Kubernetes Security Training

Join our comprehensive Kubernetes security training program and discover how to protect your containerized applications against cyber threats.

Navigating Security in the Cloud Era

- The shift to cloud and multi-cloud as dominant operational models has revolutionised businesses.
- New coding cultures like Cloud Native and DevOps bring speed and agility.
- Cybersecurity is now imperative to prevent major cyber attacks and protect company reputation.
- Security is a perpetual challenge, even with the adoption of Cloud Native and DevOps.
- Goal: Guide through fundamental steps for an effective cloud security strategy.

Building a Strong Foundation

- Adoption of cloud technologies transforms operational models and cybersecurity landscape.
- Measures must be robust and regularly tested through simulations and forensic analysis.
- Lack of penetration testing in Cloud Native environments due to time, resource or budget constraints usually ends with LinkedIn recruitment posts for Security Experts (Hi Uber!).
- Cloud Native = new way of coding and delivering code as new application services.
- "You Build it, You Run it" philosophy for rapid development and innovation but …….

Promoting a Secure Culture and Structure

- Beyond non-negotiable guidelines (MITRE ATT&CK for Cloud, CIS Benchmarks), additional best practices are crucial.
- Establishing a secure "landing zone" in the cloud is essential.
- Challenges during migration require a well-thought-out Zero Trust security approach.
- Cloud Native applications heavily use containers and Kubernetes.
- Security challenges in Kubernetes require a specific approach.

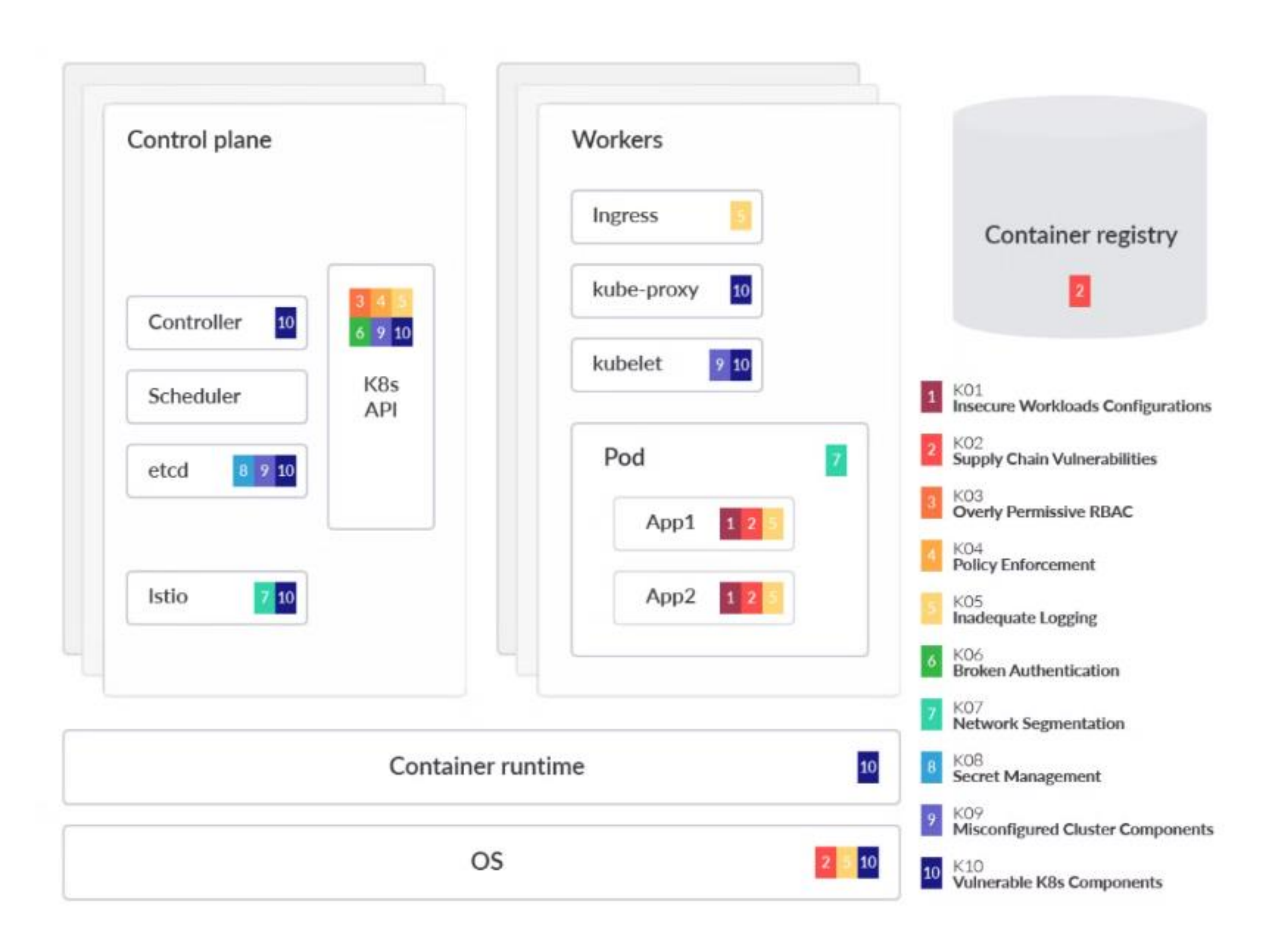
Ensuring Security Over Time

- Security is not just an instant concern but a long-term commitment.
- Factors include team training, "shift left" vision, gamification, and reducing TCO.
- Wise tool selection among CNAPP, CWPP, CSPM, automic pentesting, SAST, DAST, SCA.

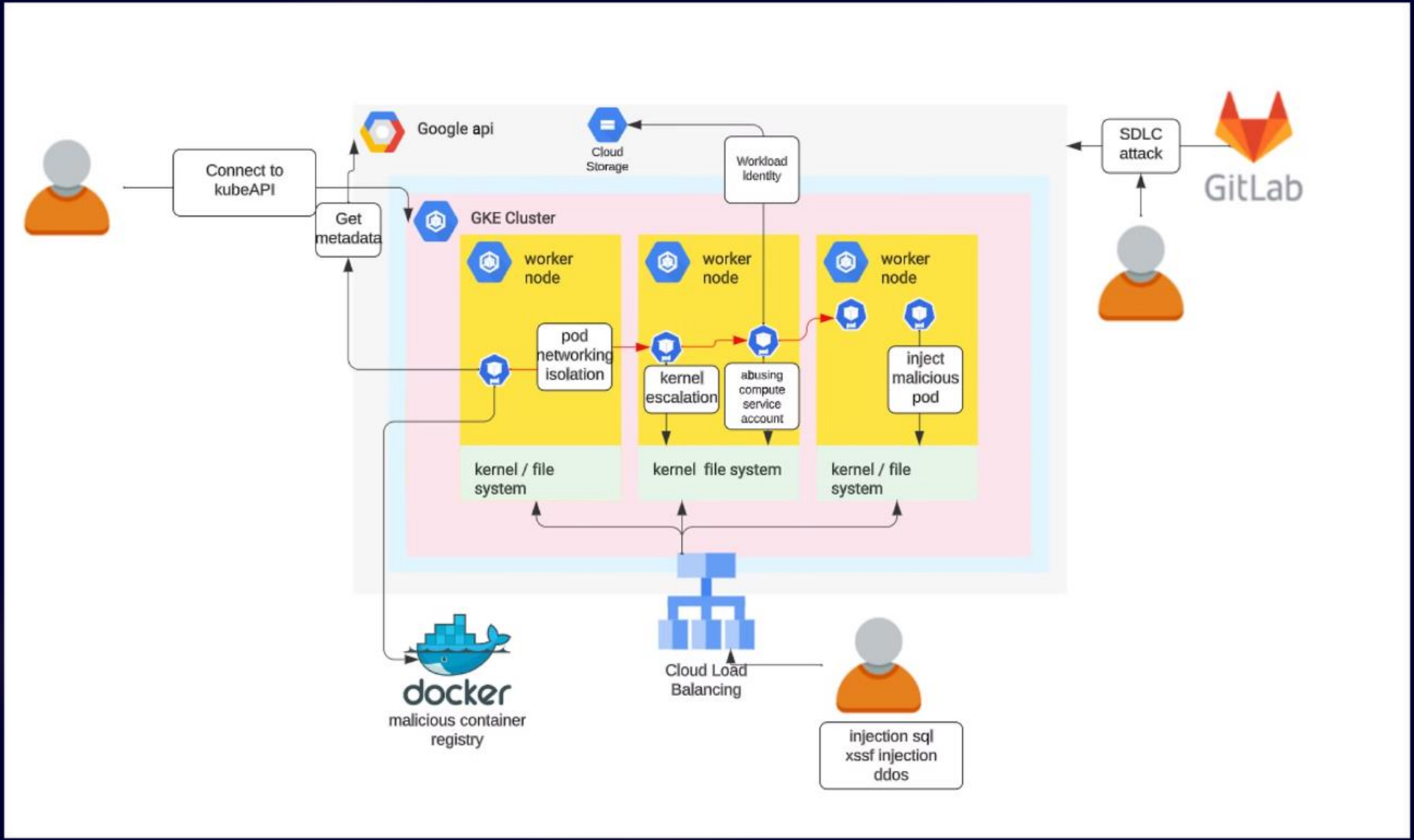
Closing Thoughts

As business decision-makers, it's crucial to have clear information on the real risks your organization faces and prioritize security measures accordingly. We also acknowledge the challenging position of CISOs and RSSIs, who play a vital role in safeguarding your company's assets. Together, let's navigate the ever-evolving landscape of cybersecurity.

OWASP Top 10 Kubernetes vulnerabilities

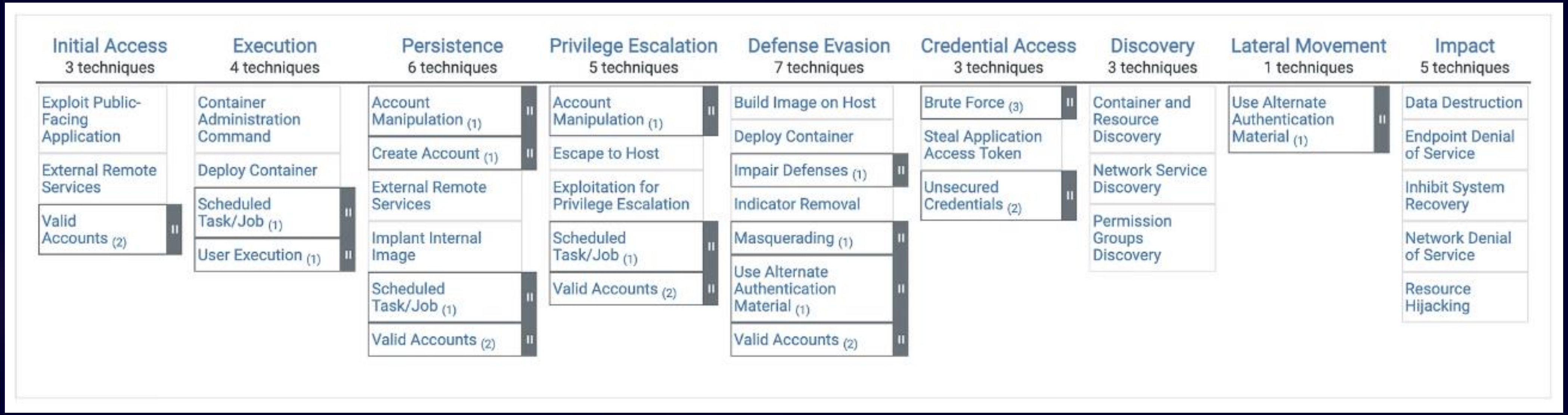


What can be done by an attacker



MITRE ATT&CK

MITRE ATT&CK, or "Adversarial Tactics, Techniques, and Common Knowledge", is a comprehensive framework mapping the tactics, techniques, and procedures (TTPs) used by cyber adversaries. Organized into matrices for various platforms, it helps cybersecurity professionals understand and counter potential threats. It's widely utilized for threat intelligence, incident response, and security operations in the cybersecurity community.



4C Security Layers

Code

Learn how to write secure code and implement security best practices at the software development stage.

Container

Understand container security and apply techniques to ensure the safety of your containerized workloads.

Cluster

Explore strategies for securing your Kubernetes cluster infrastructure and preventing unauthorized access.

Cloud

Discover how to leverage cloud-based security features, such as encryption and identity management, to enhance the overall security of your Kubernetes environment.

Workload and Container Security

1 **Container Best Practices**

Master the essential security practices for deploying and managing containers in a secure manner.

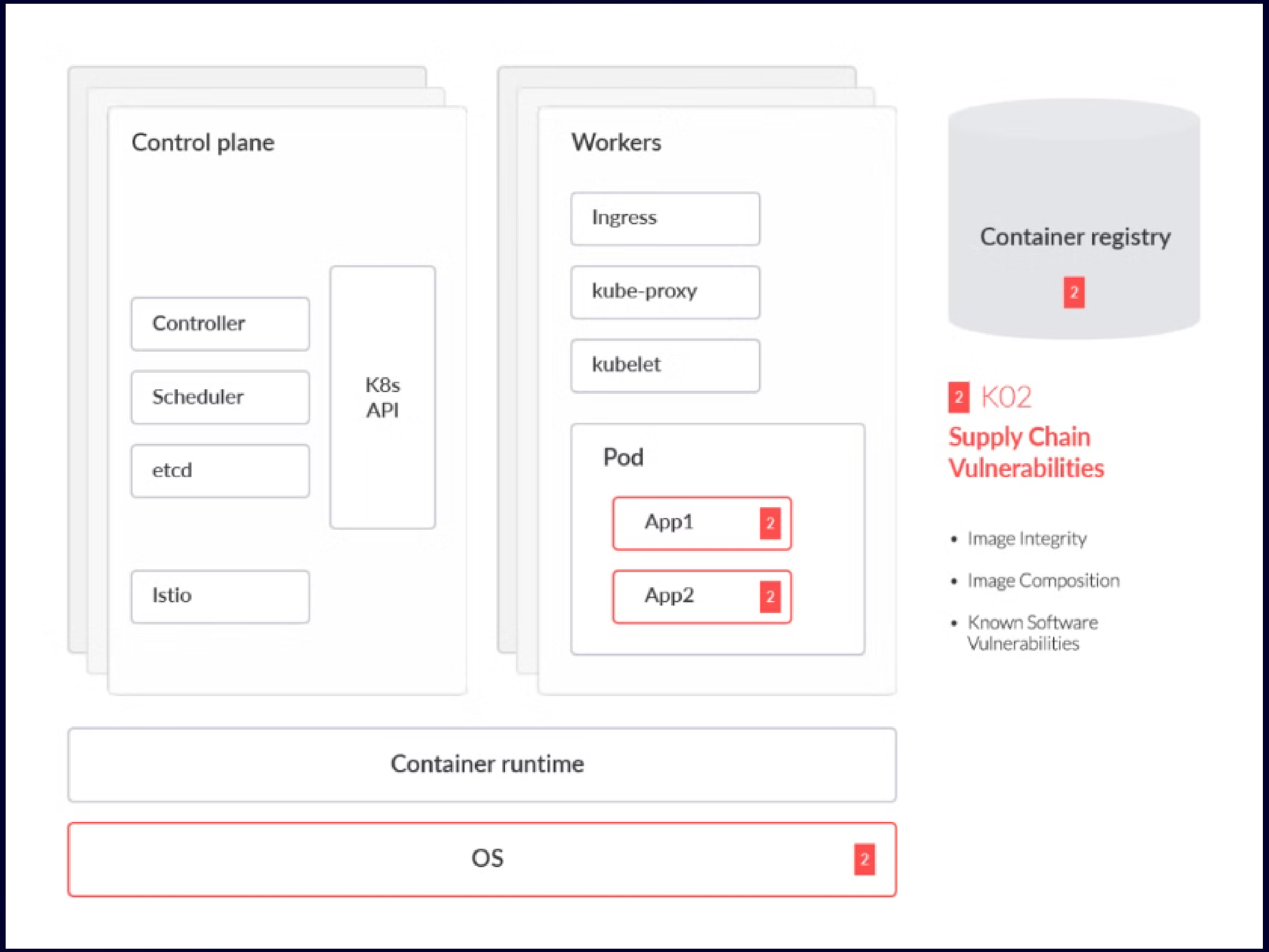
2 **Least Privilege Principle**

Learn how to apply the principle of least privilege to restrict access and limit potential attack vectors.

3 **Security Context of Pods**

Explore the security context options available for pods and understand how to configure them effectively.

Container Best Practices

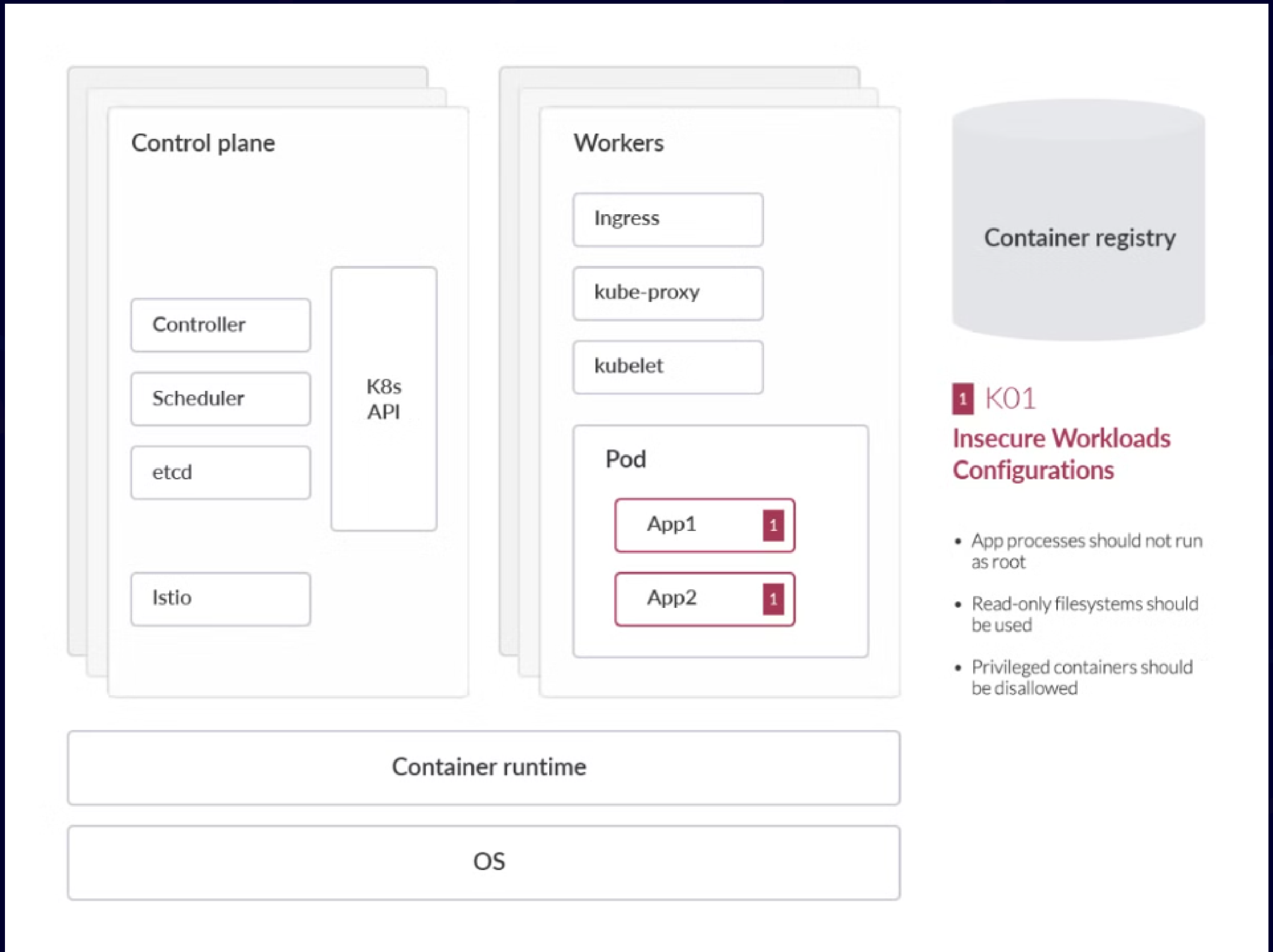


- **Start with a Minimal Base Image:**
    - Use lightweight base images
    - Minimize the attack surface by only including necessary dependencies.
- **Keep Images and Software Updated:**
    - Regularly update base images and software within containers to patch vulnerabilities.
    - Utilize automated processes to keep images up-to-date.

Least Privilege Principle

- **Implement Least Privilege:**
  - Run containers with the least necessary privileges.
  - Avoid running containers as root. Use non-root users whenever possible.
- **Use Docker Content Trust:**
  - Enable Docker Content Trust to verify the authenticity of images.
  - This ensures that only signed and verified images are used.
- **Container Image Scanning:**
  - Integrate container image scanning tools to identify vulnerabilities.
  - Regularly scan both base and application images for known security issues.
- **Secure Docker Daemon:**
  - Restrict access to the Docker daemon.
  - Limit users who can interact with the Docker daemon, and consider using TLS for secure communication.
- **Container Image Signing:**
  - Sign and verify container images using cryptographic signatures.
  - This ensures the integrity and authenticity of the images.

Security Context of Pods



- **Network Segmentation:**
  - Implement network segmentation to isolate containers.
  - Use Docker's network modes to control container communication.
- **Secrets Management:**
  - Avoid hardcoding sensitive information like API keys or passwords in Dockerfiles.
  - Use Docker's secret management or external tools like HashiCorp Vault.
- **Isolation with Namespaces:**
  - Leverage Linux namespaces for process isolation.
  - Namespaces help in creating isolated environments for containers.
- **Regular Auditing and Logging:**
  - Implement logging for containers to track and monitor activities.
  - Regularly audit logs for any suspicious behavior.
- **Secure Deployment Practices:**
  - Avoid exposing unnecessary ports in the container.
  - Use environment variables for configuration, not hardcoded values

Dangerous configuration

```yaml
spec:
  hostPID: true
  hostIPC: true
  hostNetwork: true

  securityContext:
    allowPrivilegeEscalation: true
    runAsUser:
    supplementalGroups:
      - mygroup1
      - mygroup2
    capabilities:
      add:
        - NET_RAW
        - SYS_ADMIN
```

Better configuration

```yaml
spec:
  securityContext:
    allowPrivilegeEscalation: false
    runAsUser: 1000
    runAsGroup: 1000
    supplementalGroups: []
    capabilities:
      drop:
        - ALL
    seccompProfile: runtime/secure
    seLinuxOptions:
      type: container_runtime_t
```

Demo : Inject malicious pod

Closing Thoughts

To foster a secure containerized environment, it is essential to **educate** development teams. This involves offering comprehensive **security training** to teams engaged in container development. Emphasize the **adoption of secure coding practices** within their workflows and promote heightened awareness regarding container-specific security issues. By instilling a **culture of security** consciousness among development teams, organizations can significantly **enhance the overall robustness and resilience** of their containerized applications.

LAB 1 Sign and validate container

Networking in Kubernetes

1. ## Micro Network Segmentation

Learn the techniques for implementing micro network segmentation to isolate workloads and reduce the impact of potential breaches.

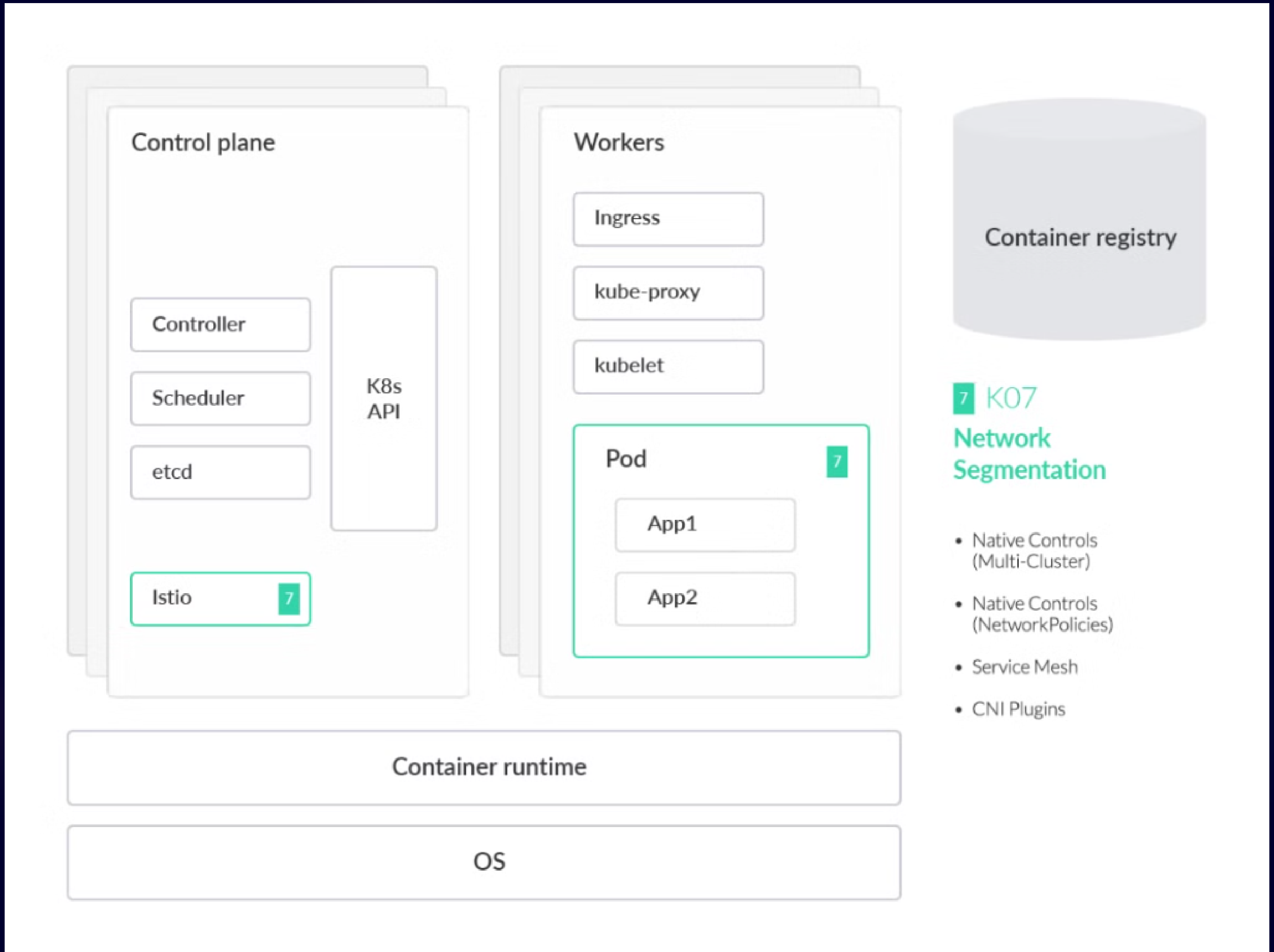
2. ## Network Policies

Discover how to enforce granular network policies to control traffic flow within your Kubernetes cluster.

3. ## Application Ingresses

Understand how to configure and secure application ingress controllers to control access to your services.

Network segmentation



First...

To comprehend the significance of discussing networking in Kubernetes, it's crucial to keep in mind that :

- In a default Kubernetes cluster, all pods can communicate with each other, even across different namespaces, and can also exit the cluster.

Therefore, it is crucial to isolate pods based on application logic and only allow necessary flows for the proper functioning of the application.

Micro Network Segmentation

Here's how micro-segmentation works in Kubernetes:

- **Default Communication:** This unrestricted communication might not align with the security requirements of certain applications.
- **Network Policies:** Micro-segmentation involves implementing network policies that explicitly define communication rules between pods. These policies are akin to firewall rules for your Kubernetes cluster.
- **Implementation:** Micro-segmentation is implemented using tools like Kubernetes Network Policies. These policies are applied to specific namespaces, controlling the traffic within that namespace. Additionally, cloud providers or third-party solutions may offer extended capabilities for network security.
- **Service Mesh Integration:** In more complex scenarios, micro-segmentation can be complemented by integrating with service mesh solutions like Istio. Service meshes provide additional features such as traffic monitoring, load balancing, and advanced routing.

Network Policies

Network Policies in Kubernetes serve as a **crucial mechanism for controlling the communication** between Pods within a cluster. These policies act as a **set of rules** that define how Pods can communicate with each other, both at the **intra- and inter-namespace** levels. The primary goal is to **enhance security** by **restricting the flow of traffic**, allowing only authorized connections based on specified criteria.

How it works

Here's a simplified breakdown of how Network Policies work:

- **Pod Selection:**
  - Policies begin by selecting a group of Pods based on labels, namespaces, or other identifying attributes.
- **Rule Definition:**
  - Once Pods are selected, administrators define rules that dictate how these Pods can send or receive network traffic.
- **Egress and Ingress Rules:**
  - Policies often include both egress (outgoing) and ingress (incoming) rules. Egress rules control what traffic Pods are allowed to send, while ingress rules define what traffic Pods are permitted to receive.
- **Protocol and Port Specifications:**
  - Rules can specify the protocols (TCP, UDP) and port numbers for the allowed traffic. This granularity ensures that only necessary and safe communications are permitted.
- **Default Deny:**
  - By default, Kubernetes follows a principle of "deny by default." This means that if a Pod doesn't match any of the defined rules, its communication is automatically restricted. This approach enhances security by minimizing unintended connections.
- **Rule Enforcement:**
  - The Kubernetes network policy controller enforces these rules by working with the underlying network plugin (e.g., Calico, Cilium). The network plugin then configures the network layer to adhere to the specified policies.

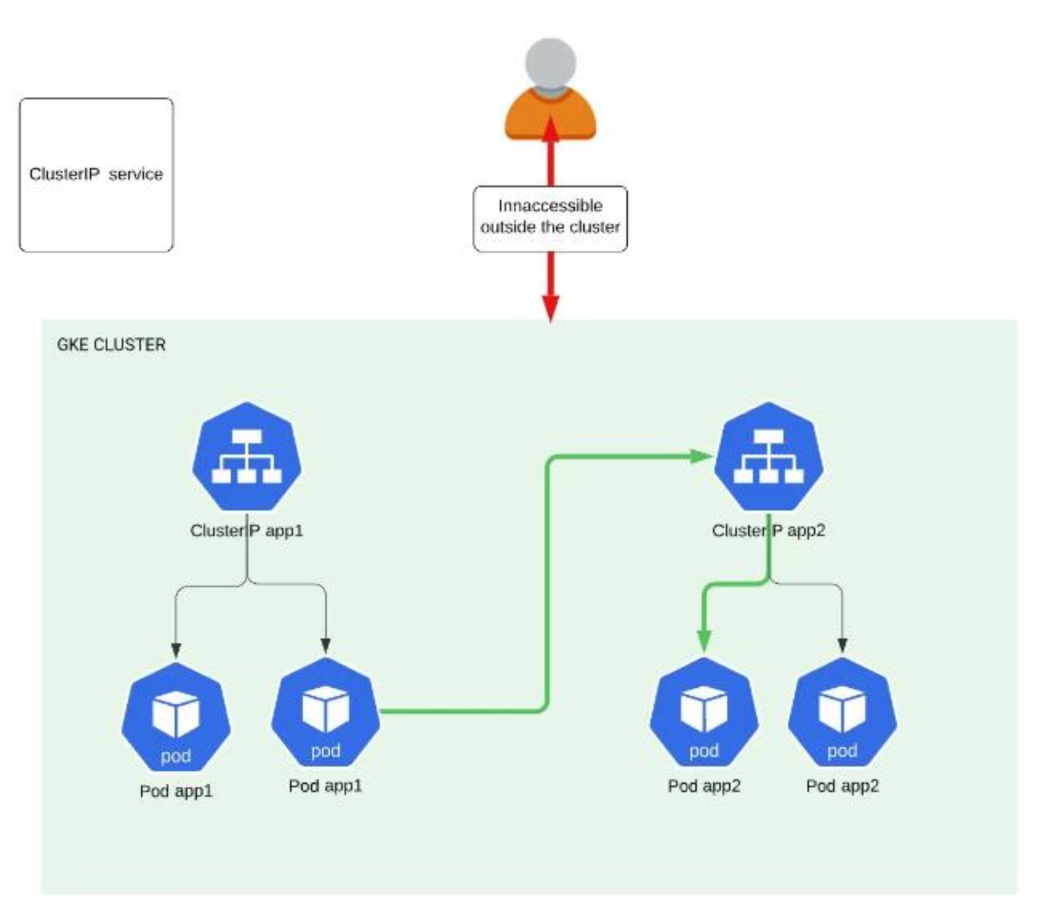
```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: example-network-policy
  namespace: mynamespace
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              project: myproject
          podSelector:
            matchLabels:
              role: frontend
      ports:
        - protocol: TCP
          port: 3306
  egress:
    - to:
        - namespaceSelector:
            matchLabels:
              project: backupproject
          podSelector:
            matchLabels:
```

LAB network policies

Back to basics

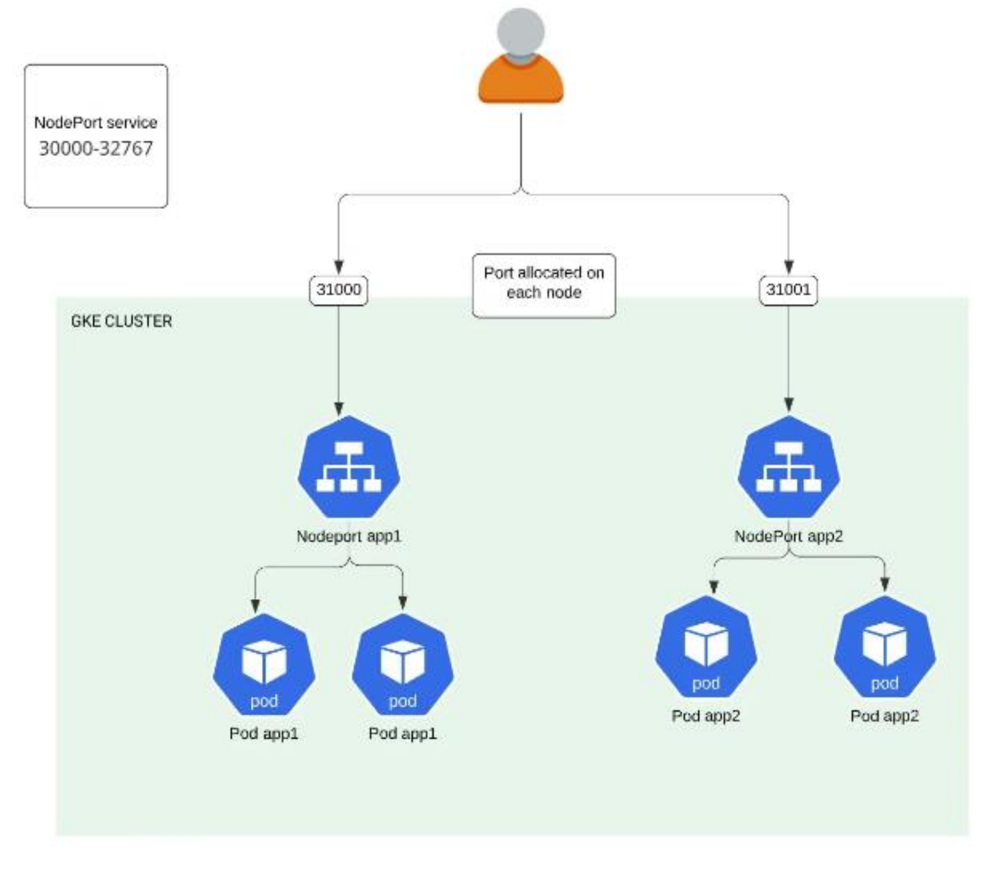
ClusterIP Service

- **Purpose:** Enable inter-service communication between pods with internal load balancing.
- **Functionality:** Restricts direct exposure outside the Kubernetes cluster.



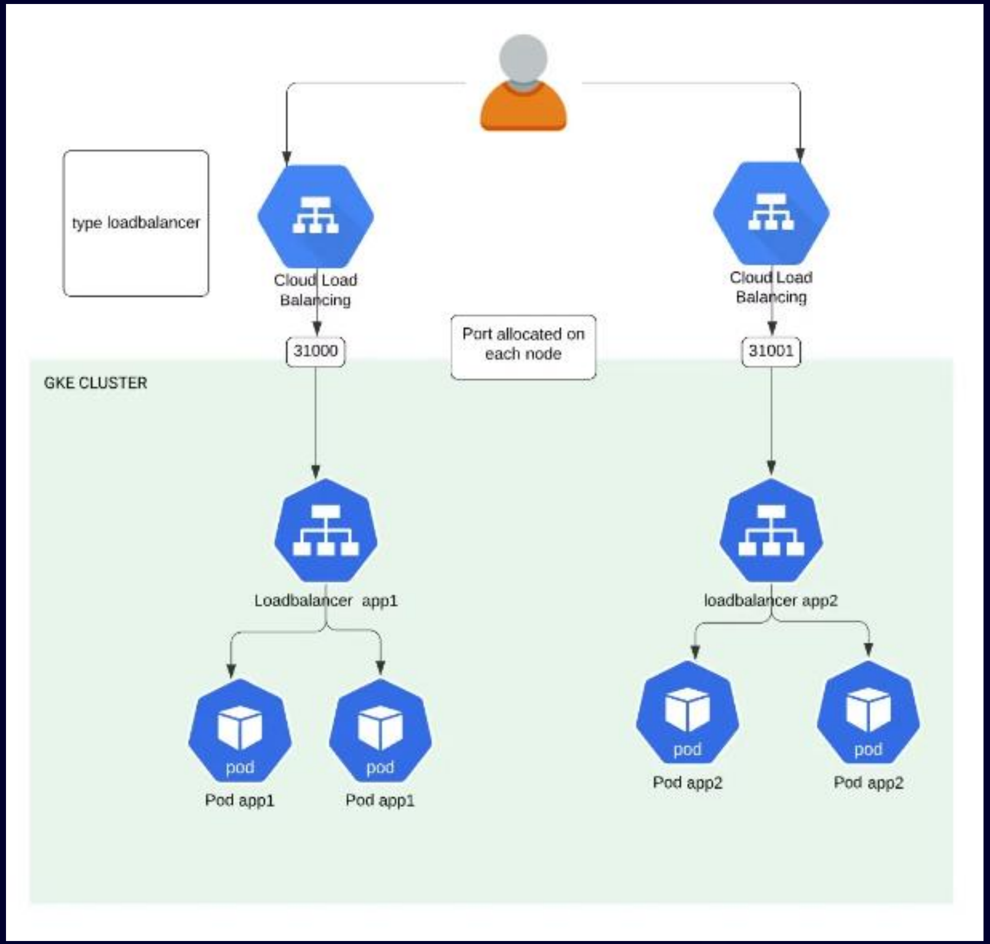
NodePort Service

- **Purpose**: Expose a service outside the cluster.
- **Mechanism**: Kubernetes allocates a port on each worker node (30000–32767) to route traffic to pods.
- **Considerations**:
  - Increased attack surface.
  - Lack of centralized control.
  - Limited Layer 7 features.
  - Reduced observability.



LoadBalancer Service

- **Purpose:** Utilizes NodePort service as a backend.
- **Feature:** Creates a native cloud provider load balancer (e.g., Google Cloud HTTP/S Load Balancer).
- **Effect:**
    - Complicates architecture.
    - Multiplies load balancers.
    - Challenges in traffic control.
    - Limited support for application updates and advanced traffic management features.



Made with Gamma

Application Ingresses

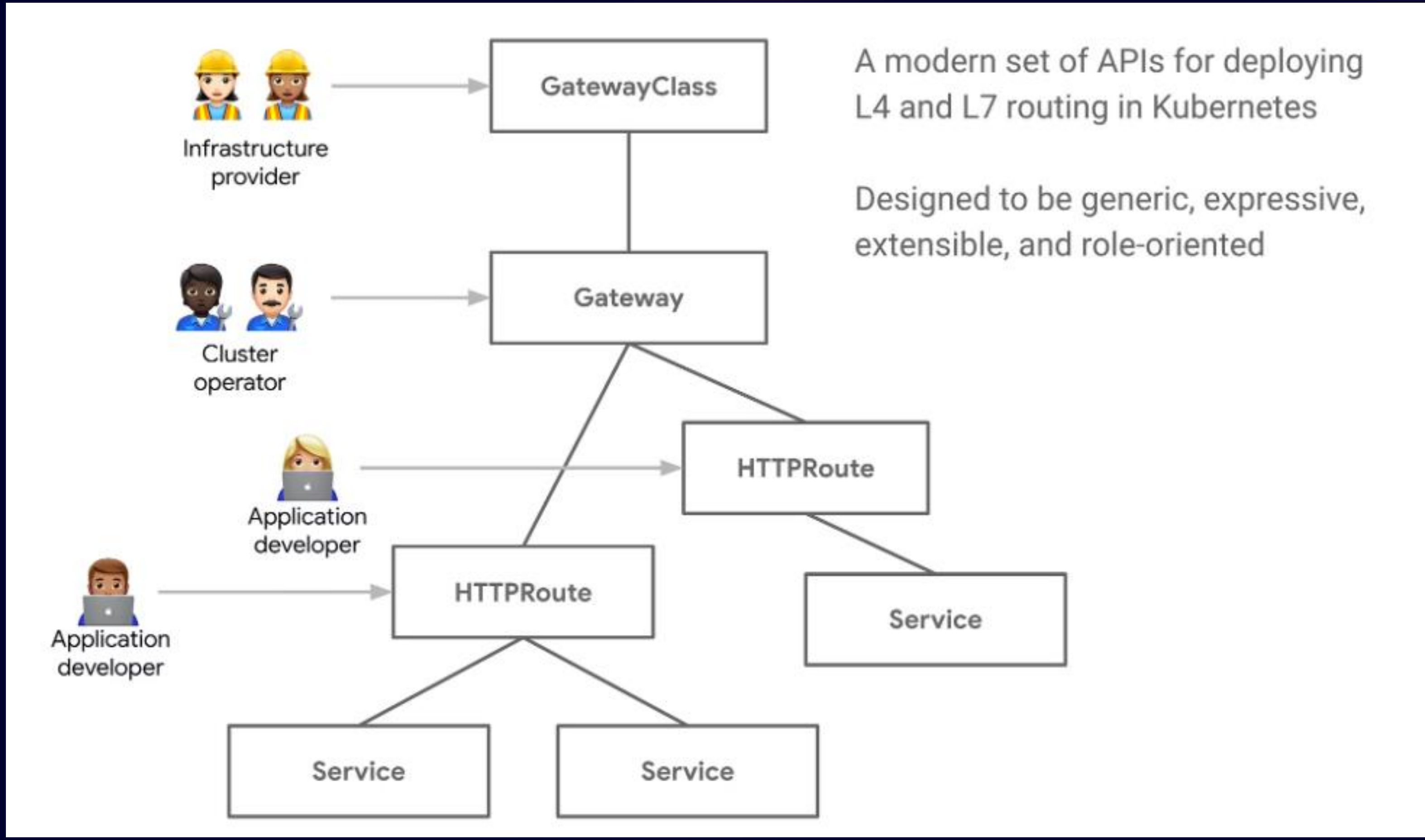
In Kubernetes, Application **Ingresses** serve as a vital mechanism for **managing external access** to services. By defining routing rules, Ingress allows centralized control over how external HTTP/S traffic is directed to specific services within the cluster. This abstraction streamlines connectivity management, offering features like SSL termination, load balancing, and path-based routing. It plays a crucial role in enhancing flexibility and security in microservices architectures.

One challenge lies in the fact that the Ingress object is a **single resource**, combining **application logic** (routes, services, etc.) with **load balancer configuration** (HTTP redirect, infrastructure annotations, protocol). Typically, these aspects should be **handled by different teams** (infrastructure DevOps, security) and product or development teams. This setup makes it **challenging to maintain** a clear separation of responsibilities. Also, the native Ingress object in Kubernetes provides **limited advanced features** such as telemetry and advanced traffic management.

Gateway API

To address the limitations mentioned above, the Kubernetes community has been actively addressing the limitations of the INGRESS object, leading to the development of Kubernetes Gateway API.

- Multitenancy
- Responsibility Separation (Cluster Operator, Dev/Product Team, SRE)
- Increased Flexibility
- Advanced Traffic Management



- **GatewayClass (SRE) :** The "GatewayClass" kind represents a list of available load balancers that will be referenced in your "Gateway" object based on your requirements.
- **Gateway (SRE) :** This resource allows you to configure the load balancer, listening protocols, ports, and more. Example : HTTPS External Gateway with TLS Certificate.

```yaml
kind: Gateway
apiVersion: gateway.networking.k8s.io/v1beta1
metadata:
  name: external-http
  annotations:
    networking.gke.io/certmap: store-example-com-map
spec:
  gatewayClassName: gke-l7-gxlb
  listeners:
  - name: https
    protocol: HTTPS
    port: 443
```

- **Policy (SRE) :** Authorizations are determined using role-based access control (RBAC). Due to the hierarchical nature of the Gateway API, it's possible to associate a Policy resource with a higher-level resource (Gateway) in a namespace so that all resources below this resource, even in different namespaces, inherit the characteristics of this rule.

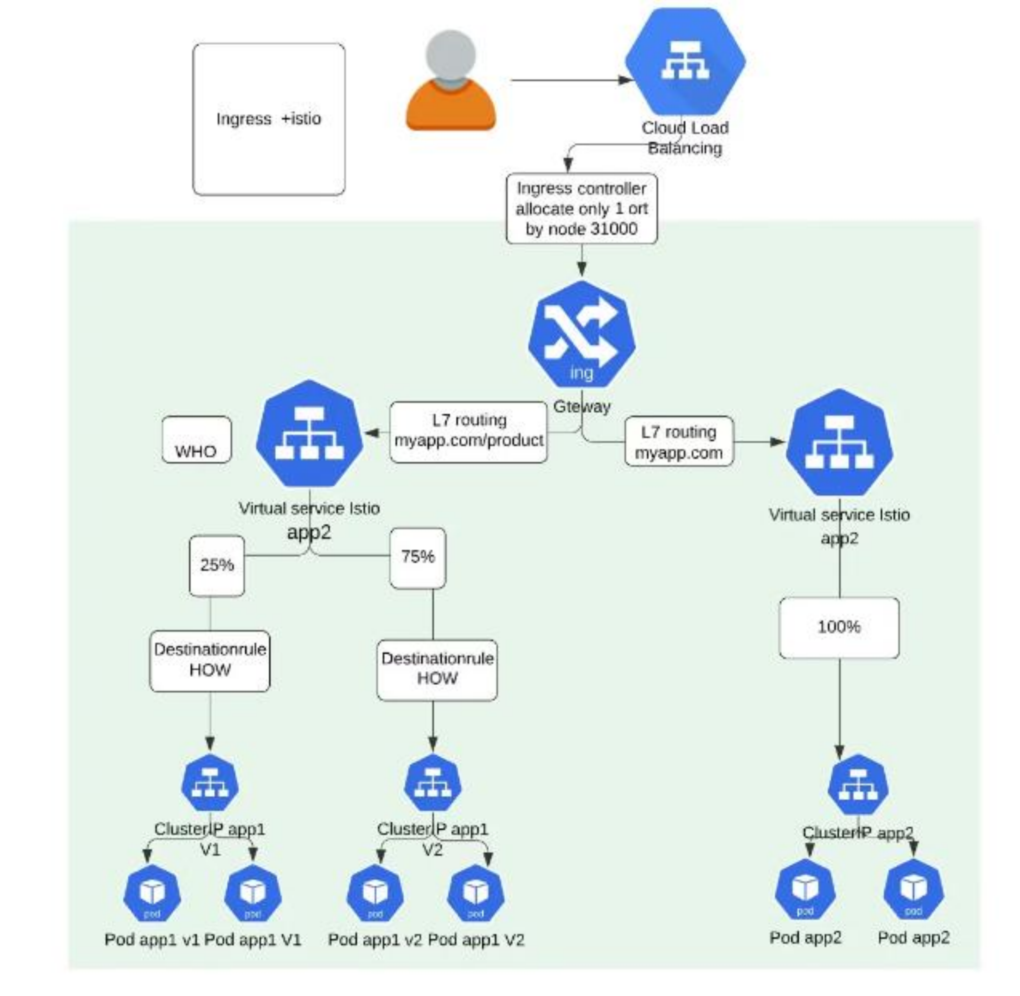
- **HTTP Route (Dev) :** Routes pod traffic to the appropriate services based on various policies, load balancing, weights, and application logic.

Object	OSI Layer	Routing Discriminator	TLS Support	Purpose
HTTPRoute	Layer 7	Anything in the HTTP Protocol	Terminated only	HTTP and HTTPS Routing
TLSRoute	Somewhere between layer 4 and 7	SNI or other TLS properties	Passthrough or Terminated	Routing of TLS protocols including HTTPS where inspection of the HTTP stream is not required.
TCPRoute	Layer 4	destination port	Passthrough or Terminated	Allows for forwarding of a TCP stream from the Listener to the Backends
UDPRoute	Layer 4	destination port	None	Allows for forwarding of a UDP stream from the Listener to the Backends.
GRPCRoute	Layer 7	Anything in the gRPC Protocol	Terminated only	gRPC Routing over HTTP/2 and HTTP/2 cleartext

Advanced Ingress Controllers

Other solution would be relying on tools like **Istio, Consul or LinkerD** that have emerged to **enhance flexibility**, improve **granularity**, and offer **more abstraction** through "VirtualService". VirtualService functions as a service for services, allowing **more granular traffic management**, including A/B testing and canary releases, ultimately providing better application SLAs.

For instance, **Istio** and **LinkerD** introduce their custom resource definitions (CRDs) for Gateways, **bridging the gaps** in Kubernetes and **delivering advanced traffic** management and various other features.



Cluster Management

Cluster management involves various aspects to ensure the efficient and secure operation of Kubernetes clusters.

1. ### Quotas

   Learn how to set resource quotas to prevent resource exhaustion and ensure fair allocation.

2. ### Resource Management

   Discover techniques for effectively managing and optimizing resource allocation within your Kubernetes cluster.

3. ### User Identity and RBAC

   Explore user authentication, authorization, and role-based access control (RBAC) mechanisms to secure cluster access.

4. ### Multi-cluster Considerations

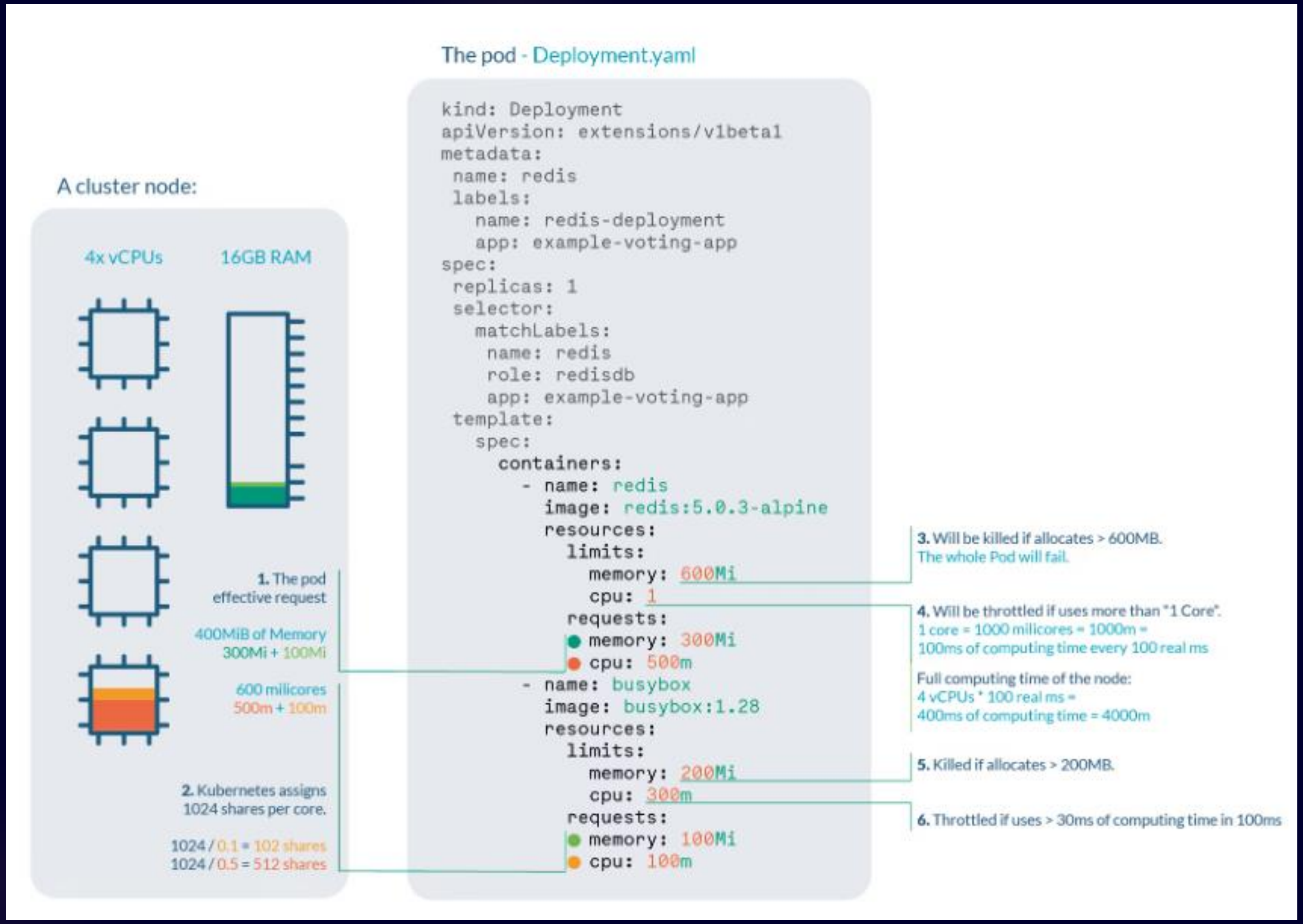
   Learn how to apply security practices across multiple Kubernetes clusters, ensuring consistent protection for your workloads.

Quotas

Quotas in Kubernetes are a mechanism used to control and limit the amount of resources that can be consumed by objects within a namespace. This is essential for maintaining fair usage, preventing resource exhaustion, and ensuring the overall stability and performance of a Kubernetes cluster.

Here are key points about quotas in Kubernetes:

1. **Resource Limitations**
2. **Namespace-Level Control**
3. **Preventing Resource Exhaustion**
4. **Fair Resource Distribution**
5. **Monitoring and Enforcement**


The pod - Deployment.yaml
A cluster node:
4x vCPUs
16GB RAM
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
name: redis
labels:
name: redis-deployment
app: example-voting-app
spec:
replicas: 1
selector:
matchLabels:
name: redis
role: redisdb
app: example-voting-app
template:
spec:
containers:
- name: redis
image: redis:5.0.3-alpine
resources:
limits:
memory: 600Mi
cpu: 1
requests:
memory: 300Mi
cpu: 500m
- name: busybox
image: busybox:1.28
resources:
limits:
memory: 200Mi
cpu: 300m
requests:
memory: 100Mi
cpu: 100m
1. The pod effective request
400MiB of Memory
300Mi + 100Mi
600 milicores
500m + 100m
2. Kubernetes assigns 1024 shares per core.
1024 / 0.1 = 102 shares
1024 / 0.5 = 512 shares
3. Will be killed if allocates > 600MB.
The whole Pod will fail.
4. Will be throttled if uses more than "1 Core".
1 core = 1000 milicores = 1000m =
100ms of computing time every 100 real ms
Full computing time of the node:
4 vCPUs * 100 real ms =
400ms of computing time = 4000m
5. Killed if allocates > 200MB.
6. Throttled if uses > 30ms of computing time in 100ms

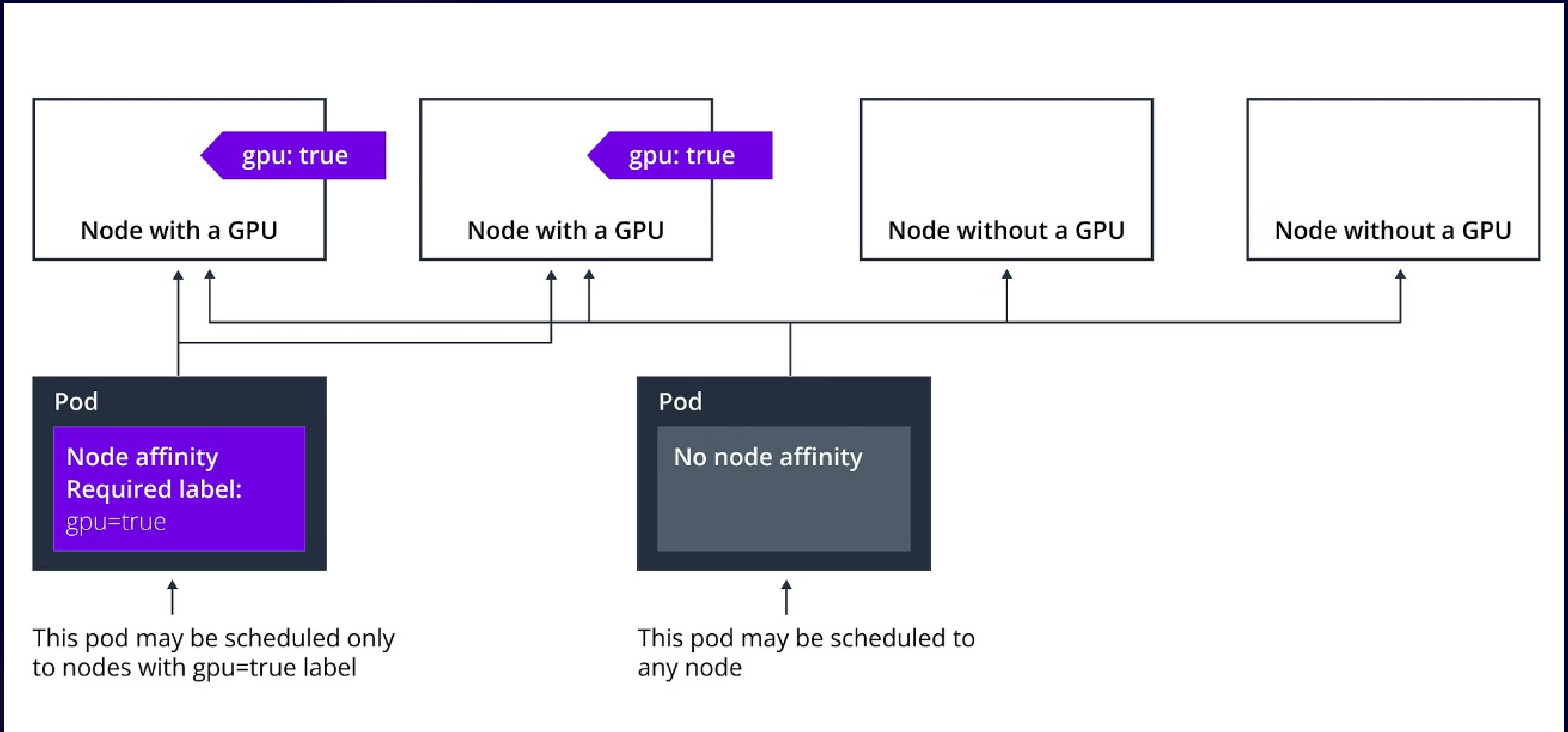
Resource Management

Resource management in Kubernetes involves the allocation, monitoring, affinity and control of computing resources within a cluster to ensure efficient and reliable operation. Here are key aspects of resource management in Kubernetes:

- **Resource Requests and Limits:**
  - **Requests:** Containers can specify the amount of resources they expect to use (requests). Kubernetes scheduler uses these requests for making decisions about where to place Pods.
  - **Limits:** Containers can have upper limits on the amount of resources they are allowed to consume. This prevents a single misbehaving container from affecting others in the same node.
- **Pod-Level Resource Management:**
  - Resources are allocated at the Pod level. All containers within a Pod share the same set of resources, and the Pod itself can be scheduled on a node based on the resource requirements.

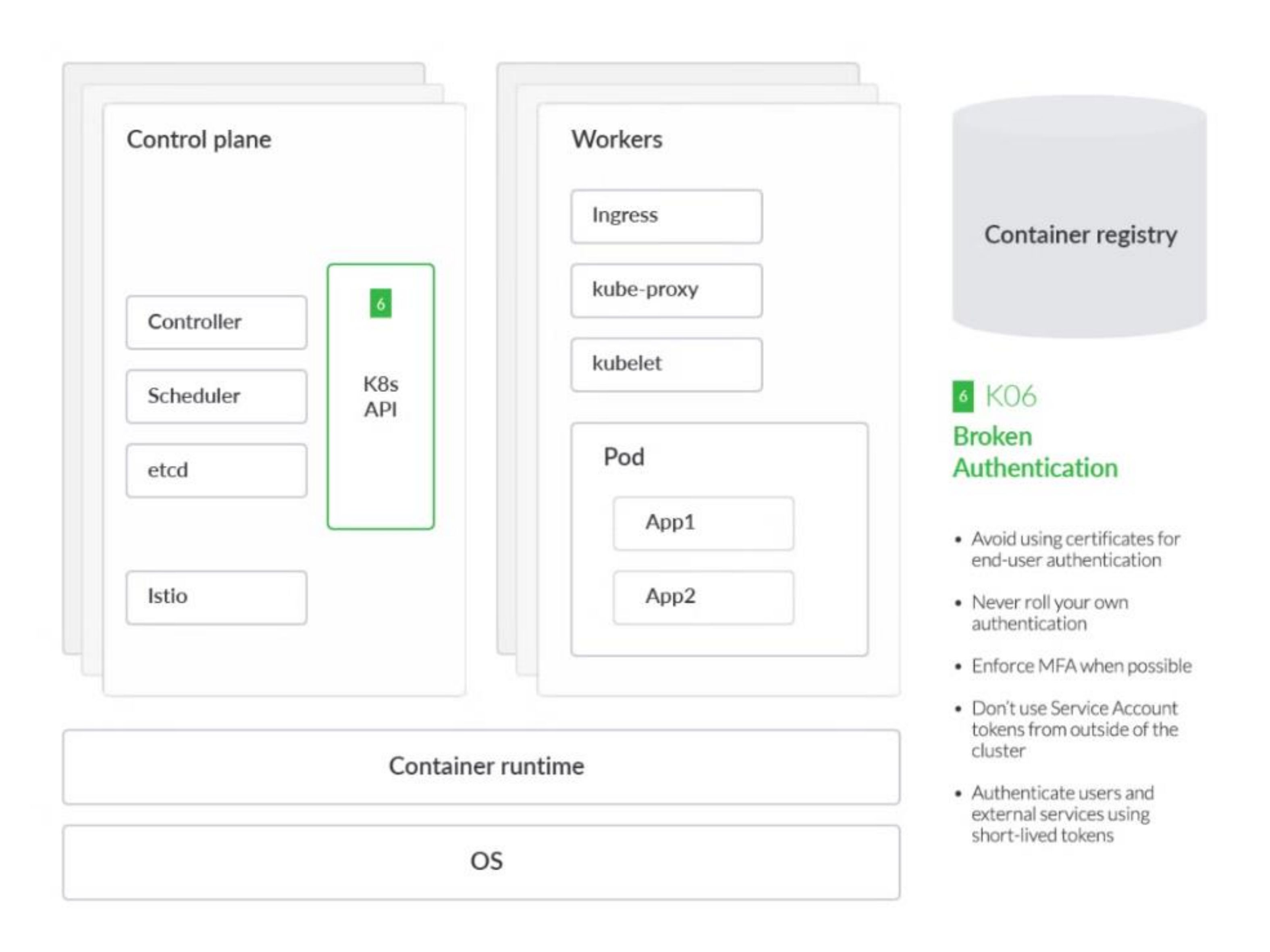
- **Quality of Service (QoS):**
  - Kubernetes categorizes Pods into three Quality of Service classes based on their resource requirements and usage: BestEffort, Burstable, and Guaranteed. This classification influences the scheduler's decisions.
- **Horizontal Pod Autoscaling (HPA):**
  - Kubernetes supports autoscaling based on observed CPU utilization or other custom metrics. With HPA, the number of replicas of a deployment or replica set can automatically scale up or down to meet the desired resource utilization.
- **Node-Level Resource Management:**
  - Kubernetes also manages resources at the node level. Node allocatable resources are considered when scheduling Pods. Nodes can be cordoned or drained for maintenance to ensure existing workloads are gracefully moved to other nodes.

- **Node Affinity & Anti-Affinity:**
  - **These rules** allow you to constrain which nodes your Pod is eligible to be scheduled (or avoid them) based on node labels.
- **Pod Affinity & Anti-Affinity :**
  - **These rules** define conditions under which a Pod should be scheduled (or not scheduled) on the same node as other Pods.
- **Topology Constraints:**
  - Topology constraints help define the specific conditions under which affinity or anti-affinity should be applied.
- **Monitoring and Metrics:**
  - Kubernetes provides metrics for resource usage, allowing administrators to monitor the performance of nodes and Pods. Tools like Prometheus are commonly used for collecting and visualizing these metrics.

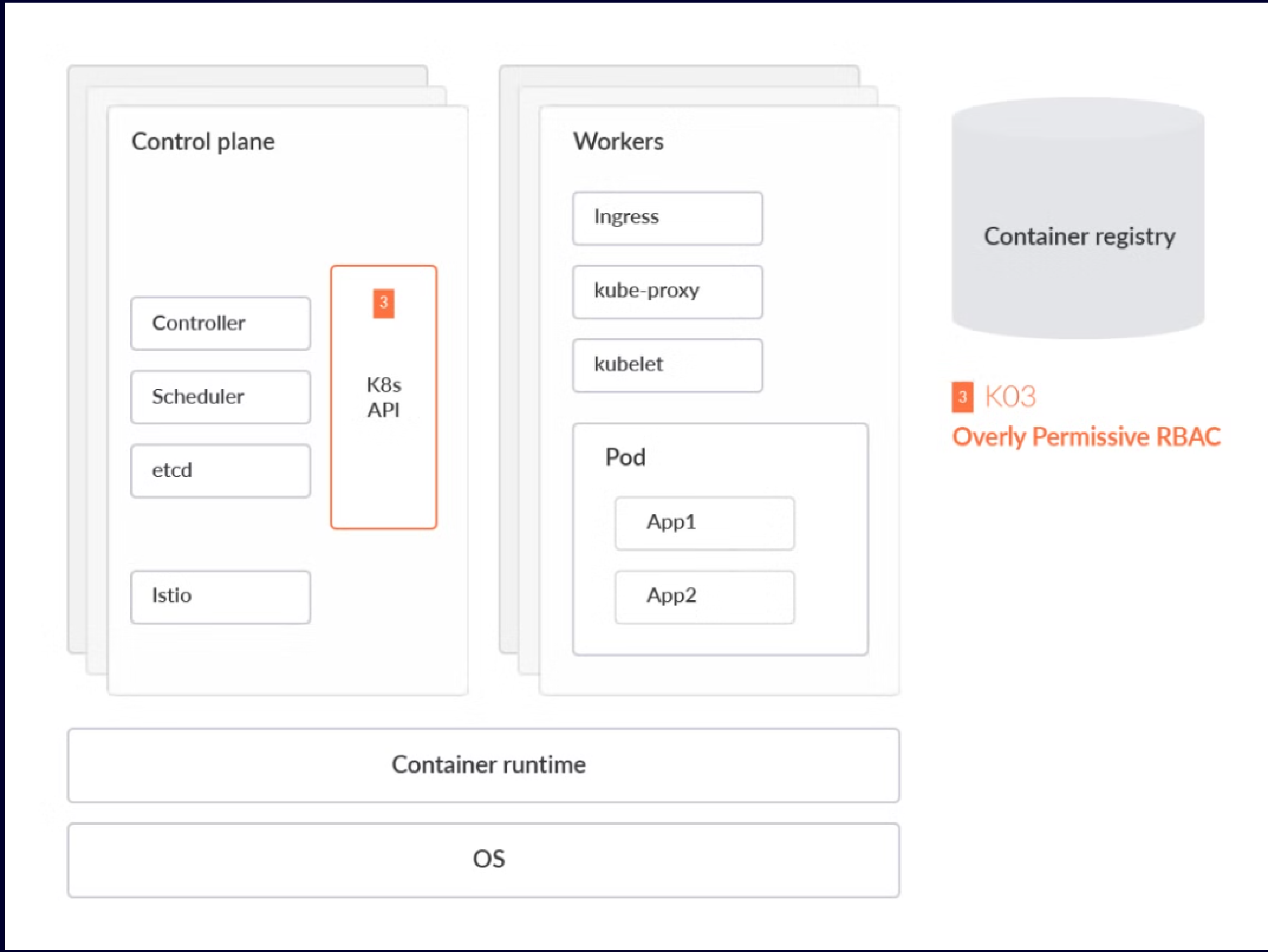

gpu: true
Node with a GPU
gpu: true
Node with a GPU
Node without a GPU
Node without a GPU
Pod
Node affinity
Required label:
gpu=true
Pod
No node affinity
This pod may be scheduled only to nodes with gpu=true label
This pod may be scheduled to any node

User Identity & RBAC

K06 : Broken Authentification



K03 : Overly permissive RBAC

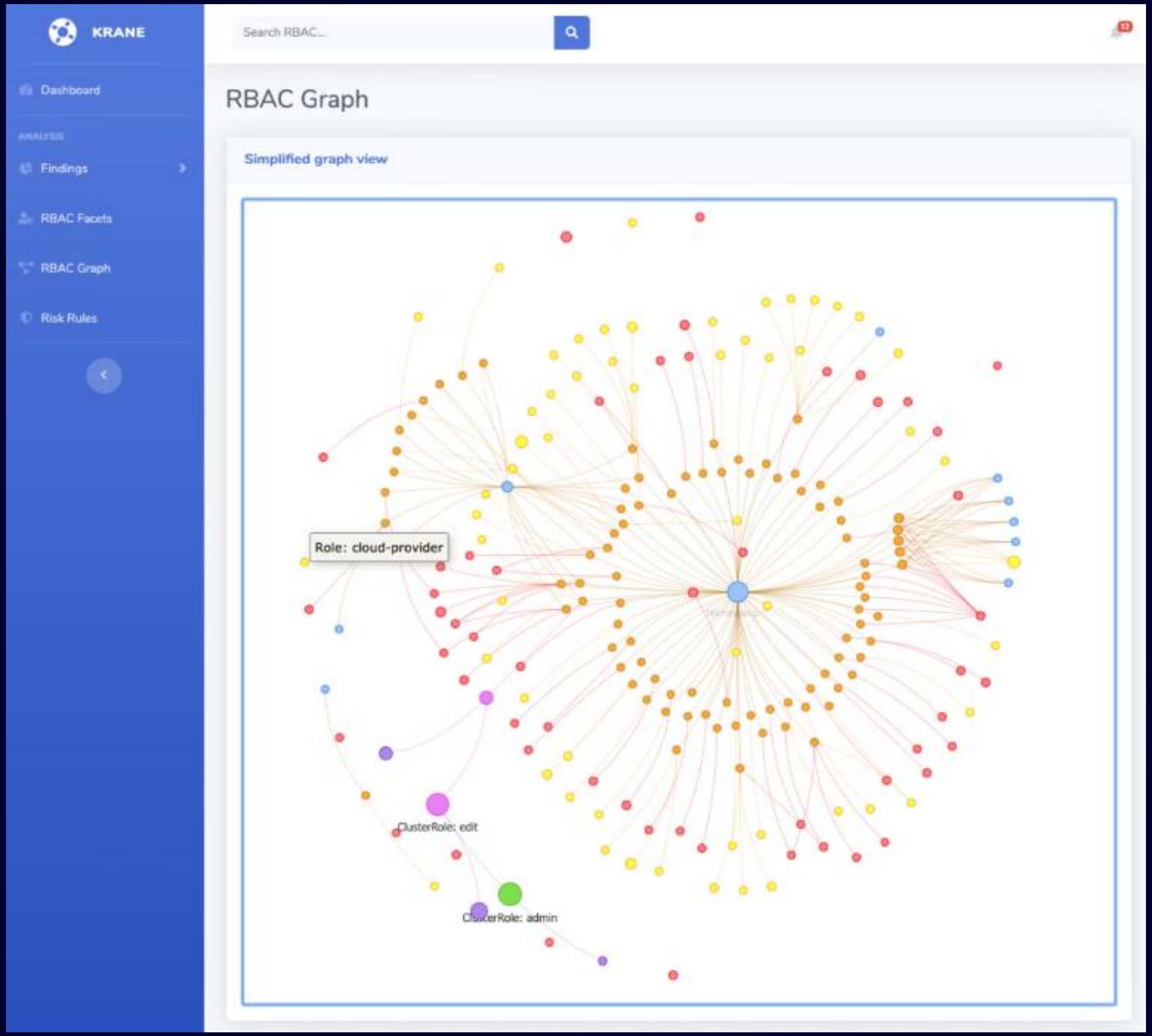


Role-Based Access Control (RBAC) in Kubernetes is a mechanism that defines and manages permissions for users or processes within a Kubernetes cluster.

- **Roles and ClusterRoles:**
  - **Role:** Defines a set of permissions within a specific namespace.
  - **ClusterRole:** Similar to Role but applies cluster-wide. It defines permissions across all namespaces.
- **RoleBindings and ClusterRoleBindings:**
  - **RoleBinding:** Binds a Role to a user, group, or service account within a specific namespace.
  - **ClusterRoleBinding:** Binds a ClusterRole to a user, group, or service account across the entire cluster.

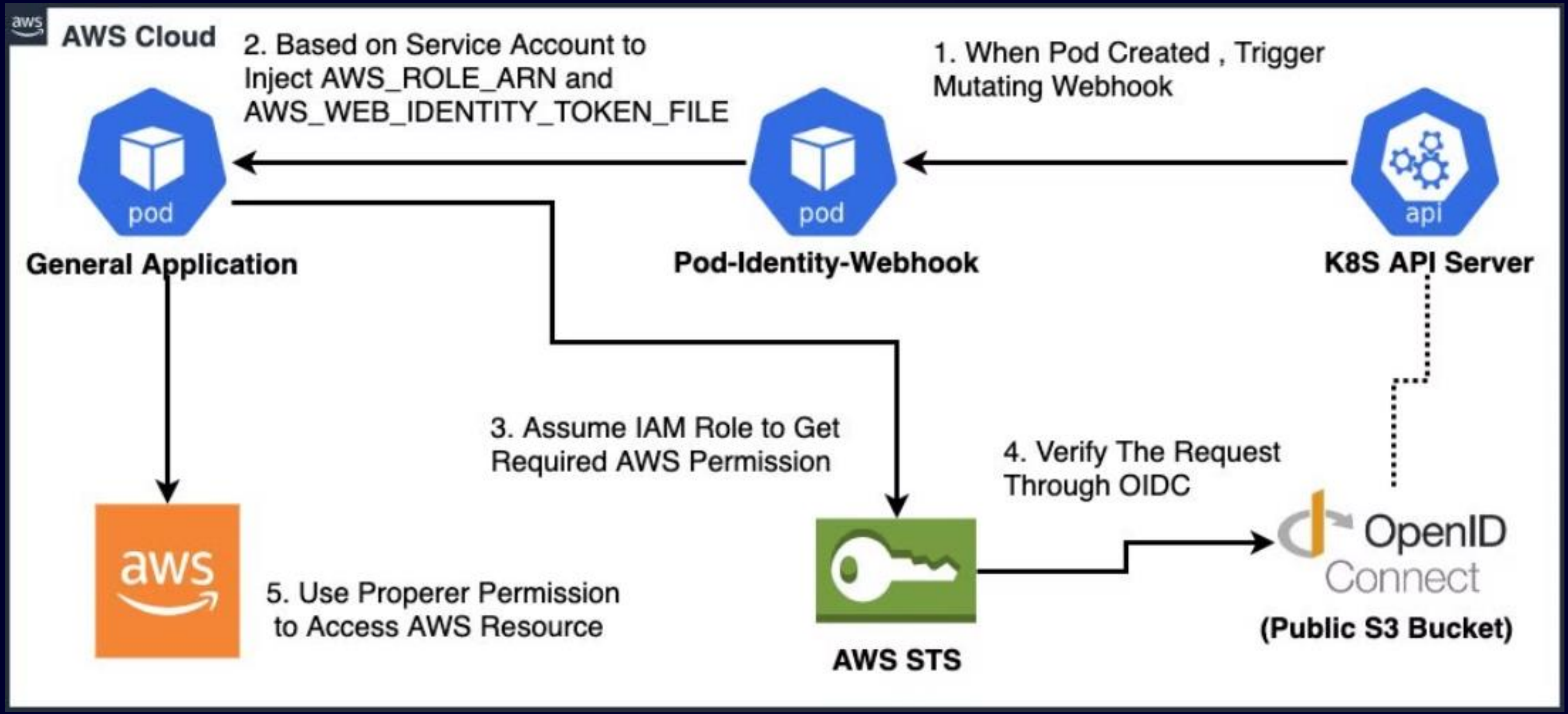
- **Service Accounts:**
    - A service account is an identity that a Pod can use to interact with the Kubernetes API server.
    - Each Pod in Kubernetes is associated with a service account, and it is assigned to the Pod when it is created.
- **Service Account Tokens:**
    - When a Pod is assigned a service account, it is automatically mounted with a service account token.
    - This token authenticates the Pod to the API server, allowing it to make requests and perform actions within the cluster.
- **RBAC and Service Accounts:**
- RBAC uses service accounts to determine the permissions a Pod has within the cluster.
- When creating roles and role bindings, you can specify service accounts as subjects to which the roles are bound.
- This allows fine-grained control over which Pods have access to specific resources and actions.

Use Kubiscan , Krane or RBAC Visualizer to detect over permissive RBAC



Kubernetes and IAM Cloud Integration:

- IAM roles can be used to provide permissions to Kubernetes service accounts within the cluster to cloud services or resources.
- Nodes in a cluster are associated with IAM roles, allowing them to interact with other Cloud services.
- Cloud IAM roles can be mapped to Kubernetes service accounts, establishing a connection between the Cloud permissions and Kubernetes RBAC.



Multi-Cluster Considerations

Multi-cluster considerations in Kubernetes involve applying security practices across multiple Kubernetes clusters to ensure consistent protection for workloads.

This is only possible if you manage to apply the concept of **"single source of truth"**, that's usually what we call**"GitOps".** It refers to the practice of managing and enforcing security configurations, policies, and changes through a centralized version-controlled repository, typically using Git. This approach ensures consistency, traceability, and audibility in the deployment and management of security-related aspects across clusters.

- **Monitoring and Logging:**
    - **Unified Observability:** Implement centralized monitoring and logging solutions to gain visibility into the health and performance of workloads across clusters.
    - **Alerting:** Set up consistent alerting mechanisms to detect and respond to security incidents or anomalies in any cluster.

- **Networking and Communication:**
    - **Cluster Networking:** Standardize networking configurations, especially for inter-cluster communication. Consider solutions like service meshes for secure and consistent communication between services across clusters.
    - **Ingress and API Gateways:** Use consistent ingress and API gateway configurations to manage external access to services across clusters.
- **Secrets Management:**
    - **Centralized Secrets Storage:** Implement a centralized secrets management solution to store and distribute sensitive information consistently across clusters.
    - **Encryption:** Apply encryption consistently, especially for communication between clusters, and use a unified approach for key management.

- **Consistent Security Policies:**
  - **Centralized Policies:** Define security policies centrally to ensure consistency across all clusters. This includes network policies, access controls, and security configurations.
  - **Policy as Code:** Use tools like Open Policy Agent (OPA) or Kyverno to implement policy as code, enabling the definition and enforcement of security policies across clusters.
- **Identity and Access Management (IAM):**
  - **Single Sign-On (SSO):** Implement SSO solutions for unified identity management across clusters. This allows users to use a single set of credentials to access resources in different clusters.
  - **RBAC Across Clusters:** Ensure consistent RBAC (Role-Based Access Control) configurations to manage user access and permissions consistently.

- **Compliance and Auditing:**
  - **Centralized Auditing:** Ensure that auditing capabilities are uniformly implemented across clusters for compliance purposes.
  - **Continuous Compliance Checks:** Use automated tools to perform continuous compliance checks across clusters and enforce security baselines.
- **Data Management:**
  - **Data Replication and Backup:** Implement consistent data replication and backup strategies across clusters to ensure data resilience and recovery.
  - **Data Encryption:** Apply encryption consistently to protect data at rest and in transit across clusters.
- **Disaster Recovery:**
  - **DR Planning:** Develop a consistent disaster recovery (DR) plan that spans multiple clusters. Consider solutions like Kubernetes Federation (kubefed) or similar tools for disaster recovery across clusters.
- **Resource Allocation and Scaling:**
  - **Resource Allocation Policies:** Define consistent resource allocation policies to ensure fair resource usage across clusters.
  - **Scaling Strategies:** Implement similar scaling strategies across clusters based on workload demands.

Multi-Tenant cluster Considerations

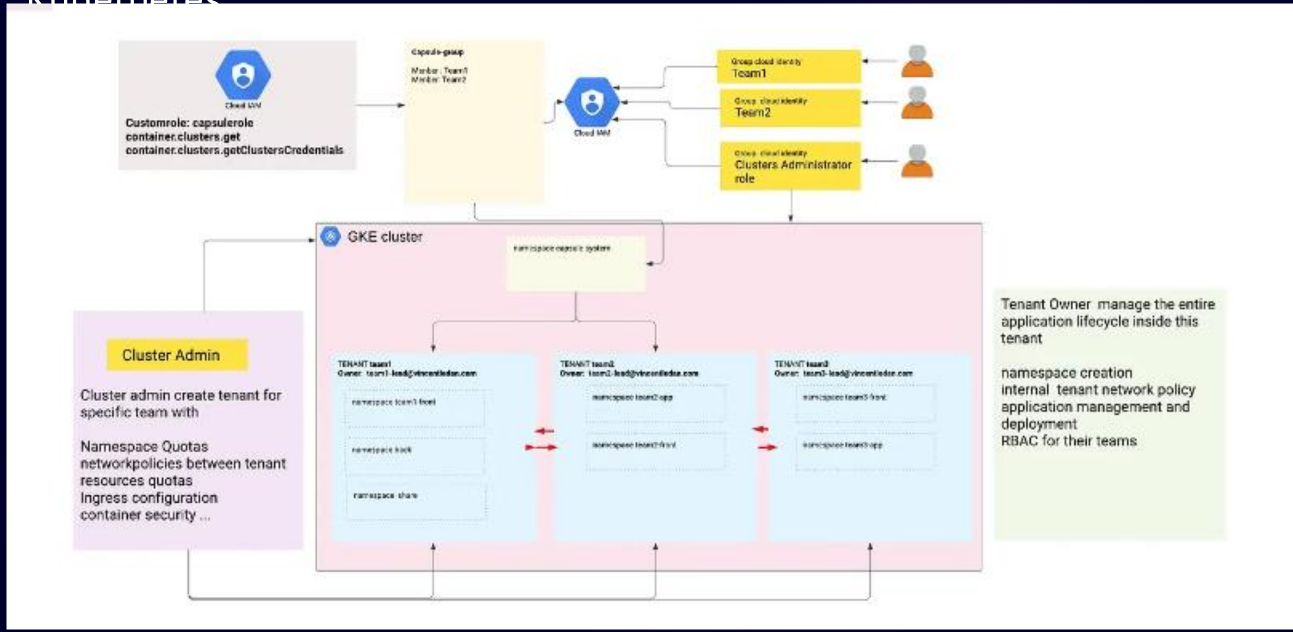
Multitenant cluster can reduce cost at scale because you manage only a small list of cluster but it same time , you implement strong security challenge

- You need to be really focus on scalability on the cluster without affected other team.
- You need to define strategie for segreagtion of teams in the cluster (by group of nodes , who has the priority)
- Implement strong admission controller strategy
- Define clearly networking traffic topology

Capsule

Capsule is a multi-tenant management solution for Kubernetes that uses Kubernetes annotations to manage namespaces, network policies and resource quotas for each tenant. Annotations are used to label Kubernetes objects for each tenant, making it easy to manage multiple tenants on a shared cluster.

Capsule allows multiple product teams to share a Kubernetes cluster while ensuring efficient tenant and resource isolation. It provides efficient resource management by setting quotas for each tenant. Capsule is a mature and widely used open-source solution for multi-tenant management in Kubernetes.

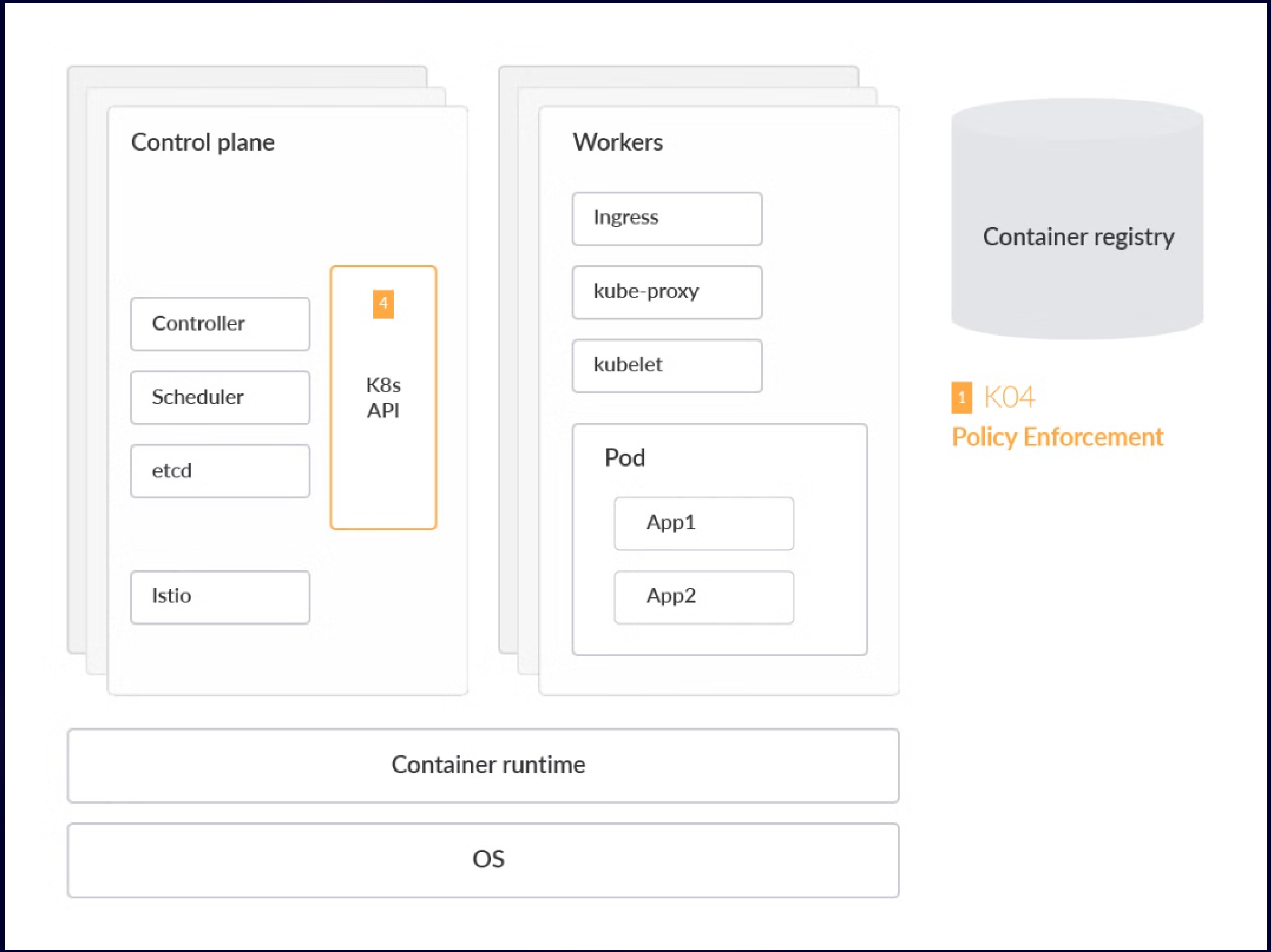


Implement the righ level of security based on your application

- Classify your application and risks associed
- Classifiy the data that used in your application
- Implement the associed security bases on your compliance requierements (backup , encryption , TLS , rbac, logging ...)
- Avoid to intall all applications in the same cluster is they have differents security requierements

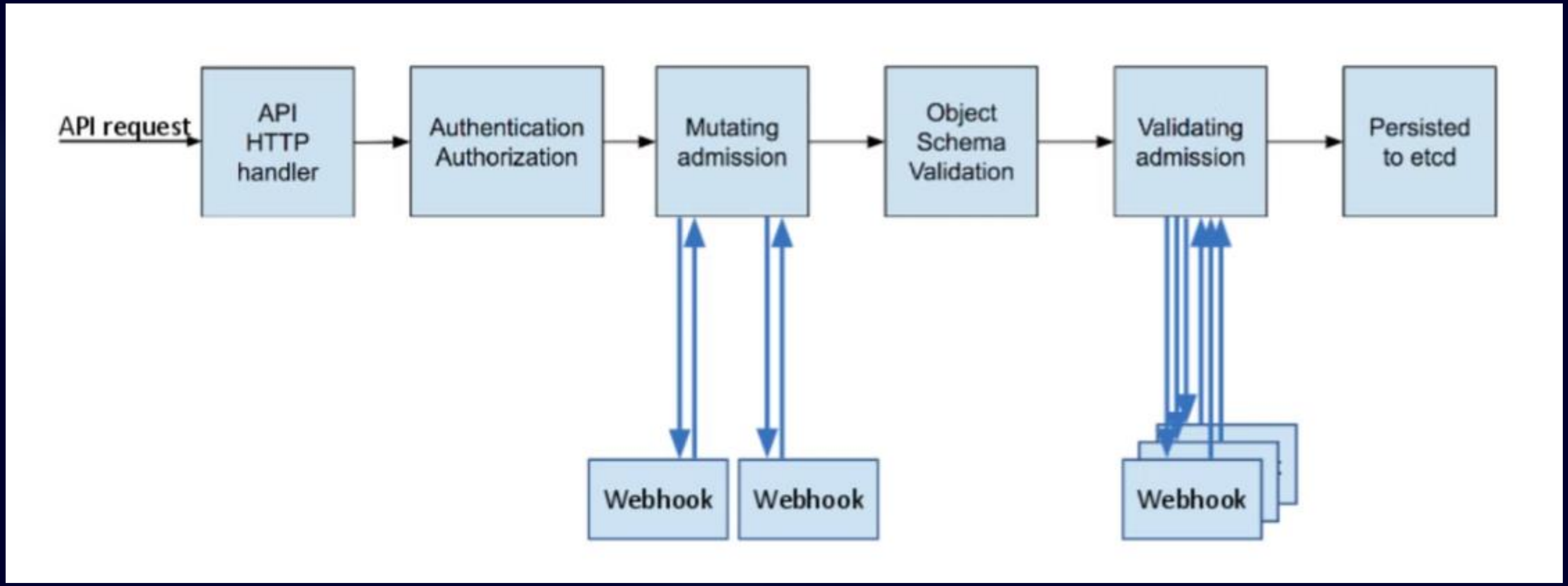
Lunch Time !

K04 : Policy Enforcement



Admission Controller

Admission controller is the logic to implement compliance as code to avoid to run misconfigured deployment or any other king need to be deployed in the cluster.



Native Validation Admission Controller, in the Kubernetes ecosystem, OPA (Open Policy Agent) and Kyverno have been popular choices for admission controllers. However, with the introduction of Validating Admission Controllers in Kubernetes 1.26, Kubernetes now provides mature and relevant solutions with its beta on 1.28.

Use cases

- Do not run pods as root
- Do not expose an application with a NodePort
- Do not mount local node disks in pods
- Have certain mandatory labels for invoicing for example
- Do not deploy without network policies
- Control the number of replicas in a deployment
- Ban orphaned pods without deployment
- Deny all Kernel CAPABILITIES

ConstraintTemplate : defines the logic of the policy and contains the control in the Rego language, such as "maximum replicas for a deployment cannot exceed 5."

```yaml
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: k8sblocknodeport
  annotations:
    metadata.gatekeeper.sh/title: "Block NodePort"
    metadata.gatekeeper.sh/version: 1.0.0
    description: >-
      Disallows all Services with type NodePort.

      https://kubernetes.io/docs/concepts/services-networking/service/#nodeport
spec:
  crd:
    spec:
      names:
        kind: K8sBlockNodePort
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8sblocknodeport

        violation[{"msg": msg}] {
          input.review.kind.kind == "Service"
          input.review.object.spec.type == "NodePort"
          msg := "User is not allowed to create service of type NodePort"
        }
```

Template : utilizes this policy to apply it to specific POD components, filtering the target with parameters.

```yaml
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sBlockNodePort
metadata:
  name: block-node-port
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Service"]
```

Native admission validating K8s

ValidatingAdmissionPolicy : defines the control policy, but this time in the CEL language.

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: ValidatingAdmissionPolicy
metadata:
  name: deny-nodeport
spec:
  failurePolicy: Fail
  matchConstraints:
    resourceRules:
    - apiGroups:   [""]
      apiVersions: ["v1"]
      operations:  ["CREATE", "UPDATE"]
      resources:   ["services"]
  validations:
    - expression: object.spec.type != "NodePort"
```

ValidatingAdmissionPolicyBinding : follows the same principle as a template. It filters the scope and parameters based on the context.

```yaml
apiVersion: admissionregistration.k8s.io/v1beta1
kind: ValidatingAdmissionPolicyBinding
metadata:
  name: "vincent-binding-test.example.com"
spec:
  policyName: "deny-nodeport"
  validationActions: [Deny]
  matchResources:
    namespaceSelector:
      matchLabels:
        env: prod
```

Kyverno

Now that you've managed to work with the Native Admission Controller, you've probably found it quite on "beta mode" ... Indeed, this feature is not yet production-ready but is heading in the right direction. When compared to Kyverno, for instance, Kyverno offers more flexibility and automation.
Let's dive a little deeper inside Kyverno's usage.

As explained, admission controllers monitor and, if necessary, block configurations that do not comply with defined compliance rules.

Kyverno use Admission , mutating admission controller , and can also generate config based on differents scenarios.

A **ClusterPolicy** in Kyverno can automatically generate a NetworkPolicy and ResourceQuota whenever a new namespace is created. This ensures default security configurations are applied consistently across namespaces.

Here is an example, but you'll have a lab ready for you to configure your own policies.

```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: default-policy-and-quota
spec:
  rules:
  - name: create-default-network-policy
    match:
      resources:
        kinds:
        - Namespace
    generate:
      kind: NetworkPolicy
      name: default-network-policy
      namespace: "{{request.object.metadata.name}}"
      data:
        spec:
          podSelector: {}
          policyTypes:
          - Ingress
          - Egress
  - name: create-default-resource-quota
    match:
      resources:
        kinds:
        - Namespace
    generate:
      kind: ResourceQuota
      name: default-resource-quota
      namespace: "{{request.object.metadata.name}}"
      data:
        spec:
          hard:
            pods: "10"
            requests.cpu: "1000m"
            requests.memory: "1Gi"
            limits.cpu: "2000m"
            limits.memory: "2Gi"
```

LAB Kyverno

"As Code"

1. **Infrastructure as Code (IaC):**
    - **Definition:** Writing code to describe and provision infrastructure components, such as servers, networks, and databases.
    - **Benefits:** Consistent, repeatable, and automated infrastructure deployment. Changes are versioned, providing better control and traceability.
2. **Policy as Code:**
    - **Definition:** Expressing security, compliance, and governance policies in a programming language.
    - **Benefits:** Policies become version-controlled, auditable, and shareable. Automation tools can enforce policies consistently across environments.
3. **Configuration as Code:**
    - **Definition:** Representing configuration settings, preferences, and parameters in code.
    - **Benefits:** Standardized configurations, easy replication, and the ability to track changes over time. Enhances collaboration and simplifies configuration management.
4. **Security as Code:**
    - **Definition:** Applying software development practices to security processes, integrating security checks into the development lifecycle.
    - **Benefits:** Identifying and addressing security issues early in the development process. Automated security testing and compliance checking.
5. **Governance as Code:**
    - **Definition:** Codifying governance policies and procedures to ensure consistency and compliance across an organization.
    - **Benefits:** Enables automation of governance checks, simplifies auditing, and provides a transparent way to enforce organizational standards.

Encryption

Encryption plays a crucial role in securing data, both in transit and at rest.

Encryption in Transit:

Encryption in transit refers to securing data as it travels between different components and services within the Kubernetes cluster or between the cluster and external entities.

- **Implementation:**
  - **TLS (Transport Layer Security)**
  - **Service Mesh Encryption**

Encryption at Rest:

Encryption at rest involves protecting data when it is stored in persistent storage, such as etcd (Kubernetes datastore), volumes, and other storage backends.

- **Implementation:**
    - **etcd Encryption:**
    - **Volume Encryption**
    - **Secrets Encryption**

Key Management:

Key management is a critical aspect of encryption, involving the generation, storage, rotation, and distribution of cryptographic keys.

- **Implementation:**
    - **KMS Integration:** Kubernetes can integrate with Key Management Systems (KMS) to manage encryption keys securely. KMS solutions, such as HashiCorp Vault or cloud provider-specific key management services, can be used to generate and manage encryption keys.

Pod-to-Pod Encryption:

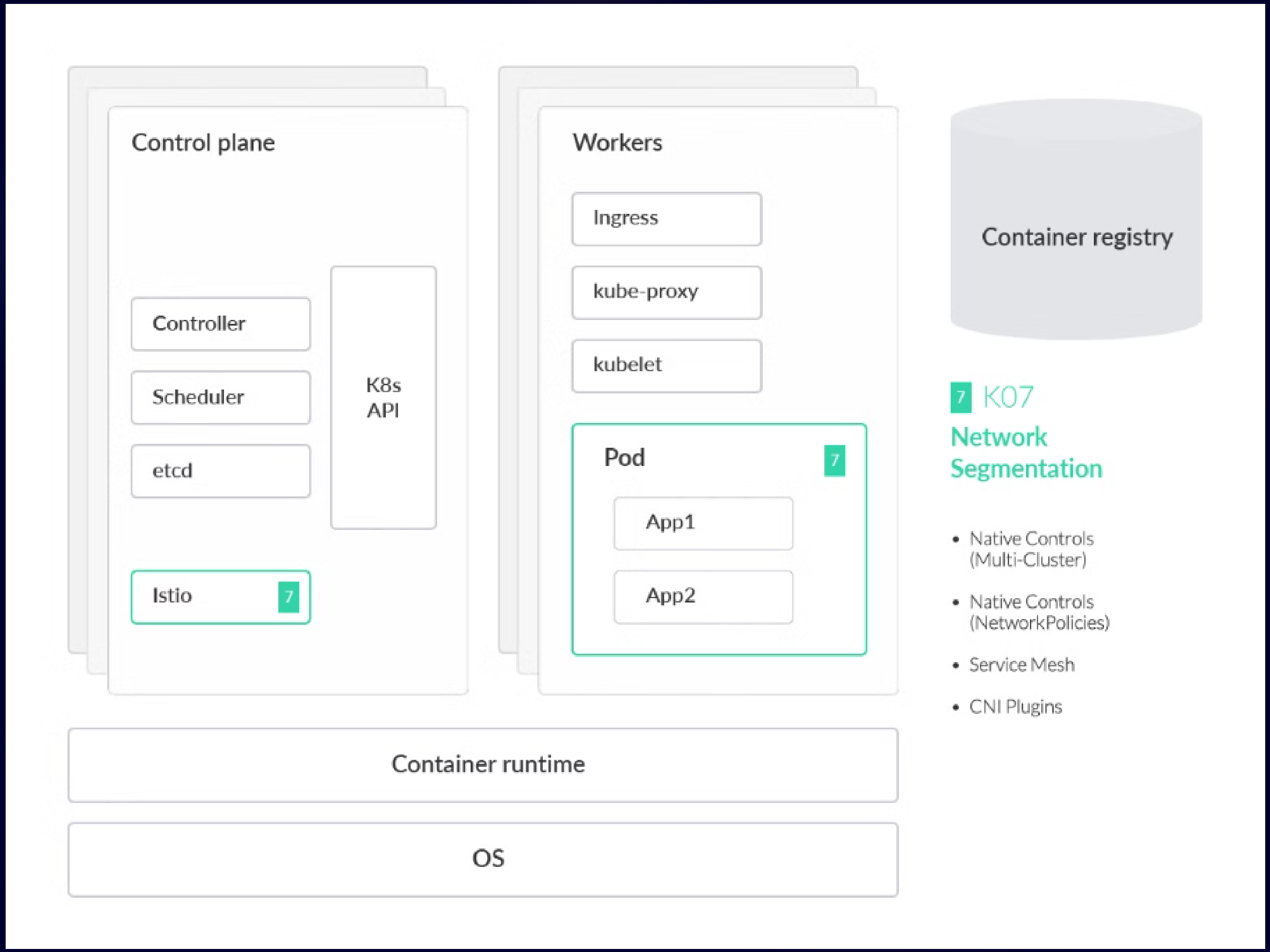
Ensuring encryption for data exchanged between pods within the cluster.

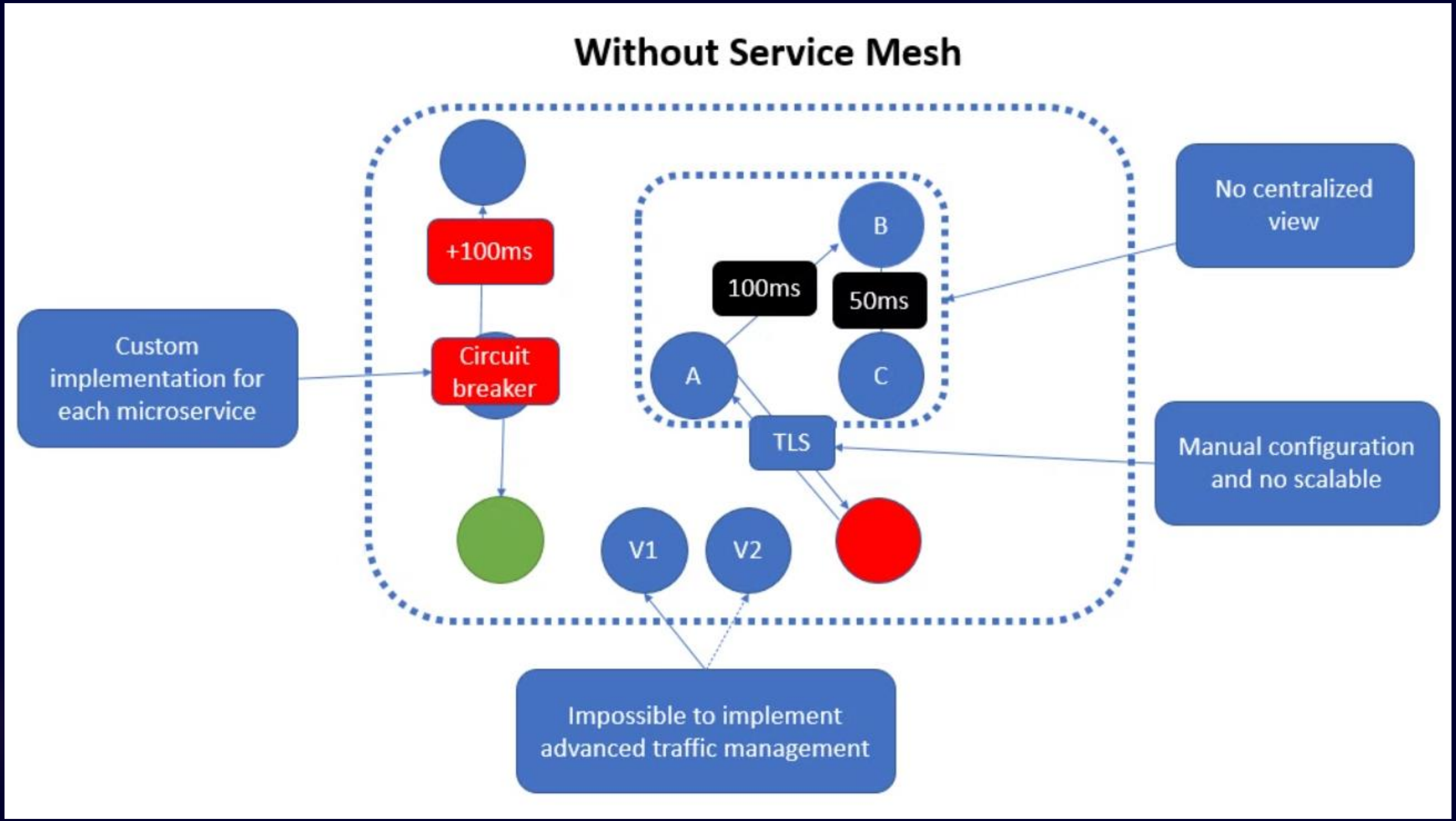
- **Implementation:**
  - **Network Policies + Mesh :** Implementing Network Policies to control pod-to-pod communication can enhance security. When combined with network encryption mechanisms like mTLS, it provides a layered approach to secure inter-pod communication.

- **Ingress and Load Balancer Encryption:**
    - Securing external communication to services exposed via Ingress or Load Balancer services.
    - **Implementation:**
        - **TLS Termination:** Ingress controllers and Load Balancers can terminate TLS connections, decrypting incoming traffic and forwarding it securely to backend services within the cluster.

Service Mesh

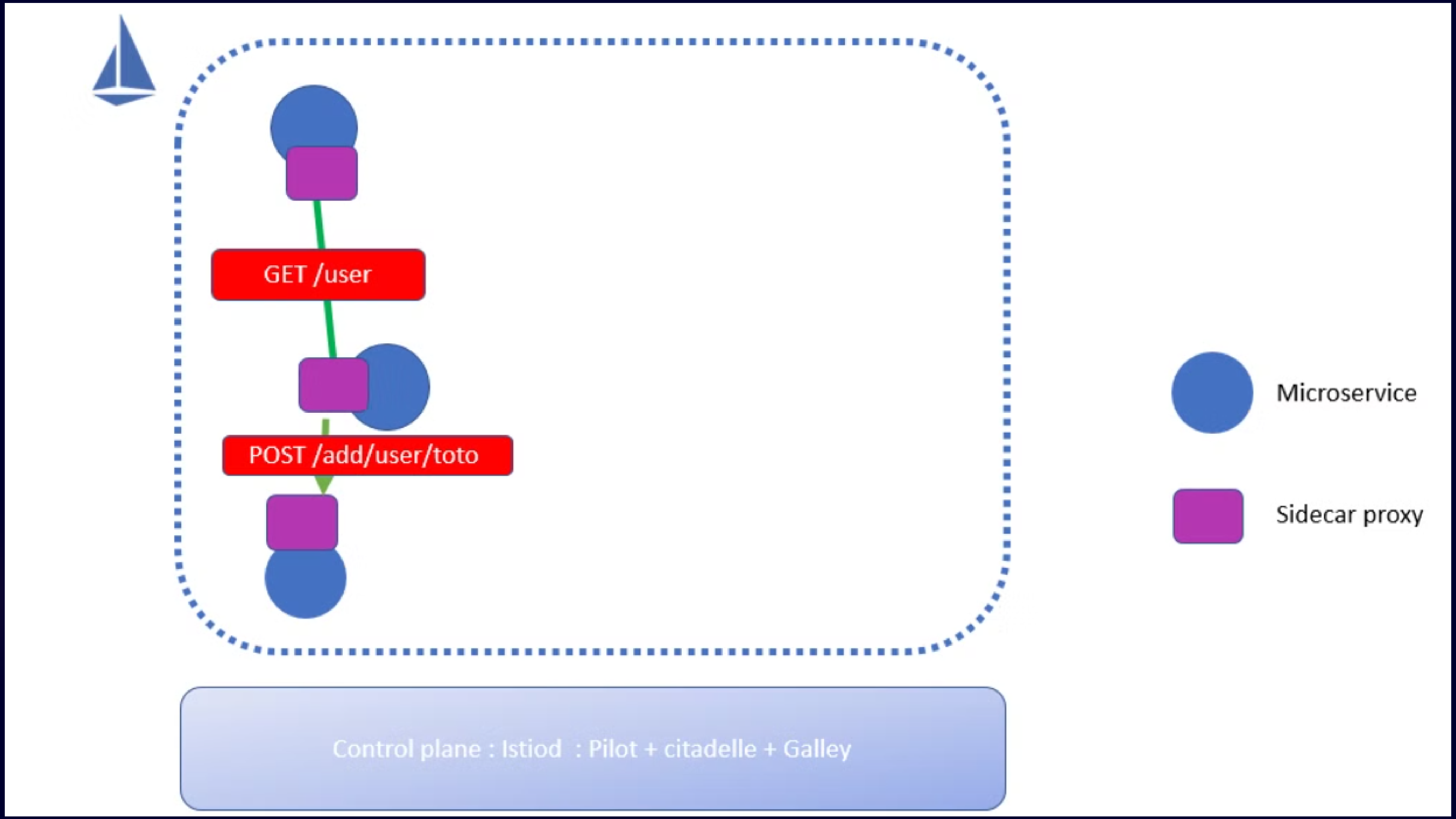
K07 : Network segmentation



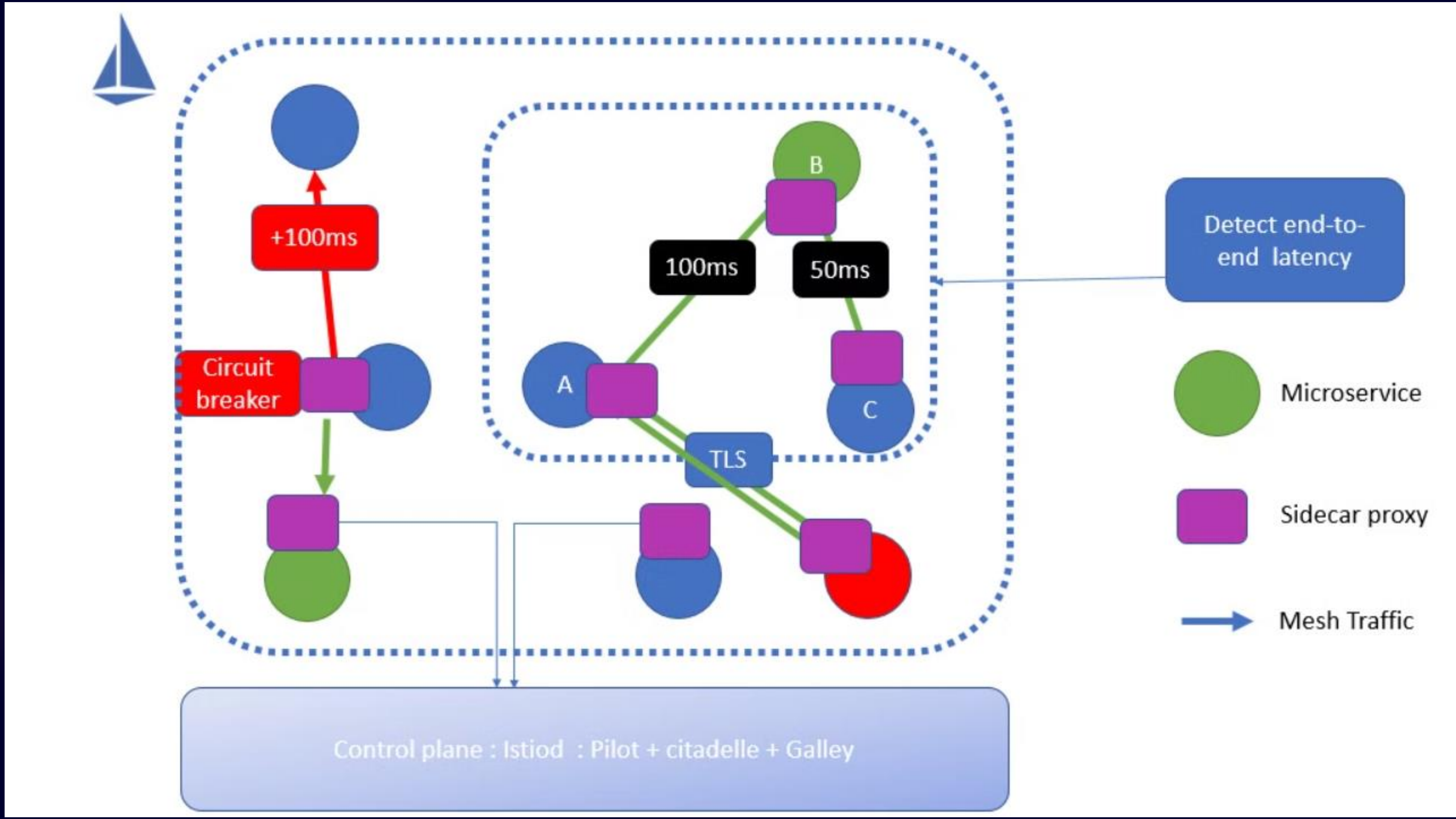

Without Service Mesh
Custom implementation for each microservice
+100ms
Circuit breaker
100ms
50ms
B
A
C
TLS
No centralized view
Manual configuration and no scalable
V1
V2
Impossible to implement advanced traffic management

Service Mesh features

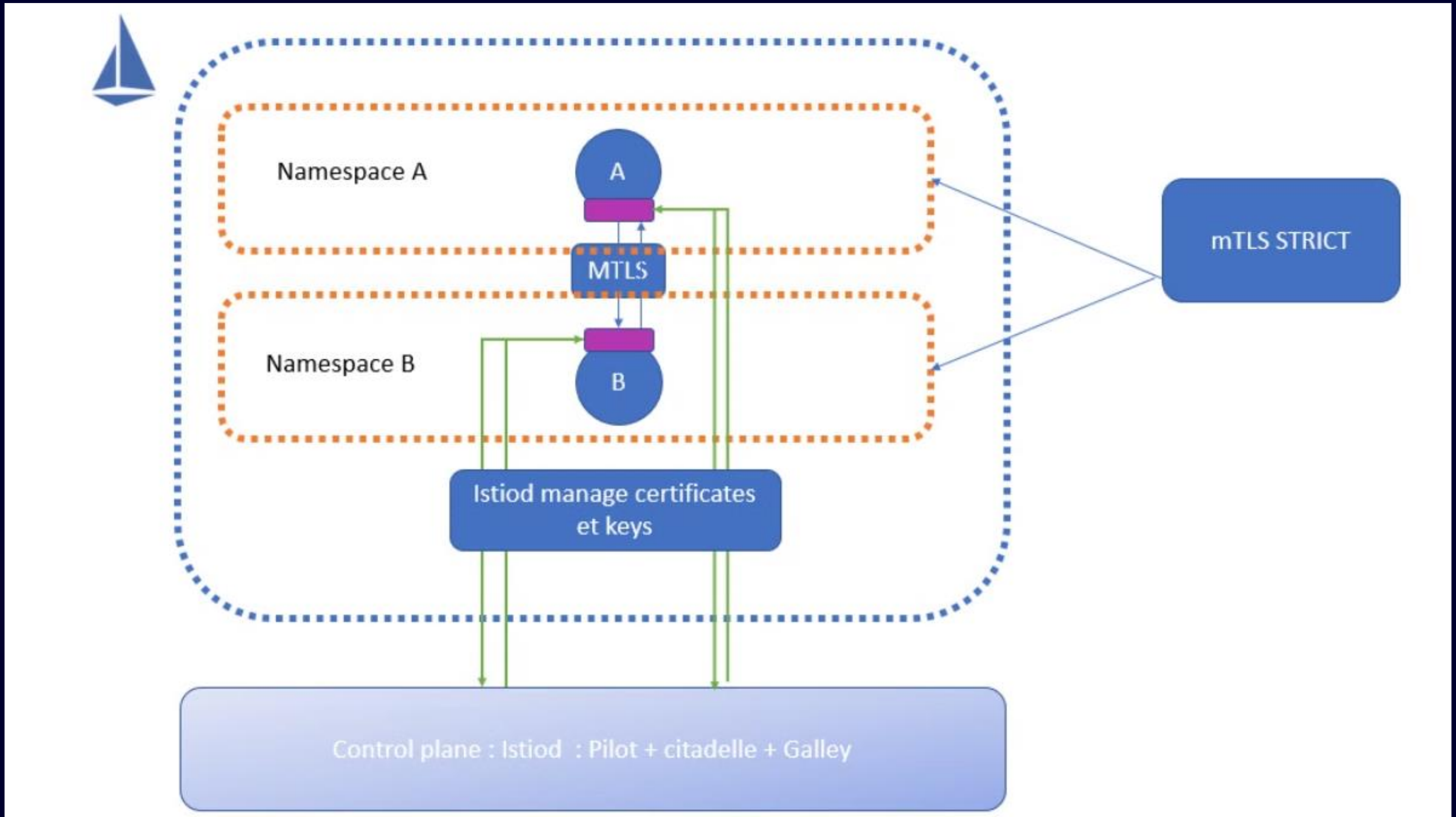
- **Service-to-Service Communication:** A service mesh ensures reliable and efficient communication between microservices. It handles tasks such as load balancing, service discovery, and communication protocols, freeing developers from dealing with these concerns at the application level.



- **Traffic Management:** Service meshes allow for sophisticated traffic management strategies like A/B testing, canary releases, and blue-green deployments. This enables controlled and gradual rollouts of new features or updates, minimizing the impact of changes on the overall system.



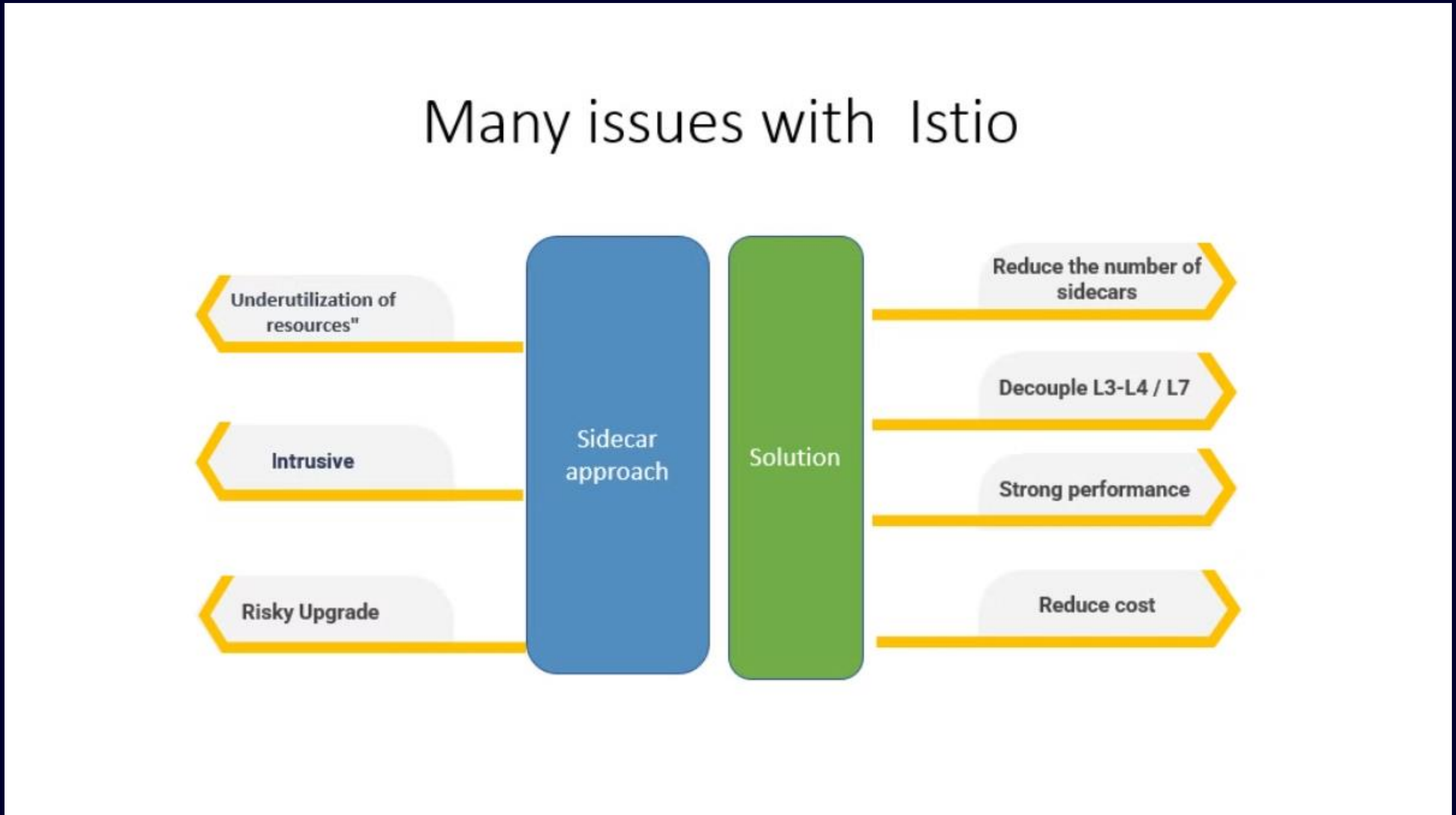
- **Security:** Security features include end-to-end encryption, authentication, and authorization. Service meshes can enforce policies to control which services can communicate with each other and under what conditions. This enhances the overall security posture of the microservices architecture.



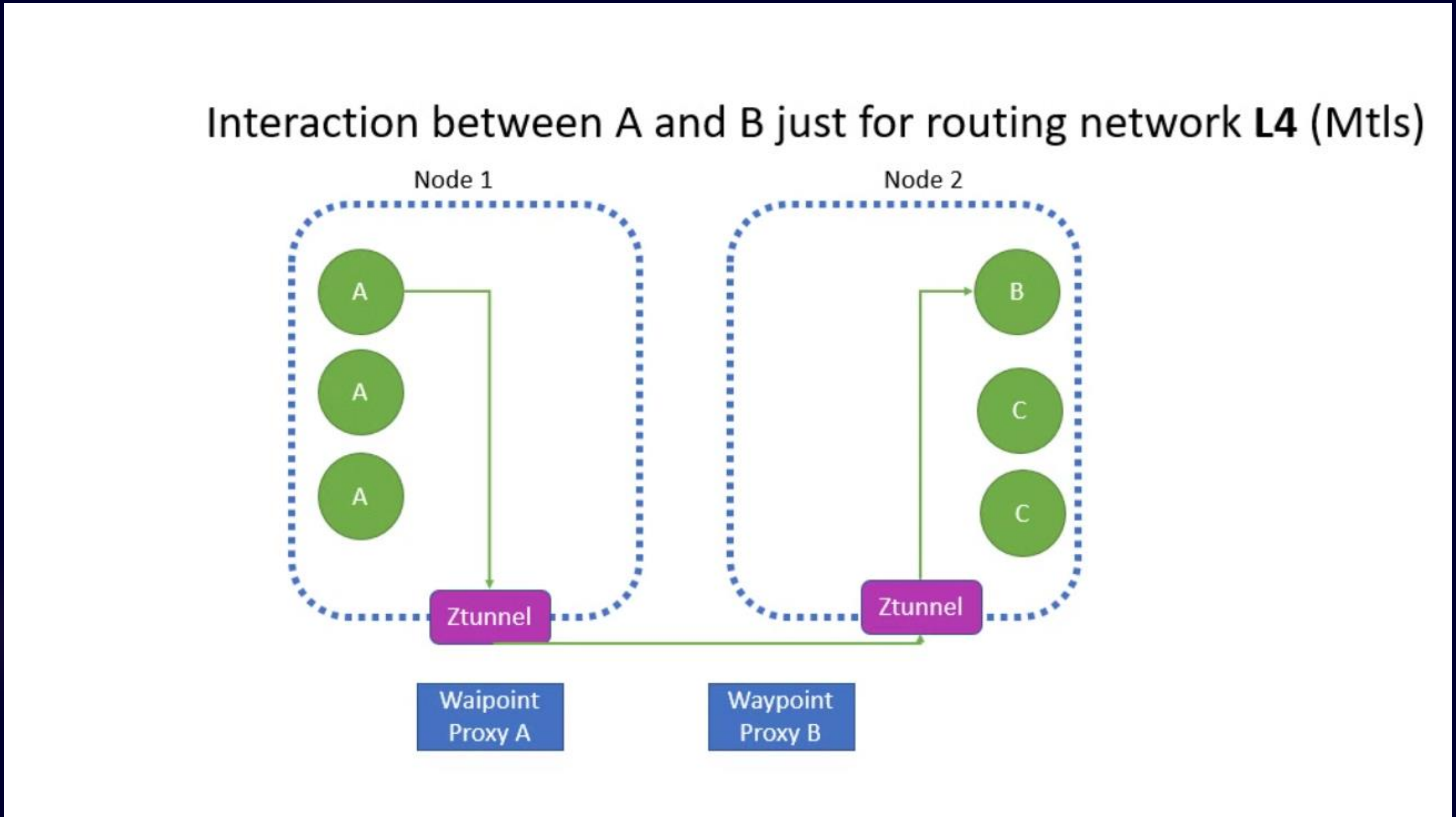
- **Observability and Monitoring:** Service meshes provide robust observability tools, including metrics, logging, and tracing. This visibility into the microservices interactions helps in diagnosing issues, monitoring performance, and optimizing the application.
- **Resilience and Fault Tolerance:** Service meshes enhance the resilience of microservices by handling retries, timeouts, and circuit breaking. They automatically manage failures, rerouting traffic away from unhealthy services to maintain overall system stability.
- **Platform-Agnostic:** While commonly associated with Kubernetes, service meshes can operate across different container orchestration platforms. This flexibility makes them suitable for multi-cloud or hybrid cloud environments.

Future of Service Mesh architecture

Actual standard architecture is based on **sidecar** logic and can present especially at scale many problems :

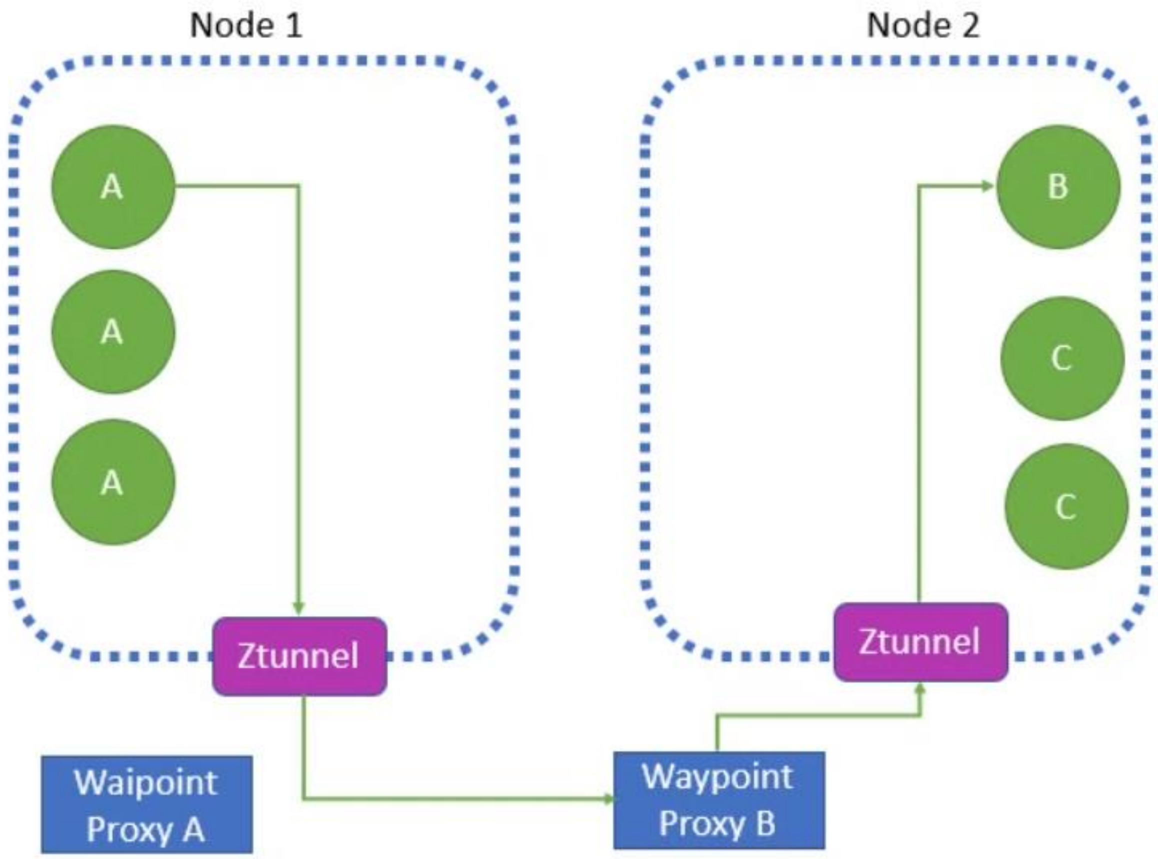


Ambient mesh is born



Made with Gamma

Interaction between A and B with **L7 routing**



Runtime & Malicious Activity Protection

1

Sandboxing

Learn how to implement sandboxing techniques to isolate and restrict the behavior of potentially malicious workloads.

2

Observability and Monitoring

Discover effective monitoring strategies to detect and respond to security incidents in real-time.

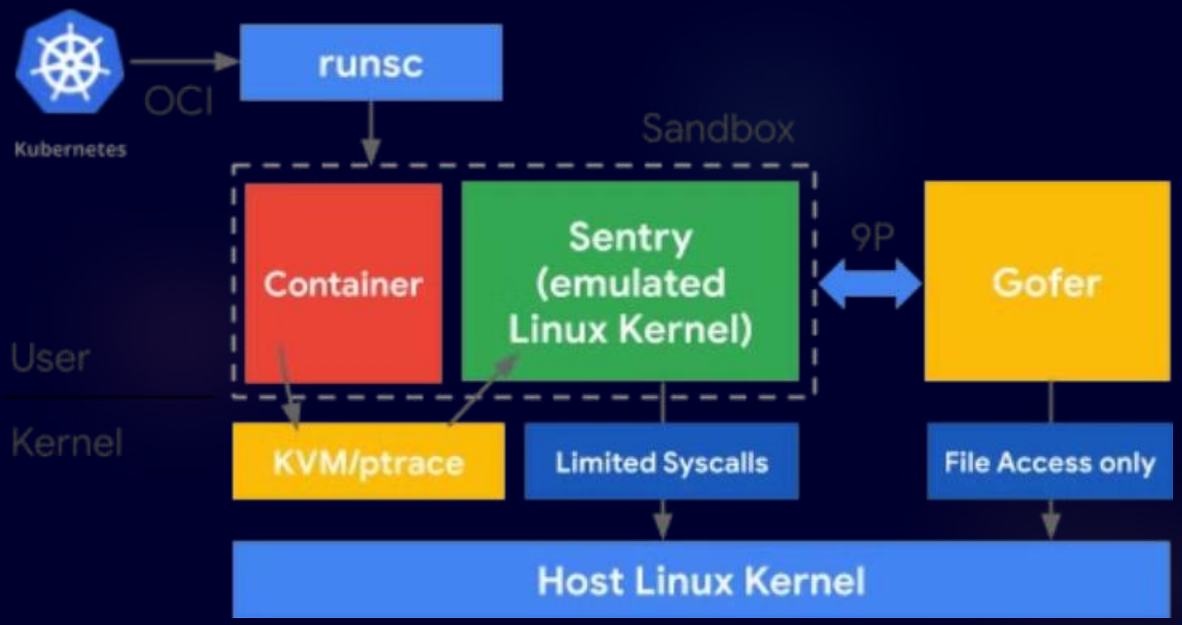
Sandboxing

In the context of container runtimes, it provides an additional layer of isolation for workloads.

gVisor is an open-source user-space kernel that provides a container runtime with a sandboxed environment. It is designed to offer lightweight and secure isolation by intercepting and handling system calls in user space rather than relying on the host kernel. It implements a subset of the Linux kernel APIs to achieve compatibility. An other example of these type of tools would be **Katacontainer**.

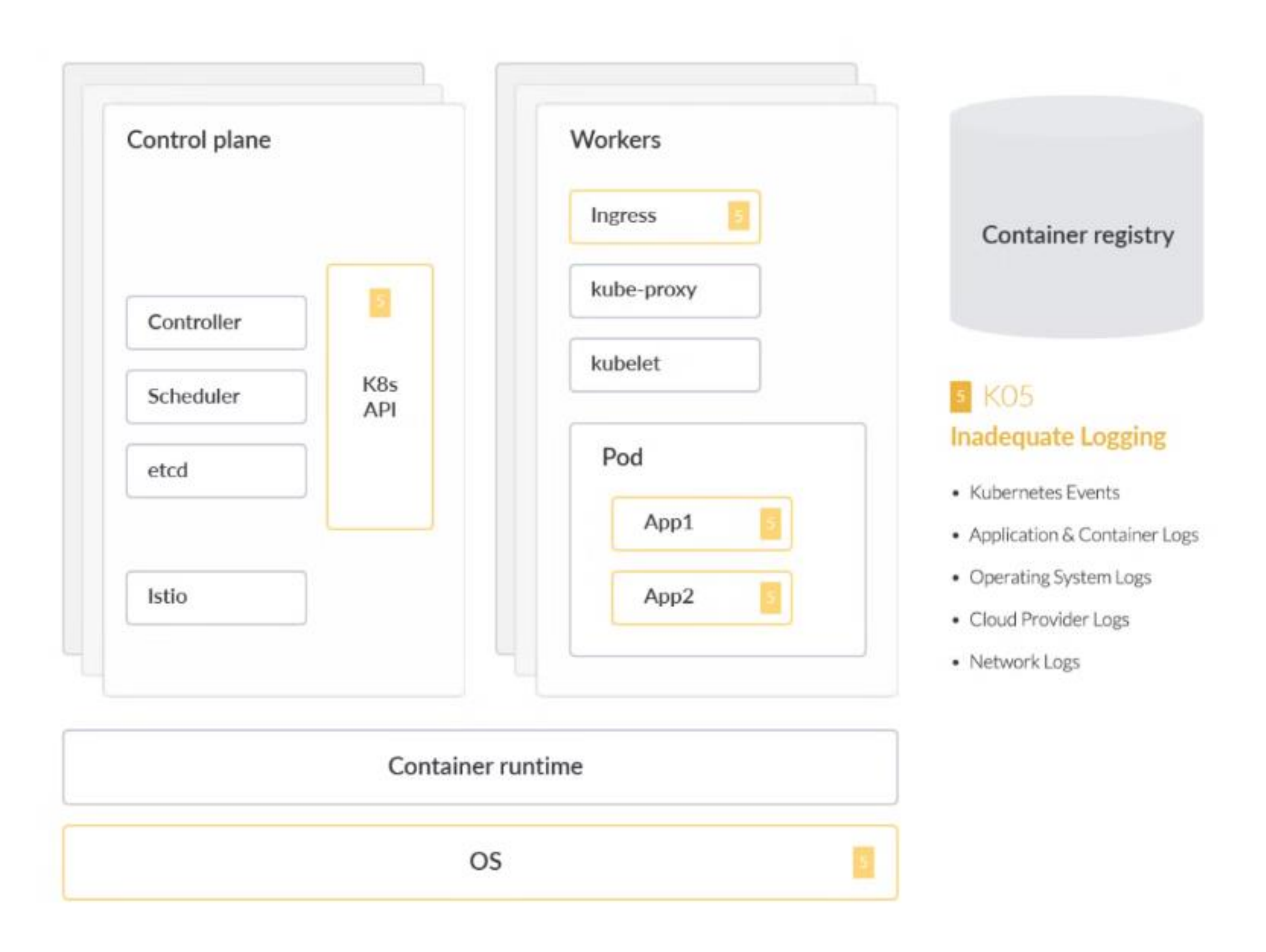
To go a step further, we could talk about **Microvirtualization** which is a technique where lightweight VMs are used for individual workloads.

- Each workload operates in its dedicated micro-VM, ensuring strong isolation.



Observability & Monitoring

K05 : Inadequate logging



Observability and monitoring play a crucial role in maintaining the security and health of a cluster. These practices involve the collection, analysis, and visualization of various metrics, logs, and events to detect and respond to security incidents in real-time. Here are key concepts and strategies related to observability and monitoring in Kubernetes:

Falco

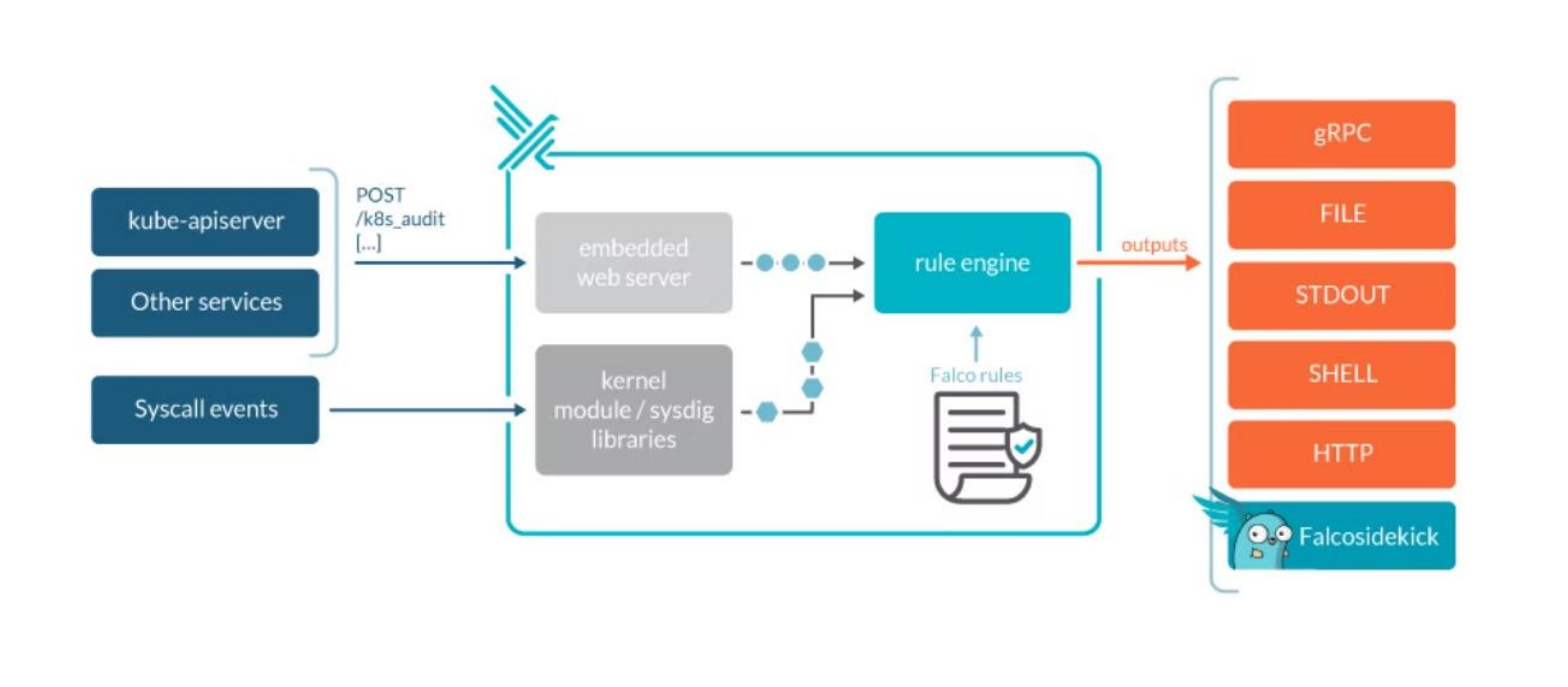
Real-time threat detection solution for containers, hosts, Kubernetes and the cloud

Falco provides real-time detection capabilities for environments from individual containers, hosts, Kubernetes and the cloud. It is able to detect and alert on abnormal behavior and potential security threats in real-time, such as **crypto mining**, **file exfiltration**, **privilege escalation** in applications, rootkit installs among many others. These malicious behaviors are detected via user-defined **Falco rules** that classify events of application activity as malicious or suspicious.

More specifically, Falco collects event data from a *source* and compares each event against a set of *rules*. Some examples of sources for Falco events are:

- Linux kernel syscalls
- Kubernetes audit logs
- Cloud events (e.g. AWS CloudTrail)
- Events from other systems (GitHub, Okta)
- New data sources can be added to Falco by developing **plugins**

How it works


kube-apiserver
Other services
POST
/k8s_audit
[...]
Syscall events
embedded
web server
kernel
module / sysdig
libraries
rule engine
Falco rules
outputs
gRPC
FILE
STDOUT
SHELL
HTTP
Falcosidekick

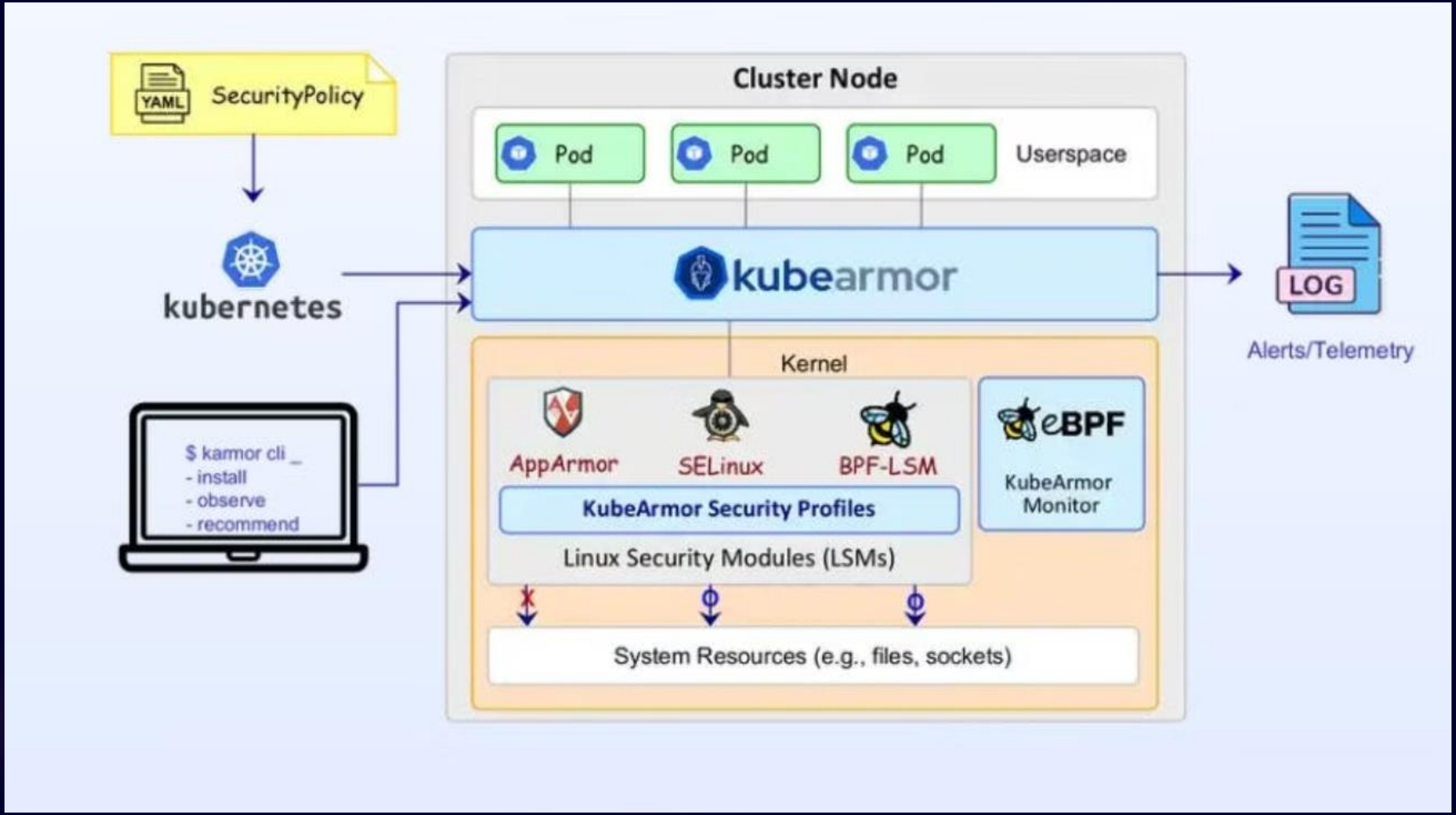
Exemple of rules for open shell in container

```
- macro: container
  condition: container.id != host

- macro: spawned_process
  condition: evt.type = execve and evt.dir=<

- rule: run_shell_in_container
  desc: a shell was spawned by a non-shell program in a container.
Container entrypoints are excluded.
  condition: container and proc.name = bash and spawned_process and
proc.pname exists and not proc.pname in (bash, docker)
  output: "Shell spawned in a container other than entrypoint
(user=%user.name container_id=%container.id container_name=%container.name
shell=%proc.name parent=%proc.pname cmdline=%proc.cmdline)"
  priority: WARNING
```

KubeArmor

KubeArmor is an open-source security solution designed to strengthen container security on Kubernetes in real-time. It utilizes access control and inline remediation techniques to prevent attacks. The approach is to proactively block abnormal behaviors before they occur, minimizing the impact on applications.

- **Real-time Protection:** Blocks attacks as they happen.
- **Inline Remediation:** Corrects security violations in real-time, reducing damages.
- **Flexibility:** Allows administrators to define custom security policies.
- **Simplicity:** Easy to configure and use.
- **Performance:** Minimal impact on container performance.
- **Audit and Reporting:** Provides audit logs and reports for container activity.
- **Integration with Other Tools:** Can be integrated with SIEM (Security Information and Event Management) and SOAR (Security Orchestration, Automation, and Response).



YAML
SecurityPolicy
kubernetes
$ karmor cli _
- install
- observe
- recommend
Cluster Node
Pod
Pod
Pod
Userspace
kubearmor
Kernel
AppArmor
SELinux
BPF-LSM
KubeArmor Security Profiles
Linux Security Modules (LSMs)
eBPF
KubeArmor Monitor
System Resources (e.g., files, sockets)
LOG
Alerts/Telemetry

Process exemple : deny apt command

COPY

```yaml
apiVersion: security.kubearmor.com/v1
kind: KubeArmorPolicy
metadata:
  name: ksp-wordpress-block-process
  namespace: default
spec:
  severity: 3
  selector:
    matchLabels:
      app: nginx
  process:
    matchPaths:
    - path: /usr/bin/apt
    - path: /usr/bin/apt-get
  action: Block
```

LAB Runtime protection

Security Overtime

1. ## Compliance

Calculate the total cost of ownership (TCO) of implementing and maintaining robust Kubernetes security practices.

2. ## Learning

Discover continuous learning opportunities to stay up-to-date with the latest Kubernetes security advancements.

3. ## Day2

Understand the importance of ongoing security practices and monitoring post-deployment.

4. ## CNAPP / Team Topology / Open Source / Risk Prioritization

Explore additional topics including security assessments, team organization, leveraging open-source tools, and risk management.

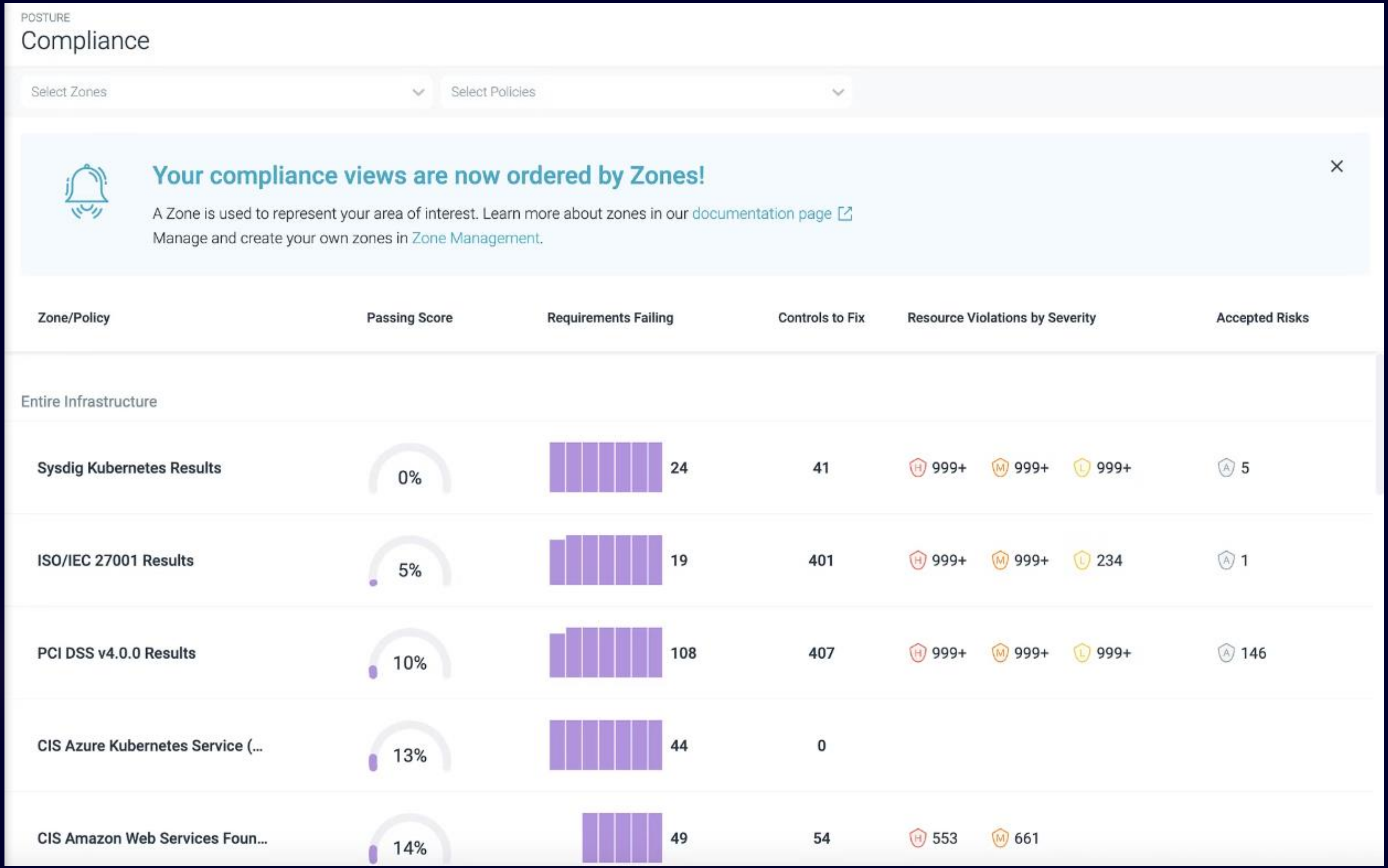
Compliance

Many tools like Kubebench , trivy , cloud providers , sysdig will provide to you CIS bencharmark

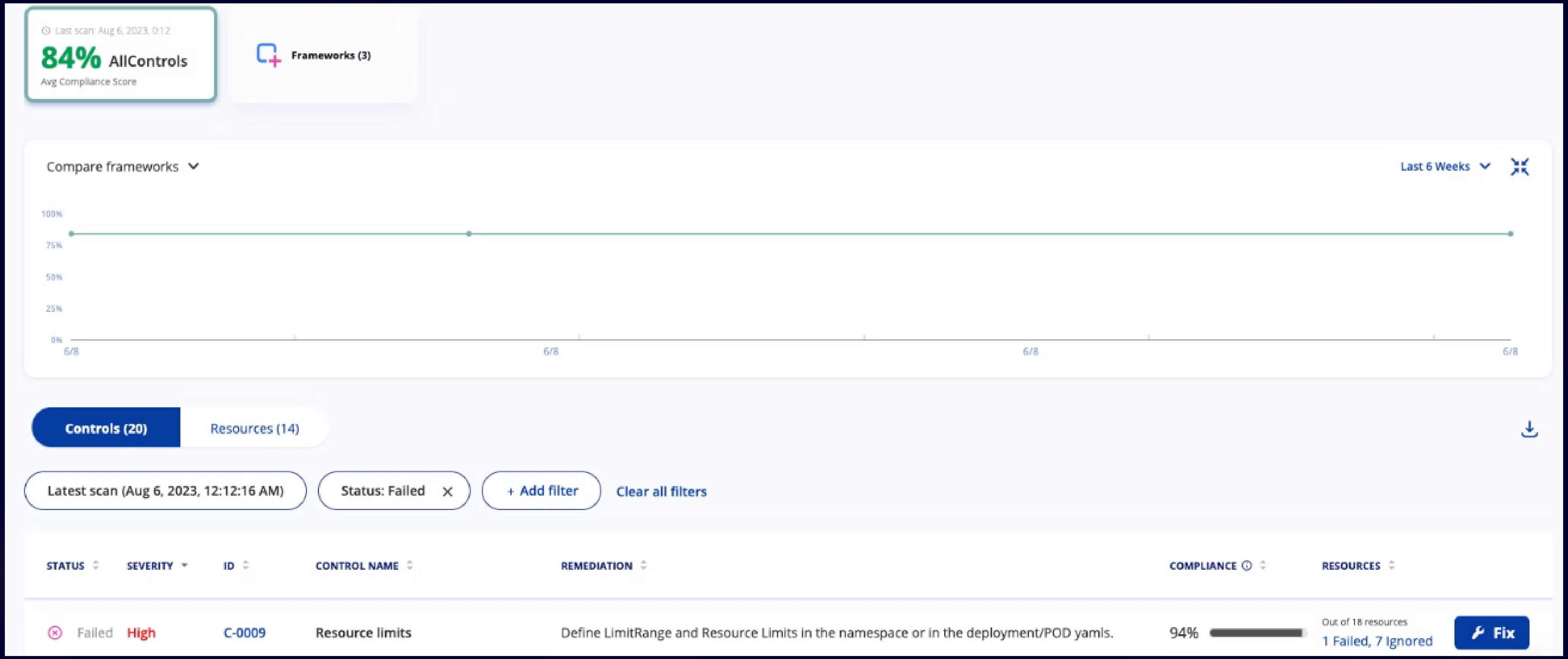
Kubebench for EKS

```
[INFO] 4 Policies
[INFO] 4.1 RBAC and Service Accounts
[WARN] 4.1.1 Ensure that the cluster-admin role is only used where required (Manual)
[WARN] 4.1.2 Minimize access to secrets (Manual)
[WARN] 4.1.3 Minimize wildcard use in Roles and ClusterRoles (Manual)
[WARN] 4.1.4 Minimize access to create pods (Manual)
[WARN] 4.1.5 Ensure that default service accounts are not actively used. (Manual)
[WARN] 4.1.6 Ensure that Service Account Tokens are only mounted where necessary (Manual)
[INFO] 4.2 Pod Security Policies
[WARN] 4.2.1 Minimize the admission of privileged containers (Automated)
[WARN] 4.2.2 Minimize the admission of containers wishing to share the host process ID namespace (Automated)
[WARN] 4.2.3 Minimize the admission of containers wishing to share the host IPC namespace (Automated)
[WARN] 4.2.4 Minimize the admission of containers wishing to share the host network namespace (Automated)
[WARN] 4.2.5 Minimize the admission of containers with allowPrivilegeEscalation (Automated)
[WARN] 4.2.6 Minimize the admission of root containers (Automated)
[WARN] 4.2.7 Minimize the admission of containers with the NET_RAW capability (Automated)
[WARN] 4.2.8 Minimize the admission of containers with added capabilities (Automated)
[WARN] 4.2.9 Minimize the admission of containers with capabilities assigned (Manual)
[INFO] 4.3 CNI Plugin
[WARN] 4.3.1 Ensure that the latest CNI version is used (Manual)
[WARN] 4.3.2 Ensure that all Namespaces have Network Policies defined (Automated)
[INFO] 4.4 Secrets Management
[WARN] 4.4.1 Prefer using secrets as files over secrets as environment variables (Manual)
[WARN] 4.4.2 Consider external secret storage (Manual)
[INFO] 4.5 Extensible Admission Control
[WARN] 4.5.1 Configure Image Provenance using ImagePolicyWebhook admission controller (Manual)
[INFO] 4.6 General Policies
[WARN] 4.6.1 Create administrative boundaries between resources using namespaces (Manual)
[WARN] 4.6.2 Apply Security Context to Your Pods and Containers (Manual)
[WARN] 4.6.3 The default namespace should not be used (Automated)

== Remediations policies ==
4.1.1 Identify all clusterrolebindings to the cluster-admin role. Check if they are used and
if they need this role or if they could use a role with fewer privileges.
Where possible, first bind users to a lower privileged role and then remove the
clusterrolebinding to the cluster-admin role :
kubectl delete clusterrolebinding [name]
```

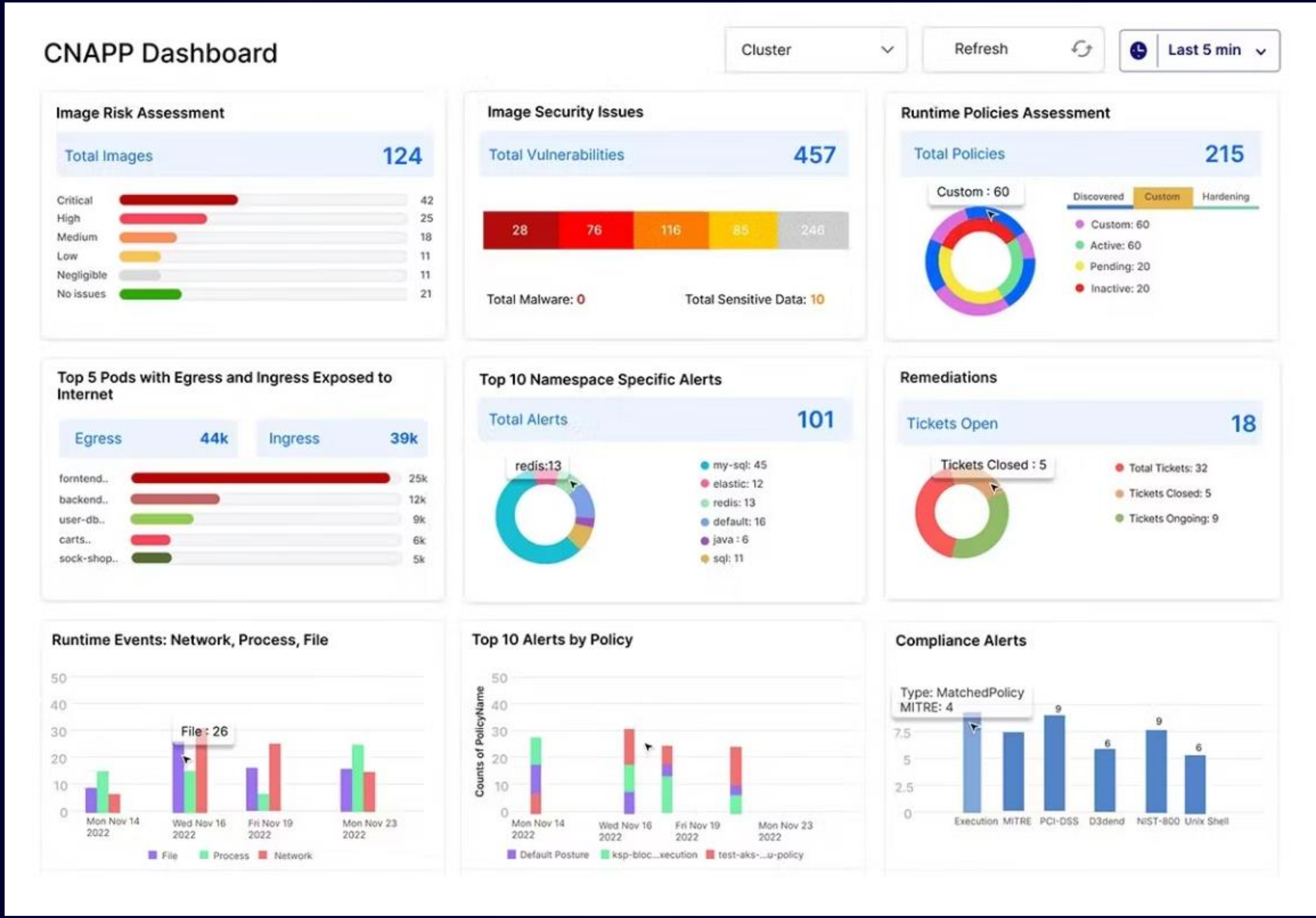
Sysdig


POSTURE
Compliance
Select Zones
Select Policies
Your compliance views are now ordered by Zones!
A Zone is used to represent your area of interest. Learn more about zones in our documentation page
Manage and create your own zones in Zone Management.
Zone/Policy
Passing Score
Requirements Failing
Controls to Fix
Resource Violations by Severity
Accepted Risks
Entire Infrastructure
Sysdig Kubernetes Results
0%
24
41
999+
999+
999+
5
ISO/IEC 27001 Results
5%
19
401
999+
999+
234
1
PCI DSS v4.0.0 Results
10%
108
407
999+
999+
999+
146
CIS Azure Kubernetes Service (...
13%
44
0
CIS Amazon Web Services Foun...
14%
49
54
553
661

Armor platform



Accuknox

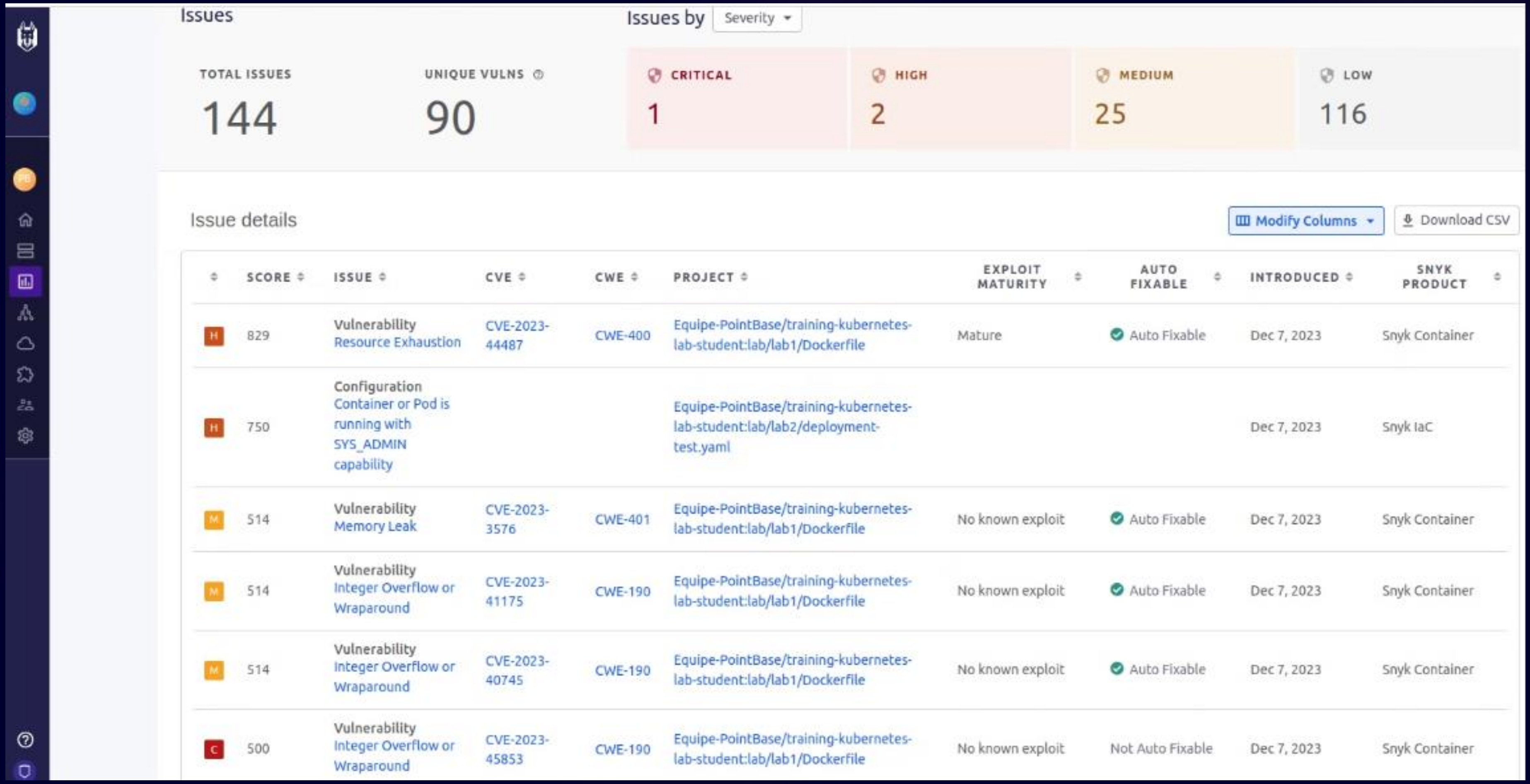


CNAPP Dashboard
Cluster
Refresh
Last 5 min
Image Risk Assessment
Total Images 124
Critical 42
High 25
Medium 18
Low 11
Negligible 11
No issues 21
Image Security Issues
Total Vulnerabilities 457
28 76 116 85 246
Total Malware: 0
Total Sensitive Data: 10
Runtime Policies Assessment
Total Policies 215
Custom : 60
Discovered Custom Hardening
Custom: 60
Active: 60
Pending: 20
Inactive: 20
Top 5 Pods with Egress and Ingress Exposed to Internet
Egress 44k
Ingress 39k
forntend.. 25k
backend.. 12k
user-db.. 9k
carts.. 6k
sock-shop.. 5k
Top 10 Namespace Specific Alerts
Total Alerts 101
redis:13
my-sql: 45
elastic: 12
redis: 13
default: 16
java : 6
sql: 11
Remediations
Tickets Open 18
Tickets Closed : 5
Total Tickets: 32
Tickets Closed: 5
Tickets Ongoing: 9
Runtime Events: Network, Process, File
File : 26
Mon Nov 14 2022
Wed Nov 16 2022
Fri Nov 19 2022
Mon Nov 23 2022
File Process Network
Top 10 Alerts by Policy
Counts of PolicyName
Default Posture
ksp-bloc...xecution
test-aks-...u-policy
Compliance Alerts
Type: MatchedPolicy
MITRE: 4
Execution MITRE PCI-DSS D3dend NIST-800 Unix Shell

SAST for kubernetes

Today we discuted only about Kubernetes security , when we consider "everything as code" we can implement Snyk also to automate directly from source code misconfiguration in your Dockerfile , kubernetes manifest , application code with the same tool.

For this example we connected our student lab repo to SNYK to show you :



	SCORE	ISSUE	CVE	CWE	PROJECT	EXPLOIT MATURITY	AUTO FIXABLE	INTRODUCED	SNYK PRODUCT
H	829	Vulnerability Resource Exhaustion	CVE-2023-44487	CWE-400	Equipe-PointBase/training-kubernetes-lab-student:lab/lab1/Dockerfile	Mature	Auto Fixable	Dec 7, 2023	Snyk Container
H	750	Configuration Container or Pod is running with SYS_ADMIN capability			Equipe-PointBase/training-kubernetes-lab-student:lab/lab2/deployment-test.yaml			Dec 7, 2023	Snyk IaC
M	514	Vulnerability Memory Leak	CVE-2023-3576	CWE-401	Equipe-PointBase/training-kubernetes-lab-student:lab/lab1/Dockerfile	No known exploit	Auto Fixable	Dec 7, 2023	Snyk Container
M	514	Vulnerability Integer Overflow or Wraparound	CVE-2023-41175	CWE-190	Equipe-PointBase/training-kubernetes-lab-student:lab/lab1/Dockerfile	No known exploit	Auto Fixable	Dec 7, 2023	Snyk Container
M	514	Vulnerability Integer Overflow or Wraparound	CVE-2023-40745	CWE-190	Equipe-PointBase/training-kubernetes-lab-student:lab/lab1/Dockerfile	No known exploit	Auto Fixable	Dec 7, 2023	Snyk Container
C	500	Vulnerability Integer Overflow or Wraparound	CVE-2023-45853	CWE-190	Equipe-PointBase/training-kubernetes-lab-student:lab/lab1/Dockerfile	No known exploit	Not Auto Fixable	Dec 7, 2023	Snyk Container

Learning

- Understand the basics of Kubernetes for risks understanding
  - Certification CKA or bootcamp for your team
- Not only focus on Kubernetes but in all DevSecOps phase
  - https://www.practical-devsecops.com/
  - Snyk ambassador program
  - Cloud providers certifications
- Gamify your learning with hackaton by example
- Continuous learning (technologies change)
- Follow CNCF project to understand new phylosophie and avoid to lost time to develop custom solution already build by an opensource project
- Discuss with the community (Devoxx , Kubecon , france devops)
- Always have a critical eye on solutions

And of course, do not forget that **we also have a wide range of trainings** including :

Okta, Auth0, Kubernetes Security, Kubernetes User, Kubernetes Admin, Docker, Terraform, Ansible, Git, CI/CD, AWS, ArgoCD, DevOps, DevSecOps, Observability, SRE

Day2

Day 1 is often more simple because you don't need to manage upgrade , API deprecated , new logic ...

- Test you security in your code , for example implement Framework testing for OPA, Kyverno, networkpolicy because your rules can working to day but can drift after few month (evolution OPA , Kubernetes API)
- Pull logic (Gitops)
- Don't reinvent the wheel when possible
- Use the righ tool for the righ usage
- Avoid duplication of code , centralize configuration and automate when is possible
- Don't implement tech for tech

CNAPP - Open Source

In any organization, the decision to implement an open-source solution versus a commercial solution can be a matter of debate. There is no easy answer.

Pro CNAPP

- Easy to use (plug and play)
- Prebuild configurations
- Directs insight
- Compliant dashboard
- Generaly based on opensource product
- Focus on prioritization and contextualisation
- No need to have experts on your team
- More easy to manage and scale

Pro Opensource

- highly configurable and customizable
- Free
- Community driven
- Build custom solution based on opensource project