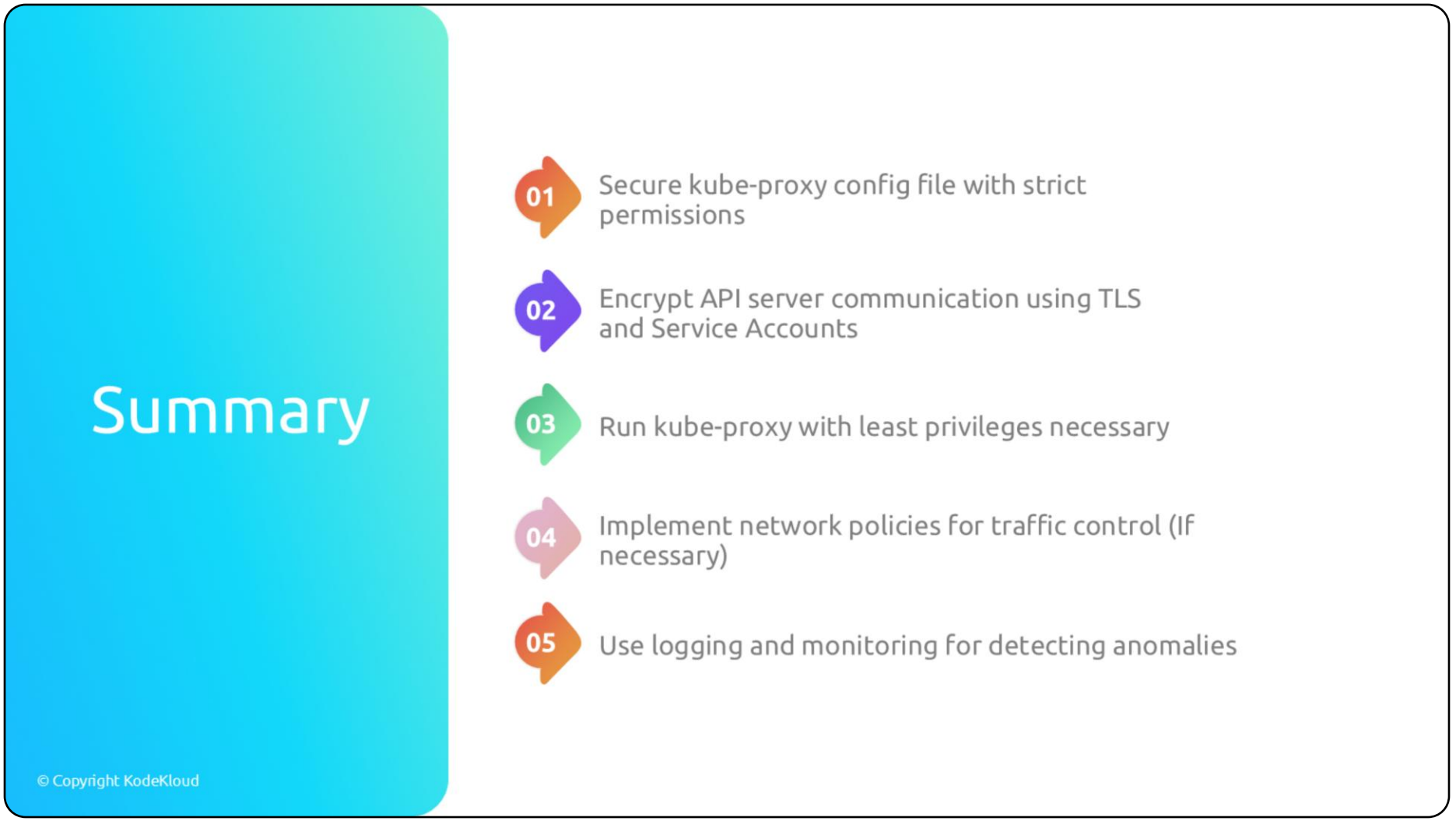
Use this configuration to create a policy to log everything kube-proxy does.

# Regular Updates and Patches


Regular updates and patches are fundamental for security.



Keeping kube-proxy updated with the latest patches is fundamental. Regular updates often include important security fixes for known vulnerabilities. Using automated systems for updates can help ensure that kube-proxy and other Kubernetes components are always running the most secure versions.




To summarize, to secure kube-proxy you must secure

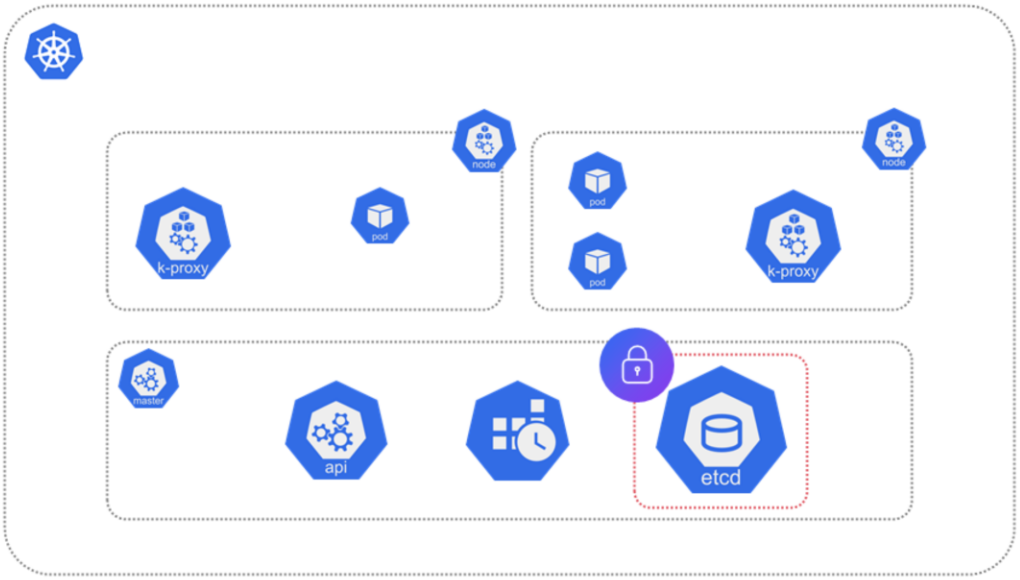
# Summary

06 Regularly update and patch kube-proxy for security

07 Enable audit logs to track kube-proxy actions

# Securing ETCD

# Safeguarding ETCD – Important Security Measures




Cluster configuration data

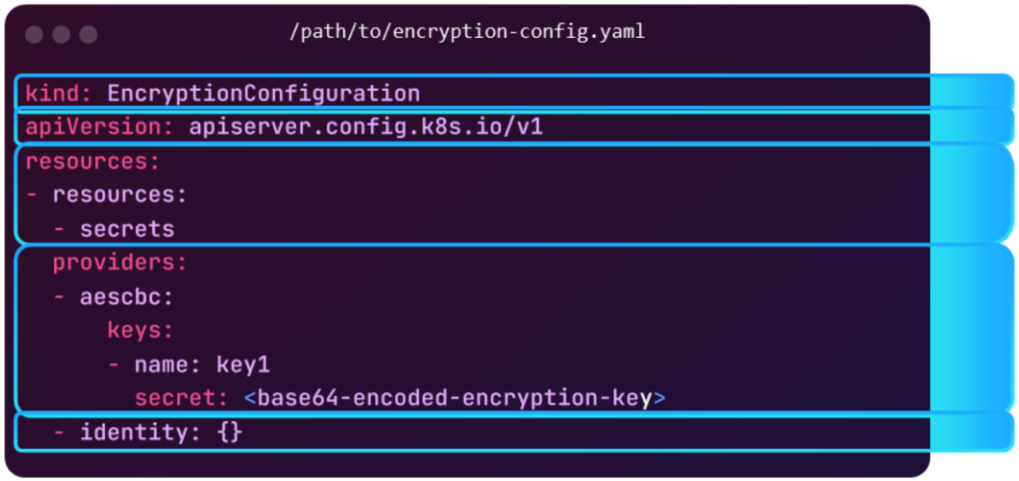
Secrets and state information

Certificates and keys



Etcd is a critical component in Kubernetes, acting as the key-value store that holds the configuration and state data of the cluster. Securing etcd is essential to protect this sensitive information from vulnerabilities and potential attacks. In this lesson, we'll cover important security measures to safeguard etcd.

# Enabling Data Encryption at Rest




Specifies the type of Kubernetes object: EncryptionConfiguration

Specifies the Kubernetes API version: apiserver.config.k8s.io/v1

Lists the resources to be encrypted, like secrets

Defines encryption providers, using aescbc as the algorithm

Fallback provider in case aescbc is unavailable



1 - Enabling Data Encryption at rest

One of the most important steps in securing etcd is to ensure that its configuration and data are protected. This begins with modifying the etcd pod specification file, which is typically located at /etc/kubernetes/manifests/etcd.yaml.

First, you need to make sure that encryption is enabled for the data stored in etcd. Encryption ensures that sensitive

information is protected, even if an attacker gains access to the etcd data store.

Here’s how you can enable encryption:

1. Create an Encryption Configuration File:

You need to create an encryption configuration file that Kubernetes will use to encrypt data. Here is an example configuration :

Let's break down the important parts:

- kind: Specifies the type of Kubernetes object. Here, it's an EncryptionConfiguration.
- apiVersion: Specifies the version of the Kubernetes API to use. Here, it's apiserver.config.k8s.io/v1.
- resources: Lists the types of resources to be encrypted. In this example, it’s set to secrets, meaning Kubernetes Secrets will be encrypted.
- providers: Defines the encryption providers to use. In this case, aescbc is used, which is a type of encryption algorithm. The keys section contains the encryption keys. The name is a label for the key, and secret is the actual encryption key, base64-encoded.
- identity: Acts as a fallback in case the aescbc provider cannot be used.

# Enabling Data Encryption at Rest

Encryption Configuration File

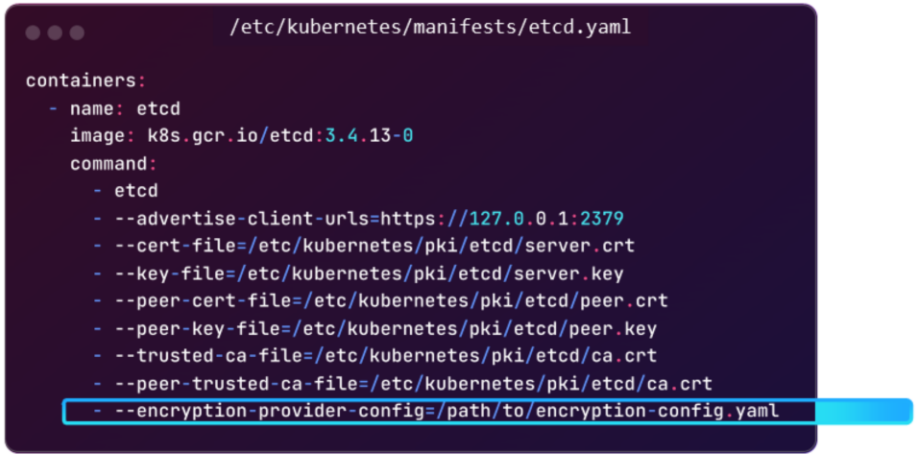
```
openssl rand -base64 32
```

Generating the encryption key



You should replace <base64-encoded-encryption-key> with your actual base64-encoded encryption key. You can generate a base64-encoded key using OpenSSL or another encryption tool.

# Enabling Data Encryption at Rest




The command includes the --encryption-provider-config flag pointing to the encryption configuration file



2. Update the Etcd Pod Specification:

Next, you need to update the etcd pod specification to use the encryption configuration file. Open the /etc/kubernetes/manifests/etcd.yaml file and add the --encryption-provider-config flag to the etcd command. Here's how to do it:

The --encryption-provider-config flag points to the encryption configuration file you created.

# Using TLS for Secure Communication

/etc/kubernetes/manifests/etcd.yaml

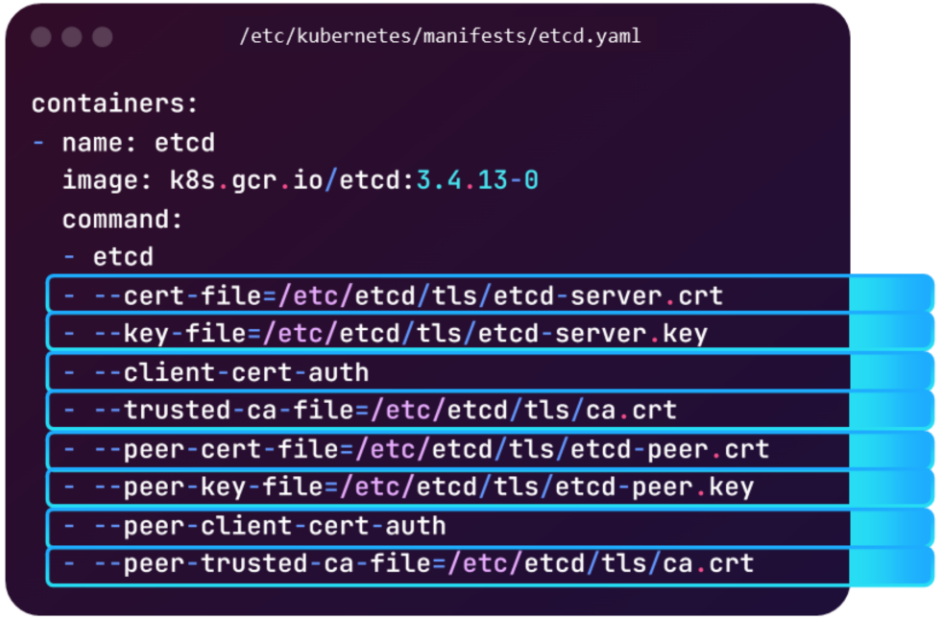
```
containers:
- name: etcd
  image: k8s.gcr.io/etcd:3.4.13-0
  command:
  - etcd
  - --cert-file=/etc/etcd/tls/etcd-server.crt
  - --key-file=/etc/etcd/tls/etcd-server.key
  - --client-cert-auth
  - --trusted-ca-file=/etc/etcd/tls/ca.crt
  - --peer-cert-file=/etc/etcd/tls/etcd-peer.crt
  - --peer-key-file=/etc/etcd/tls/etcd-peer.key
  - --peer-client-cert-auth
  - --peer-trusted-ca-file=/etc/etcd/tls/ca.crt
```

Updating etcd configuration



"Next, update the etcd configuration to include the paths to the TLS certificates in the etcd pod specification file."

# Using TLS for Secure Communication




Specifies the server's certificate file for secure identity presentation

Specifies the server's key file used with the certificate

Enables client certificate authentication to allow only trusted clients

Specifies the CA certificate for verifying client certificates

Specifies the server's certificate file for secure identity presentation

Specifies the server's key file used with the certificate

Enables client certificate authentication to allow only trusted clients

Specifies the CA certificate for verifying client certificates

Updating etcd configuration



First,
Generate TLS certificates for etcd and the clients that connect to it. You can use tools like openssl or cfssl to create the necessary certificates.

Next,

Update the etcd configuration to include the paths to the TLS certificates. Here's an example of what this might look like in

the /etc/kubernetes/manifests/etcd.yaml file:

So in this yaml configuration file,

--cert-file the path to the server's certificate file, ensuring the server can present its identity securely.

--key-file: The path to the server's key file, used alongside the certificate.

--client-cert-auth: Enables client certificate authentication, ensuring only trusted clients can communicate with etcd.

--trusted-ca-file: The path to the CA certificate used to verify client certificates.

--peer-cert-file: The path to the peer's certificate file for secure peer-to-peer communication.

--peer-key-file: The path to the peer's key file.

--peer-client-cert-auth: Enables peer client certificate authentication.

--peer-trusted-ca-file: The path to the CA certificate used to verify peer certificates.

# Regular Backups


Using etcd's Snapshot functionality



4 - Regular Backups

Regularly backing up the etcd data is crucial to ensure you can recover your cluster state in case of data loss or corruption. Use etcd's built-in snapshot functionality to create backups:

ETCDCTL_API=3 etcdctl snapshot save /path/to/backup.db --endpoints=<etcd-endpoints> --cacert=/path/to/ca.crt --

cert=/path/to/etcd-client.crt --key=/path/to/etcd-client.key

Let’s break down the command:

- ETCDCTL_API=3: Specifies the etcdctl API version to use. Version 3 is the latest and most commonly used version.
- etcdctl snapshot save: The command to take a snapshot of the etcd data.
- /path/to/backup.db: The path where the backup file will be saved. Replace this with your desired backup location.
- --endpoints=<etcd-endpoints>: Specifies the etcd endpoints to connect to. Replace <etcd-endpoints> with the actual addresses of your etcd instances.
- --cacert=/path/to/ca.crt: The path to the CA certificate used to verify the etcd server certificate.
- --cert=/path/to/etcd-client.crt: The path to the client certificate used to authenticate the client to the etcd server.
- --key=/path/to/etcd-client.key: The path to the client key used to authenticate the client to the etcd server.

Always, Store these backups in a secure and reliable location to ensure you can restore your cluster if needed.

# Regular Backups

```
ETCDCTL_API=3 etcdctl snapshot save /path/to/backup.db \
  --endpoints=<etcd-endpoints> \
  --cacert=/path/to/ca.crt \
  --cert=/path/to/etcd-client.crt \
  --key=/path/to/etcd-client.key
```

- Specifies the etcdctl API version to use (Version 3)
- Command to take a snapshot of the etcd data
- Path where the backup file will be saved
- Specifies the etcd endpoints to connect to
- Path to the CA certificate used to verify the etcd server certificate
- Path to the client certificate used to authenticate the client
- Path to the client key used to authenticate the client



Let’s break down the command:

- ETCDCTL_API=3: Specifies the etcdctl API version to use. Version 3 is the latest and most commonly used version.
- etcdctl snapshot save: The command to take a snapshot of the etcd data.
- /path/to/backup.db: The path where the backup file will be saved. Replace this with your desired backup location.

- --endpoints=<etcd-endpoints>: Specifies the etcd endpoints to connect to. Replace <etcd-endpoints> with the actual addresses of your etcd instances.
- --cacert=/path/to/ca.crt: The path to the CA certificate used to verify the etcd server certificate.
- --cert=/path/to/etcd-client.crt: The path to the client certificate used to authenticate the client to the etcd server.
- --key=/path/to/etcd-client.key: The path to the client key used to authenticate the client to the etcd server.

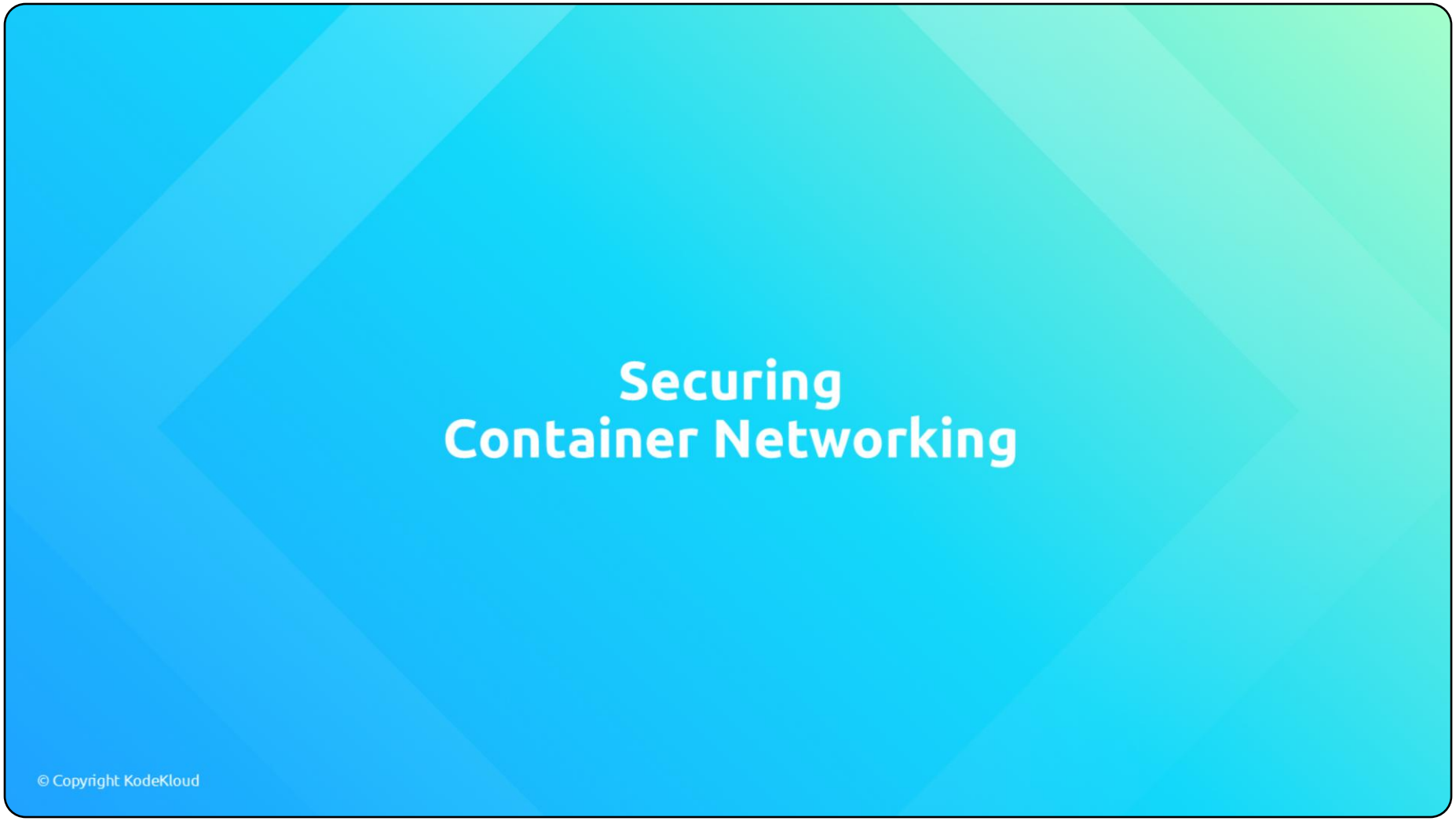
Always, Store these backups in a secure and reliable location to ensure you can restore your cluster if needed.

# Summary

01 Enable data encryption at rest for etcd security

02 Use TLS to secure etcd communication

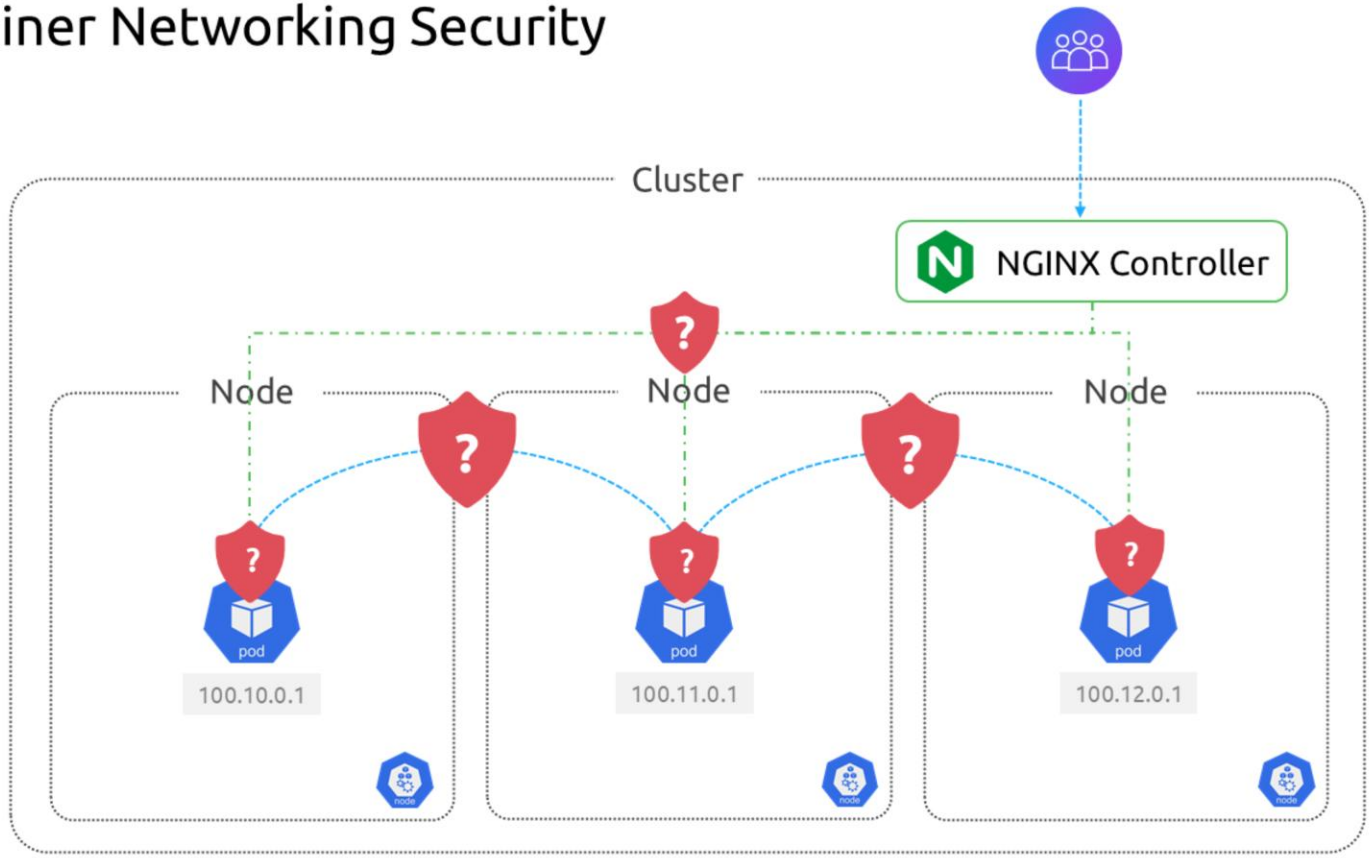
03 Regularly back up etcd data for recovery




By default, Kubernetes networking is designed to be simple and flat, allowing each pod in the cluster to get its own IP address. Containers within a pod share the same network namespace, including the IP address and network ports, enabling seamless communication.

This design allows all pods to communicate with each other without needing Network Address Translation (NAT) and lets pods reach services using standard DNS resolution. Additionally, external users can access services through ingress controllers or external load balancers.

# Container Networking Security






Narration:

By default, Kubernetes networking is designed to be simple and flat, allowing each pod in the cluster to get its own IP address. Containers within a pod share the same network namespace, including the IP address and network ports, enabling seamless communication.

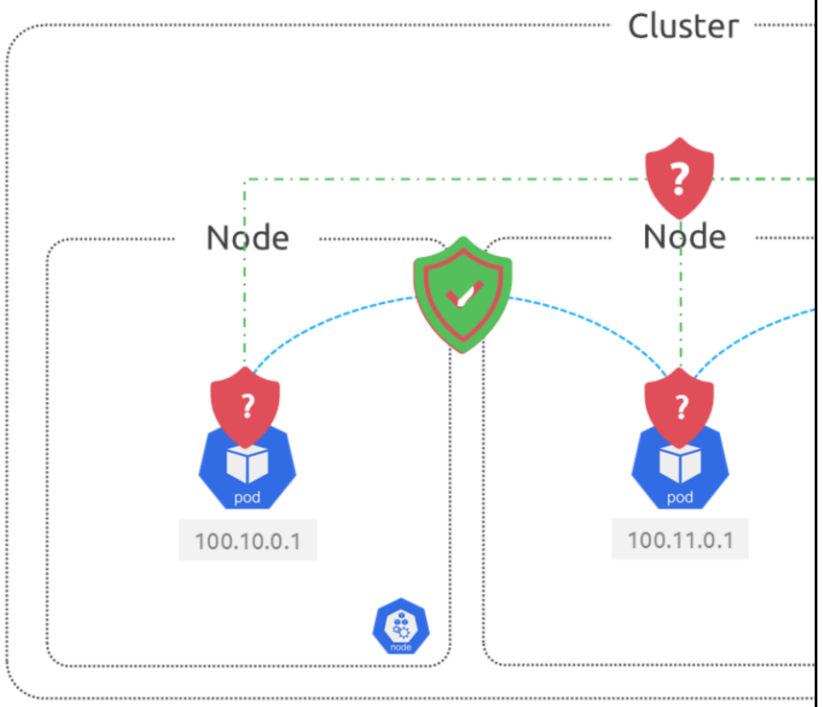
This design allows all pods to communicate with each other without needing Network Address Translation (NAT) and lets

pods reach services using standard DNS resolution. Additionally, external users can access services through ingress controllers or external load balancers.

While this flat network structure simplifies communication, it also necessitates robust security measures to prevent unauthorized access and ensure secure data transfer.

# Implementing Network Policies

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
  namespace: default
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
  ingress: []
  egress: []
```



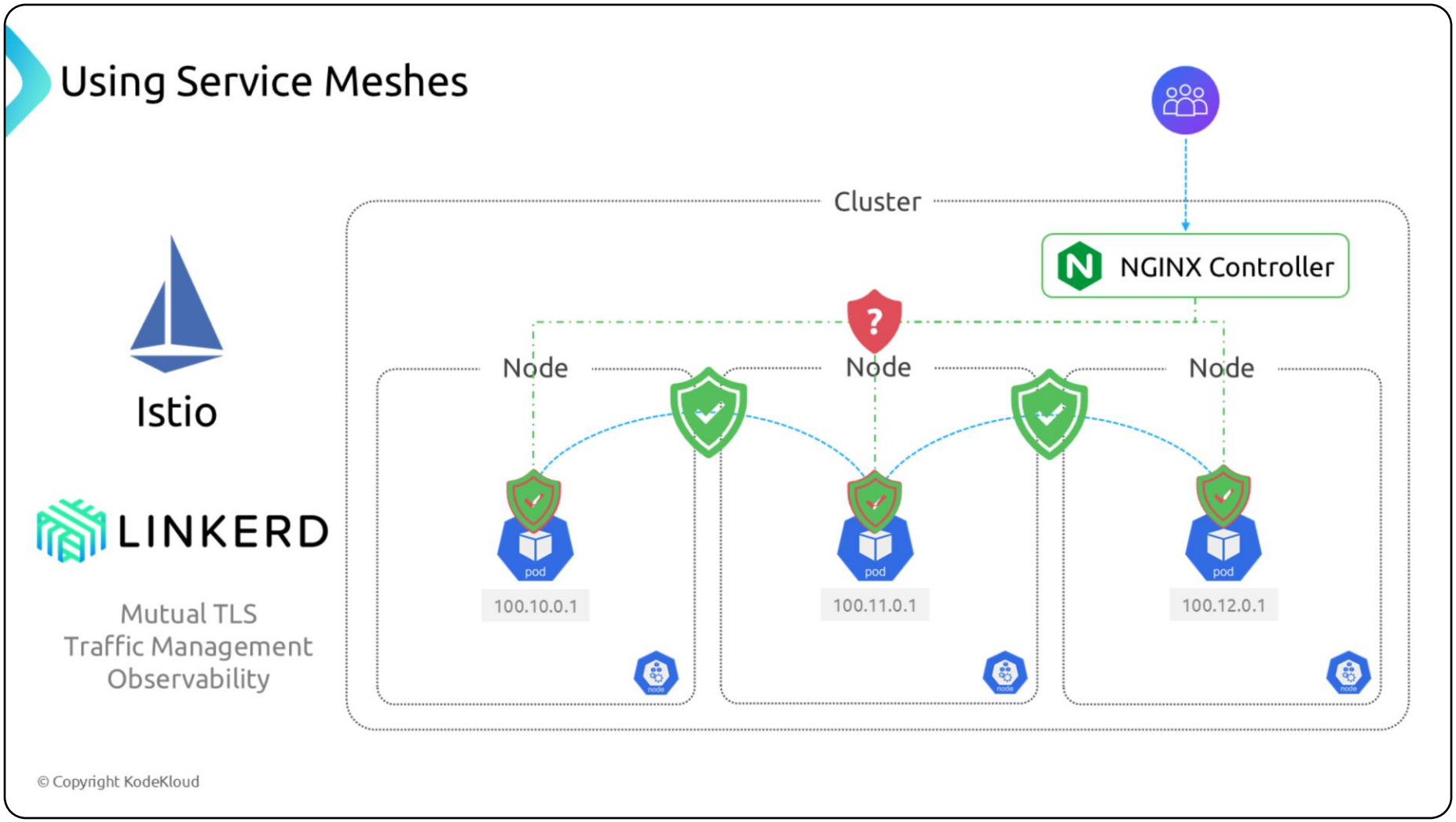



Let's start with Implementing Network Policies.

As mentioned, by default, Kubernetes allows all traffic between pods, but you can create network policies to restrict this communication based on specific rules.
You can see an example of a network policy that restricts traffic to a specific pod on the screen.

This policy denies all incoming and outgoing traffic to pods in the default namespace, providing a baseline level of security. By defining what traffic is allowed and denied, you can better control and protect the communication between pods. In the next section we have a dedicated, detailed lesson around network policies.

# Using Service Meshes

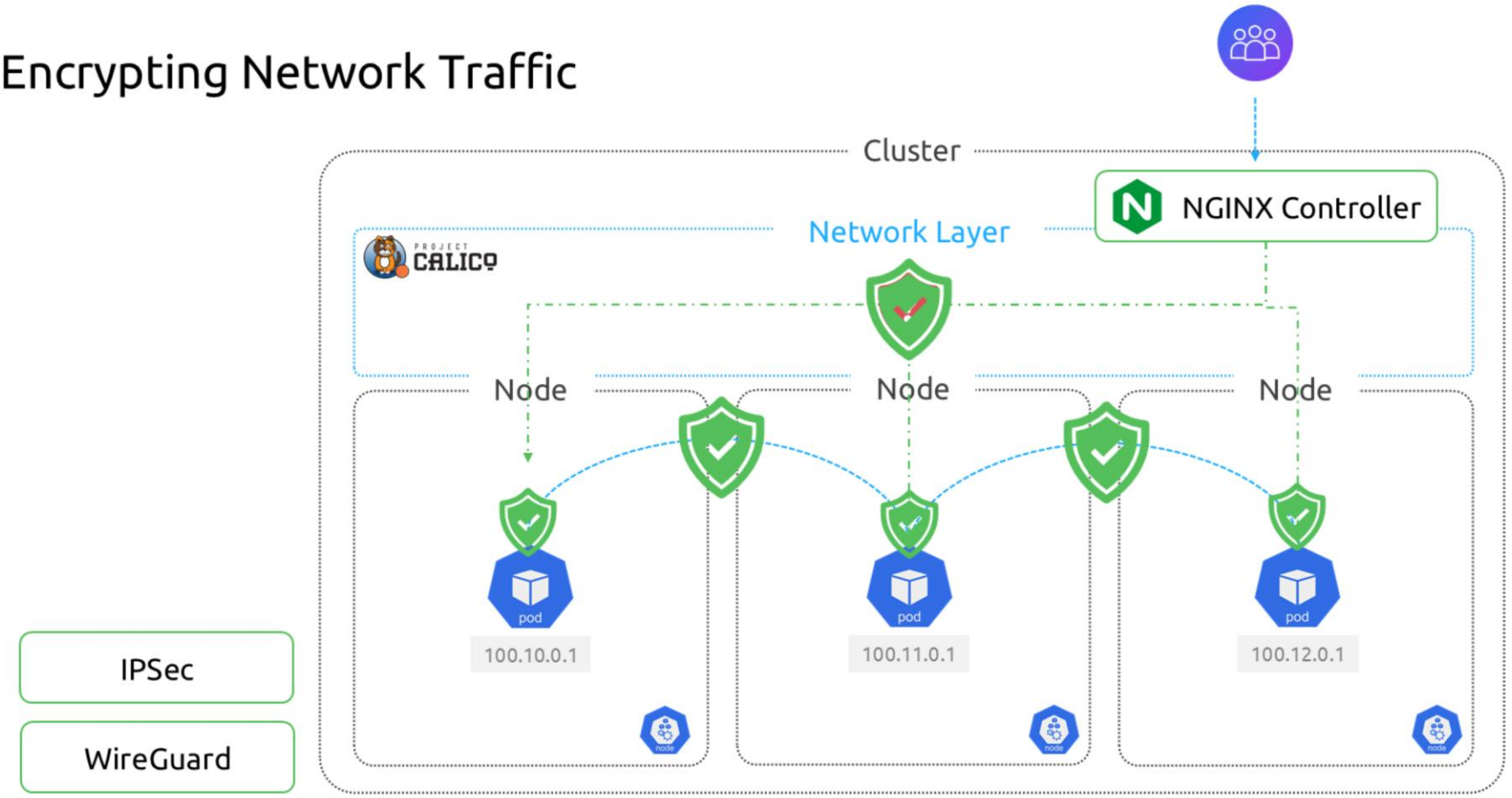





Service meshes like Istio or LinkerD provide advanced networking features, including mutual TLS for encrypted communication, traffic management, and observability. They help secure, control, and monitor the traffic between microservices.

For example, Istio can be used to enforce mutual TLS, ensuring that all communication between services is encrypted and authenticated. This significantly reduces the risk of man-in-the-middle attacks.

# Encrypting Network Traffic

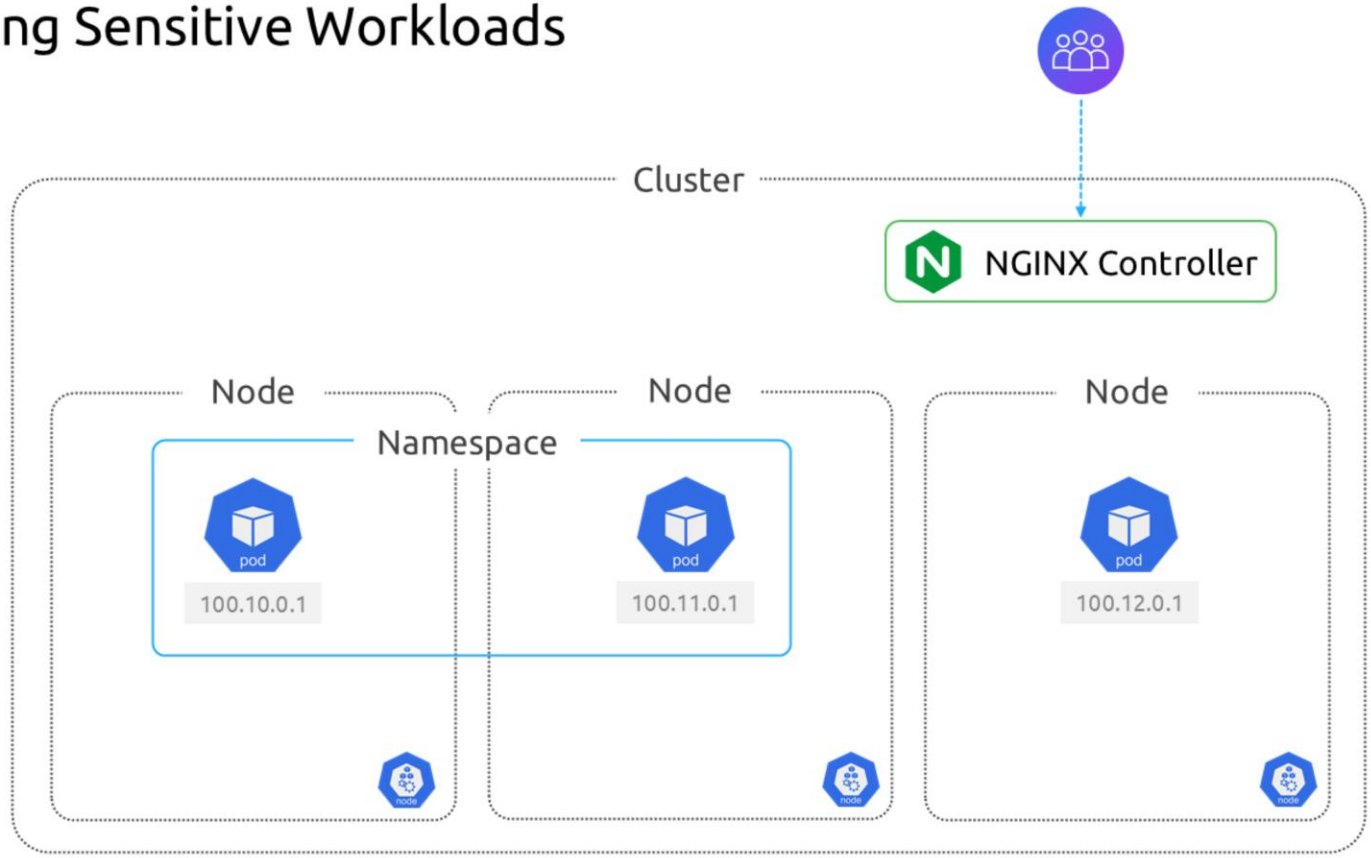





Encrypting network traffic between containers is crucial for protecting data in transit. Kubernetes supports network encryption through various mechanisms, including using IPsec or WireGuard to encrypt traffic at the network layer.

For instance, you can use a CNI plugin like Calico to enable IPsec encryption for network traffic. This ensures that all data transferred between nodes is encrypted, preventing unauthorized access and ensuring the integrity and confidentiality of your data.

# Isolating Sensitive Workloads






Isolating sensitive workloads by using namespaces and network policies can help contain potential security breaches. By segregating workloads into different namespaces and applying strict network policies, you can limit the blast radius of any security incident.

This isolation helps ensure that even if one part of your system is compromised, the impact is minimized and does not affect other critical parts of your application.

# Summary

01 Implement network policies to control pod traffic flow

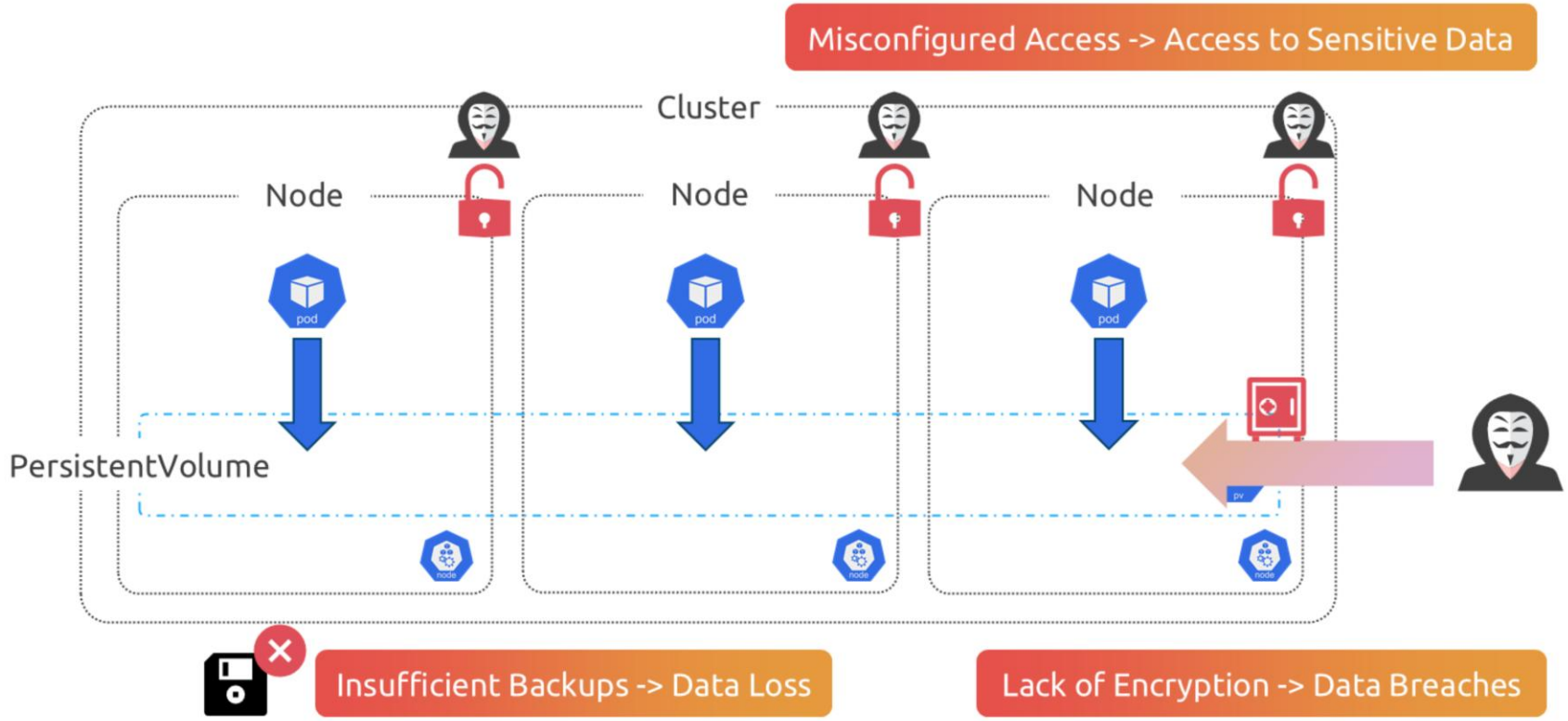
02 Use service meshes for encrypted, secure service communication

03 Encrypt network traffic between containers using IPsec or WireGuard

04 Isolate sensitive workloads with namespaces and network policies

# Securing Storage

# Storage in Kubernetes






In Kubernetes pods rely on PVs and PVCs – that's persistent volumes and persistent volume claims to access storage.

Storage security issues can have significant impacts on Kubernetes environments.

For example misconfiguration of storage access can allow unauthorized users to access sensitive data.

Lack of encryption can lead to data breaches where attackers intercept and read data stored on persistent volumes.

Another common issue involves insufficient backup strategies, which may result in data loss during system failures or ransomware attacks.

These incidents highlight the need for robust storage security measures to protect data integrity, confidentiality, and availability.

# Using Encryption

AWS EBS

Azure Disk Storage

Google Cloud Persistent Disk



Encrypting data both at rest and in transit is crucial for protecting sensitive information. Kubernetes supports encryption for data stored in etcd and for Persistent Volumes. We have already discussed etcd data encryption in previous lessons.

Many storage providers support encryption at the disk level. For instance, AWS EBS, Azure Disk Storage, and Google Cloud Persistent Disks all offer encryption options.

# Using Encryption


Enabling encryption in StorageClass



Here’s how you can enable encryption by setting the encrypted parameter to true in the StorageClass: by setting the encrypted parameter to true.

This configuration ensures that all data stored on these volumes is encrypted, providing an additional layer of security.

# Implementing Access Controls

Example: Granting read-only access

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pvc-reader
rules:
apiGroupstype: [""]
  resources: ["persistentvolumeclaims"]
  verbs: ["get", "list"]
```

Assigning roles with RoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pvc-binding
  namespace: default
subjects:
kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pvc-reader
  apiGroup: rbac.authorization.k8s.io
```



Next, let's talk about Access Controls.

Access controls ensure that only authorized users and services can access storage resources. We use Role-Based Access Control (RBAC) to manage permissions effectively.

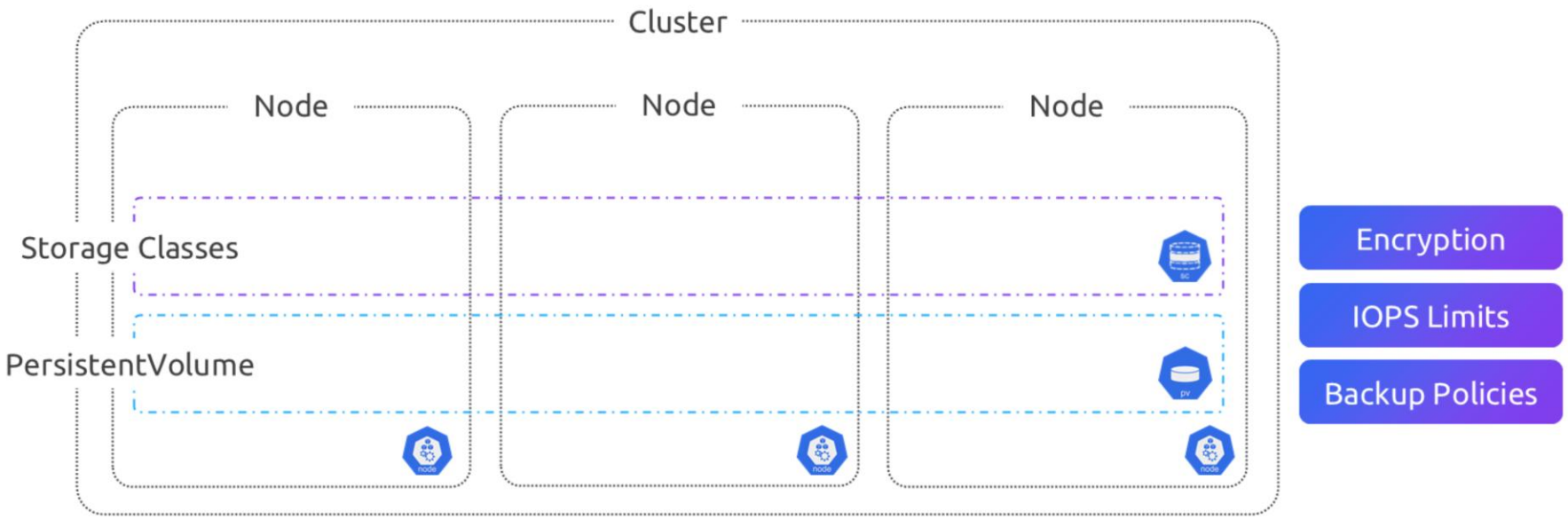
For example here we grant a user read-only access to PVCs in a namespace using get and list:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pvc-reader
rules:
-      apiGroupstype: [“”]
  resources: [“persistentvolumeclaims”]
  verbs: [“get”, “list”]
```

Then, create a RoleBinding to assign this role to a user:

By controlling access to storage resources, you can prevent unauthorized users from accessing or modifying sensitive data.

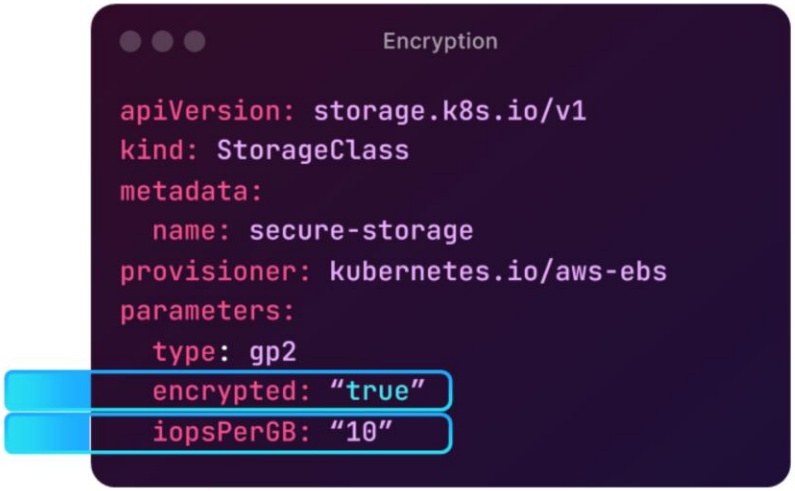
# Storage in Kubernetes






Storage Classes in Kubernetes define the types of storage available in the cluster and their parameters. By using Storage Classes, you can enforce policies such as encryption, IOPS limits, and backup policies.

# Using Storage Classes and Policies




Enforcing policies with storage classes



A Storage Class can specify parameters that enforce security measures, such as enabling encryption or setting performance limits:

For example this storage class has a set of parameters that help define certain standards. Here, IOPS represents the number of read and write operations a storage device can perform in a second. Higher IOPS means better performance for applications that require frequent data access.

Using this type of storage Classes allows you to standardize storage configurations across your cluster, ensuring that security measures are consistently applied.

# Implementing Backup and Disaster Recovery




Next let's look at taking Regular backups and a robust disaster recovery plan. These are essential to protect data against loss or corruption. You can Use tools and solutions that integrate with Kubernetes to automate backups and ensure data can be restored quickly.

Velero, PortWorx, OpenEBS, and Kasten are some of the backup solutions available.

# Implementing Backup and Disaster Recovery

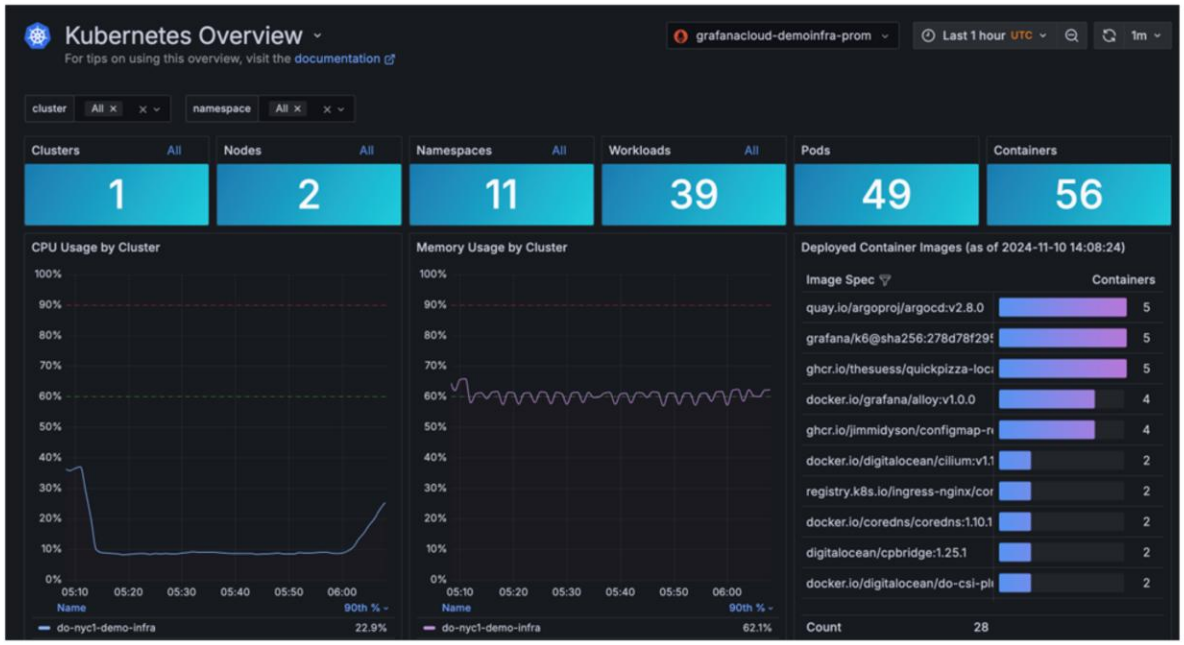

## Use

- Disaster recovery
- Cluster migration
- Enable API group versions
- Resource filtering
- Backup reference
- Backup hooks
- Restore reference
- Restore hooks
- Restore Resource Modifiers
- Run in any namespace
- CSI Support
- CSI Snapshot Data Movement
- Node-agent Concurrency
- Data Movement Backup PVC Configuration
- Data Movement Pod Resource Configuration
- Backup Repository Configuration
- Verifying Self-signed Certificates
- Changing RBAC permissions
- Behind proxy
- Repository Maintenance



Velero is an open-source tool that provides backups, restores, and disaster recovery for Kubernetes clusters. It can back up Kubernetes resources and Persistent Volumes.

# Monitoring and Auditing



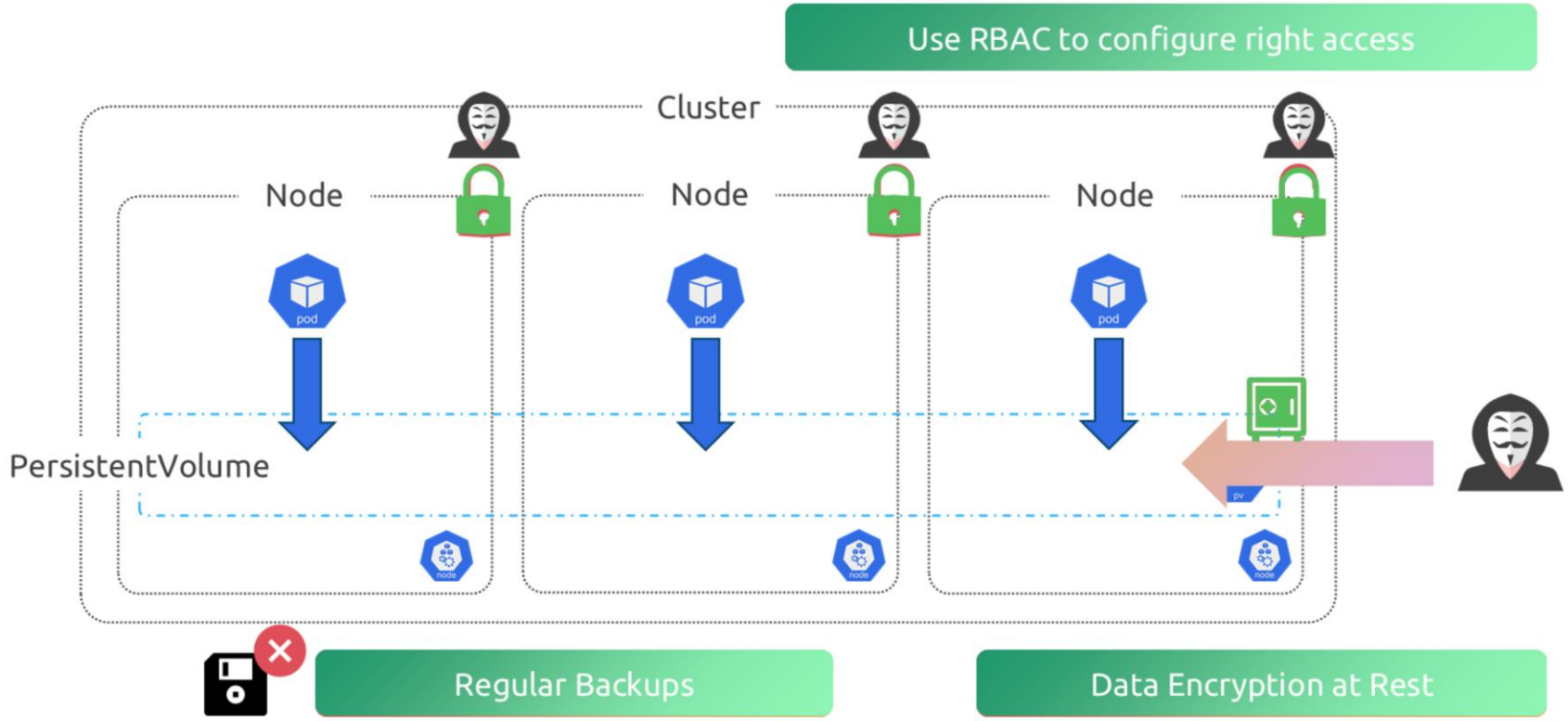

Prometheus

Grafana

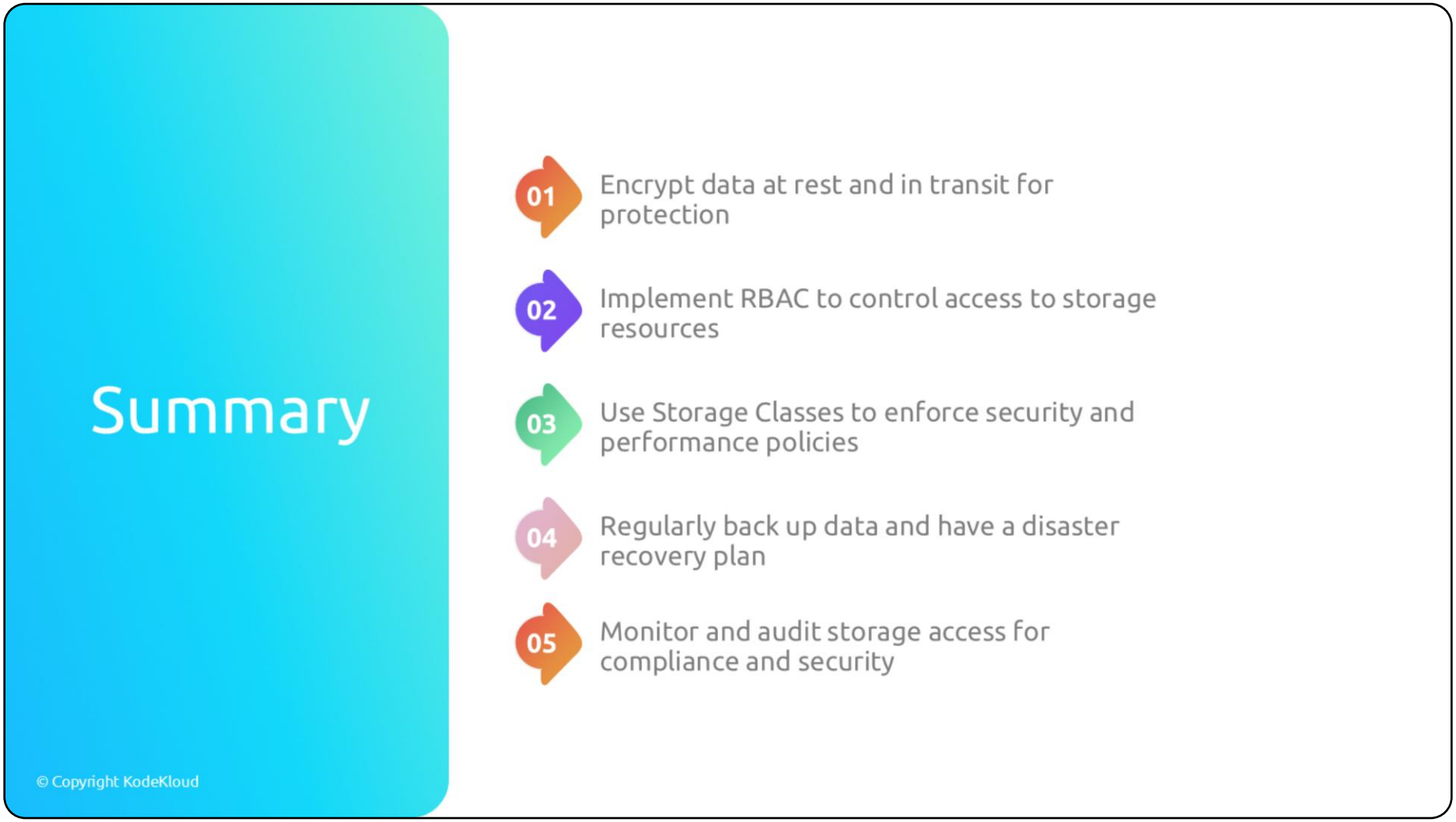


Additionally, using tools like Prometheus and Grafana allows you to monitor storage metrics and visualize access patterns, helping identify unusual activities and potential security breaches.

# Storage in Kubernetes




So to summarize use Role based access controls to secure who has what access to PVs and PVCs. Enable backups using suitable backup tools. And enable data encryption at rest using encryption objects.




and make sure to setup monitoring and audit logs for storage access for compliance and security purposes.