POLITECNICO DI TORINO

Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Analisi delle Policy di controllo dell'accesso in Kubernetes

Relatori

Prof. Fulvio VALENZA

Dott. Francesco PIZZATO

Candidato

Vincenzo FALCONE

Dicembre 2024

Alla mia famiglia

Abstract

Nel contesto del cloud computing, la gestione della sicurezza e dei controlli di accesso è fondamentale per proteggere le risorse e i dati aziendali. Kubernetes, un sistema di orchestrazione di container ampiamente adottato, gestisce ambienti complessi e distribuiti dove l'accesso sicuro alle risorse rappresenta una delle sfide principali. Con la crescente complessità e scalabilità delle applicazioni distribuite, è essenziale disporre di meccanismi avanzati e flessibili per il controllo degli accessi, che siano in grado di adattarsi dinamicamente ai requisiti di sicurezza.

La ricerca si basa su un'analisi approfondita delle modalità esistenti sul controllo degli accessi in Kubernetes, in particolare il Role-Based Access Control (RBAC). La metodologia prevede lo sviluppo di un nuovo strumento per automatizzare il processo di perfezionamento e verifica delle *policy*, sulla base di un duplice modello: *Specification Model* e *Implementation Model*.

L'obiettivo di questa tesi è analizzare e implementare metodologie automatizzate per il perfezionamento e la verifica delle politiche di accesso in Kubernetes, al fine di rilevare e correggere eventuali anomalie nella loro definizione. Una configurazione manuale e non strutturata delle *policy* di accesso può infatti esporre a vulnerabilità significative, come dimostrato da vari incidenti di sicurezza legati a configurazioni errate. Tali strumenti diventano quindi essenziali per supportare gli amministratori di rete nelle loro attività di gestione delle *policy* di accesso, migliorando al contempo la sicurezza dei sistemi Kubernetes.

Ringraziamenti

Desidero ringraziare tutti coloro che, in diversi modi, hanno contribuito al mio percorso, fornendomi il proprio supporto.

Indice

Elenco delle tabelle

Elenco delle figure

Acronimi

VM
Virtual Machine

CPU
Central Processing Unit

IP
Internet Protocol

K8s
Kubernetes

HTTP
Hypertext Transfer Protocol

API
Application Program Interface

DNS
Domain Name System

URL
Uniform Resource Locator

CRD
Custom Resource Definitions

OPA
Open Policy Agent

RBAC

Role-Based Access Control

ABAC

Attribute-Based Access Control

REST

Representational State Transfer

YAML

"Yet Another Markup Language"

JSON

JavaScript Object Notation

Capitolo 1

Introduzione

1.1 Contesto

Nel contesto del cloud computing, la sicurezza e il controllo degli accessi sono aspetti importanti per proteggere le risorse e i dati aziendali. Tuttavia, va sottolineato che l'adozione massiccia di architetture distribuite e multi-tenant da parte delle infrastrutture cloud aumenta la sfida nella configurazione delle funzionalità di sicurezza e, contemporaneamente, accresce il rischio di errori che potrebbero compromettere l'integrità dei dati. Kubernetes, la piattaforma di orchestrazione di container più diffusa, gestisce ambienti distribuiti complessi, portando con sé forti preoccupazioni per la sicurezza delle risorse e il controllo dell'accesso ai dati.

Le nuove esigenze delle applicazioni distribuite includono scalabilità, flessibilità e la capacità di affrontare dinamicamente condizioni variabili, con la disponibilità di risorse adattata in base alla domanda.

In un contesto così dinamico, sono necessari meccanismi avanzati di controllo degli accessi per garantire la sicurezza senza sacrificare la reattività e l'efficienza del sistema. Tuttavia, la gestione e configurazione delle policy in Kubernetes sono dettagliate e, se mal configurate, possono generare numerose vulnerabilità. La dinamicità delle minacce alla sicurezza e la crescente complessità dei sistemi operativi nelle infrastrutture cloud ora richiedono un approccio al controllo degli accessi flessibile, granulare e adattabile per rispondere in tempo reale ai requisiti di sicurezza dell'infrastruttura.

1.2 Struttura del documento

Questo lavoro di tesi è stato suddiviso nei seguenti capitoli:

- **Capitolo 1 - Introduzione**: in questo capitolo viene presentato il contesto generale della tesi, gli obiettivi del lavoro e il problema che si intende affrontare.
- **Capitolo 2 - Introduzione a Kubernetes**: questo capitolo analizza l'architettura di Kubernetes, con particolare attenzione ai componenti principali e alla definizione degli oggetti Kubernetes.
- **Capitolo 3 - Gestione dei permessi**: qui vengono presentati i moduli coinvolti nell'autenticazione e nell'autorizzazione di una richiesta. Viene inoltre fornita una panoramica dei principali modelli di controllo degli accessi, con particolare attenzione a RBAC. Il capitolo approfondisce le caratteristiche di questo modello, le sue applicazioni in Kubernetes e le sue limitazioni.
- **Capitolo 4 - Obiettivi della tesi**: questo capitolo introduce in modo formale ed esaustivo la problematica affrontata dalla tesi. In seguito viene introdotto un possibile approccio alla sua risoluzione. il linguaggio formalizzato per la definizione delle policy di accesso, descrivendo la sua struttura in formato JSON e spiegando come risponde alle esigenze di Kubernetes. Vengono illustrati esempi pratici per dimostrare l'applicabilità del linguaggio.
- **Capitolo 5 - Design e Implementazione del Sistema di Gestione delle Policy in Kubernetes**: il capitolo descrive l'architettura proposta per il sistema di gestione delle policy di accesso, includendo i moduli di generazione, traduzione e verifica delle policy. Vengono dettagliati il flusso di autorizzazione proposto, le scelte di design e la verifica della coerenza delle configurazioni RBAC. Vengono anche esplicitati 3 casi d'uso per comprendere meglio il processo di traduzione. Infine vengono elencate le fase comprendenti la verifica delle policy.
- **Capitolo 6 - Conlcusioni e Lavori Futuri**: qui vengono riassunti i principali risultati del lavoro. Successivamente vengono esplorate possibili direzioni di sviluppo futuro, come l'integrazione di modalità di controllo degli accessi più avanzate, il supporto per ambienti multi-cluster e l'automazione della correzione delle anomalie.

Capitolo 2

Introduzione a Kubernetes

2.1 Cos'è Kubernetes

Kubernetes è una piattaforma portatile, estensibile e open-source per la gestione di carichi di lavoro e servizi containerizzati in grado di facilitare sia la configurazione dichiarativa che l'automazione [1]. E' diventata uno standard de facto per orchestrare container, consentendo di gestire cluster di macchine su cui eseguire container Docker in modo efficiente. Per capire a fondo l'utilità di Kubernetes è necessario dare uno sguardo all'evoluzione del deployment.

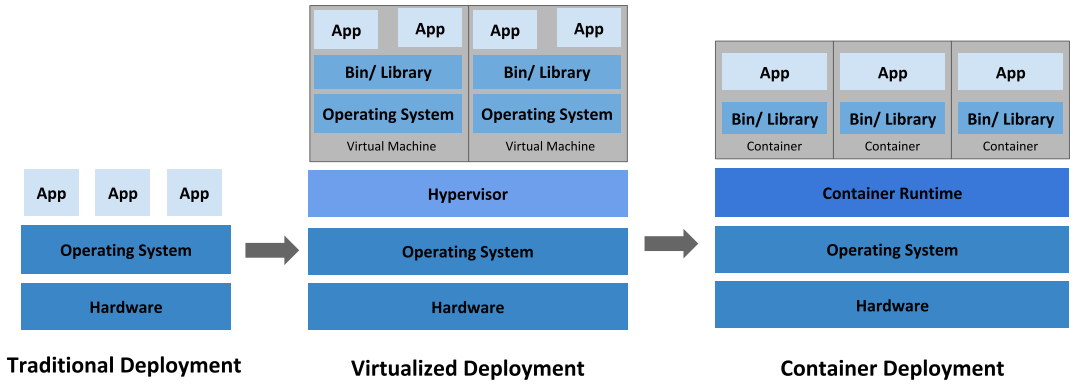

Figura 2.1: Deployment history [1].

Nel corso del tempo, le modalità di deployment delle applicazioni hanno subito un'evoluzione significativa, passando da server fisici dedicati, alla virtualizzazione[2], fino all'utilizzo dei container, come rappresentato nell'immagine sull'evoluzione del deployment disponibile sul sito di Kubernetes.

Inizialmente, le applicazioni venivano eseguite su server fisici dedicati, con conseguente inefficienza nell'uso delle risorse. Poiché un singolo server doveva essere riservato per ogni applicazione, le risorse disponibili venivano spesso sottoutilizzate, e i costi di acquisto e manutenzione dell'hardware erano molto elevati. Inoltre, l'assenza di isolamento tra le applicazioni portava a problemi di sovraccarico delle risorse: se un'applicazione assorbiva troppe risorse, le altre potevano subire rallentamenti significativi.

Con l'introduzione della virtualizzazione, si è riusciti a ottimizzare l'uso delle risorse. Le macchine virtuali (VM) permettevano di eseguire più applicazioni su un singolo server fisico, ognuna con il proprio sistema operativo virtuale isolato, migliorando la scalabilità e l'efficienza e riducendo i costi. Tuttavia, l'uso delle VM era comunque pesante in termini di risorse, poiché ciascuna VM richiedeva l'emulazione di un intero sistema operativo, occupando più memoria e CPU rispetto a quanto effettivamente necessario.

L'era dei container ha portato un'ulteriore evoluzione nel deployment delle applicazioni. I container, a differenza delle VM, condividono il sistema operativo sottostante, mantenendo però l'isolamento tra le applicazioni. Questo modello di deployment leggero riduce notevolmente l'uso delle risorse e rende i container estremamente portabili tra diversi ambienti, come differenti cloud o data center. I container sono meno invasivi in termini di risorse rispetto alle VM, offrendo un'efficienza maggiore in fase di esecuzione delle applicazioni e permettendo una gestione più semplice delle dipendenze. Grazie alla loro flessibilità e portabilità, i container hanno rivoluzionato il modo in cui vengono distribuite e gestite le applicazioni moderne, rendendo il deployment scalabile e più efficiente.

In questo contesto si inserisce Kubernetes, il cui framework garantisce l'esecuzione stabile e resiliente di sistemi distribuiti. Esso offre numerose funzionalità avanzate. Una delle più importanti è il *service discovery* e il bilanciamento del carico, che consente di esporre i container tramite un nome DNS o un indirizzo IP e distribuire il traffico su più container in caso di picchi, mantenendo così il servizio stabile. Inoltre, Kubernetes gestisce l'orchestrazione dello storage, permettendo di montare automaticamente diversi tipi di archiviazione, sia locale che *cloud*. L'ottimizzazione dei carichi di lavoro è anch'essa centrale: Kubernetes distribuisce i container sui nodi del cluster, massimizzando l'uso delle risorse disponibili in base alle esigenze di CPU e memoria. Un altro vantaggio cruciale è il *self-healing*, attraverso il quale Kubernetes rileva e risolve automaticamente i problemi, riavviando o sostituendo container non funzionanti e isolando quelli che non rispondono correttamente. Infine, Kubernetes gestisce con sicurezza i dati sensibili e la configurazione dell'applicazione, permettendo l'aggiornamento delle informazioni riservate senza

esporle nel sistema e senza necessità di ricostruire le immagini dei container.

2.2 Architettura di Kubernetes

Un cluster Kubernetes è un insieme di componenti interconnessi che lavorano in armonia per eseguire e gestire applicazioni containerizzate in modo resiliente e scalabile. Un cluster è formato da un Kubernetes Control Plane e almeno un nodo "worker". Il primo ha il compito di monitorare e gestire i nodi worker, i quali a loro volta sono le macchine responsabili di eseguire i container (o pod) che contengono i carichi di lavoro degli utenti. Nei sistemi di produzione, il control plane è solitamente distribuito su più nodi per assicurare maggiore disponibilità (high availability) e garantire il failover. Come descritto da diverse fonti, tra cui la documentazione ufficiale di Kubernetes, la sua architettura è formata dai seguenti componenti[1, 3, 4]:

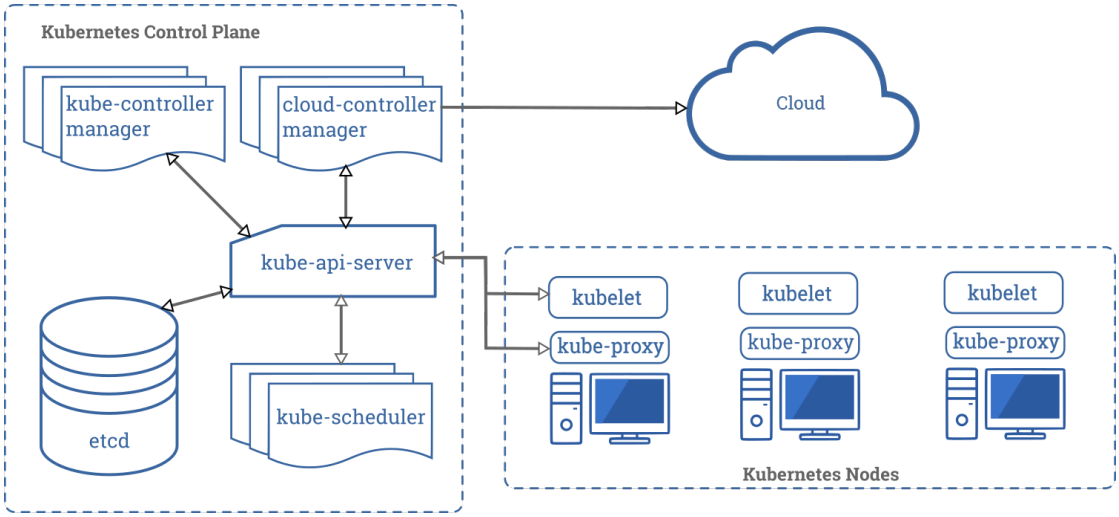

Figura 2.2: Components of Kubernetes [1].

2.2.1 Componenti del Control Plane

Il Control Plane (o master) è la componente principale di un cluster Kubernetes, poiché è responsabile del coordinamento tra i vari nodi. I nodi worker si affidano al Control Plane per gestire l'esecuzione dei container, ma quest'ultimo non esegue direttamente i container applicativi. La sua funzione principale è orchestrare le operazioni di gestione del cluster, come la distribuzione dei carichi di lavoro, il monitoraggio dello stato dei nodi e la gestione degli aggiornamenti. Il Control Plane è costituito da diversi componenti fondamentali, tra cui il kube-apiserver,

etcd, kube-scheduler, kube-controller-manager, e il cloud-controller-manager. Nei prossimi paragrafi esamineremo ciascun componente nel dettaglio.

Kube-apiserver

L'API server è uno dei componenti centrali del control plane di Kubernetes e rappresenta il punto di accesso principale per interagire con il cluster. Si tratta di un'interfaccia che gestisce tutte le comunicazioni interne ed esterne verso il sistema, fungendo da front end per l'intero Control Plane. In pratica, tutte le operazioni di gestione e coordinamento del cluster passano attraverso l'API server, il quale espone le API di Kubernetes utilizzando un'interfaccia REST. Questo permette agli amministratori e agli strumenti, come kubectl e kubeadm, di inviare comandi e richieste per monitorare, configurare o modificare lo stato del cluster.

Un aspetto fondamentale del kube-apiserver, è la sua capacità di scalare orizzontalmente. Questo significa che in ambienti ad alta disponibilità, possono essere eseguite più istanze di kube-apiserver, permettendo il bilanciamento del carico e garantendo la continuità operativa anche in presenza di un aumento delle richieste o in caso di guasti. Grazie a questa architettura, il sistema può gestire in modo efficiente cluster di grandi dimensioni, mantenendo sempre la capacità di elaborare le richieste e garantire la resilienza del cluster.

Etcd

Etcd è un componente essenziale all'interno dell'architettura di Kubernetes, responsabile della memorizzazione dello stato del cluster in un database chiave-valore. Questo sistema di archiviazione distribuito è altamente affidabile, progettato per garantire la tolleranza ai guasti e la consistenza dei dati, anche in ambienti complessi e distribuiti. Ogni dato relativo alla configurazione del cluster, alle risorse disponibili e allo stato corrente viene salvato in etcd, rendendolo l'unica fonte di verità per Kubernetes.

Poiché i componenti del control plane sono senza stato (stateless), essi si appoggiano a etcd per accedere alle informazioni necessarie al funzionamento del cluster. Questi componenti, come l'API server, interrogano etcd per recuperare lo stato corrente del sistema e garantire che tutto funzioni correttamente. Anche per questo componente, data la sua criticità, in configurazioni ad alta disponibilità, etcd viene replicato su più nodi per evitare che un singolo punto di guasto comprometta l'intero sistema.

Dovendo memorizzare informazioni fondamentali sul funzionamento del cluster, è essenziale implementare una strategia di backup adeguata per etcd, in modo da prevenire la perdita di dati in caso di problemi o malfunzionamenti del sistema.

La sua importanza rende quindi etcd un elemento centrale per la stabilità e la continuità operativa del cluster Kubernetes.

Kube-scheduler

Il Kubernetes Scheduler è un altro componente chiave del Control Plane, il cui compito principale è quello di allocare in modo efficiente le risorse del cluster e di garantire che i pod appena creati vengano assegnati ai nodi di lavoro più appropriati. Il suo ruolo è essenziale per mantenere l'integrità del cluster e per ottimizzare l'utilizzo delle risorse come CPU e memoria. Lo scheduler valuta le richieste di risorse del pod e confronta queste richieste con le risorse disponibili sui nodi worker del cluster.

Quando un nuovo pod viene creato e non ha ancora un nodo assegnato, lo scheduler esamina diversi fattori per decidere dove eseguirlo. Questi fattori includono le specifiche del pod stesso, come la CPU e la memoria richieste, le risorse già impegnate da altri carichi di lavoro presenti nel sistema, e vincoli di tipo hardware o software. Inoltre, lo scheduler considera vincoli relativi alle politiche di affinità e anti-affinità, che possono influenzare la collocazione di pod su nodi specifici per migliorare le performance o ridurre i conflitti tra carichi di lavoro.

Un altro elemento cruciale preso in considerazione dallo scheduler è la disponibilità di volumi di dati necessari per l'esecuzione del pod, oltre a eventuali interferenze tra workload già presenti. Pertanto, lo scheduler garantisce che ogni pod venga eseguito nel nodo che offre il miglior equilibrio tra disponibilità di risorse e conformità ai requisiti del sistema, mantenendo il cluster in uno stato efficiente e operativo.

Kube-controller-manager

Il controller manager è uno dei componenti cruciali del control plane di Kubernetes, e il suo compito principale è assicurare che lo stato attuale del cluster coincida con lo stato desiderato, attraverso un processo continuo chiamato reconciliation loop. In pratica, il controller manager monitora costantemente il cluster e, se rileva che qualcosa non è come dovrebbe essere (ad esempio, un pod che smette di funzionare), attiva i vari controller per correggere la situazione e riportare il sistema allo stato ideale.

I controller gestiti dal kube-controller-manager si occupano di varie funzioni all'interno del cluster. Ad esempio, il node controller monitora la disponibilità dei nodi e gestisce le azioni da intraprendere quando un nodo diventa inaccessibile. Il replication controller assicura che il numero di pod per ogni ReplicaSet sia corretto,

garantendo che il cluster disponga del numero necessario di pod attivi. Il controller degli endpoints collega i pod ai servizi, permettendo che le richieste vengano instradate correttamente ai punti finali del sistema. Infine, il service account controller crea account e token di accesso alle API per ciascun nuovo namespace.

Dal punto di vista organizzativo, anche se ogni controller potrebbe funzionare come un processo separato, per semplificare la gestione, Kubernetes raggruppa tutti i controller principali all'interno di un unico container ed esegue le loro funzioni in un singolo processo.

Cloud-controller-manager

Il cloud-controller-manager è un componente fondamentale del Control Plane di Kubernetes, progettato per gestire l'integrazione tra il cluster Kubernetes e le risorse fornite da un cloud provider. Questo modulo consente al cluster di interagire direttamente con le API del provider di cloud, separando logicamente le componenti che devono interagire con l'infrastruttura cloud dalle altre parti che gestiscono il cluster stesso. Questo approccio offre maggiore flessibilità e modularità nella gestione del cluster, in quanto il cloud-controller-manager può essere adattato o sostituito senza influenzare gli altri componenti.

Il cloud-controller-manager esegue controller specifici per il cloud provider, responsabili di attività legate alla gestione dei nodi, della rete e dei servizi del cloud. Per esempio, il node controller verifica se un nodo del cloud è stato rimosso, mentre il route controller configura le rotte di rete nell'infrastruttura cloud sottostante. Il service controller, infine, si occupa della creazione, aggiornamento e eliminazione dei load balancer nel cloud.

In contesti come ambienti on-premise o cluster locali, non è necessario utilizzare un cloud-controller-manager, poiché non esiste alcuna interazione con servizi cloud. Tuttavia, in ambienti distribuiti su cloud, questo componente diventa essenziale per collegare il cluster Kubernetes alle risorse offerte dal provider.

Analogamente al kube-controller-manager, il cloud-controller-manager combina più control loop in un unico processo per ridurre la complessità. Questo processo può essere scalato orizzontalmente per migliorare la tolleranza ai guasti e garantire una maggiore reattività del sistema. Grazie a questa struttura, Kubernetes è in grado di gestire in modo efficiente l'interazione con i servizi cloud, garantendo la continuità operativa e una facile scalabilità.

2.2.2 Componenti dei nodi

Un nodo in Kubernetes, spesso chiamato worker node, è una delle componenti essenziali per eseguire i carichi di lavoro all'interno del cluster. Un cluster Kubernetes può avere uno o più nodi, a seconda delle necessità operative. I pod, che contengono i container delle applicazioni, vengono eseguiti su questi nodi. Quando è necessario aumentare la capacità del cluster, è sufficiente aggiungere ulteriori nodi, migliorando così la capacità di elaborazione e la scalabilità del sistema.

Ogni nodo è un'entità autonoma e ha un container runtime installato, come Docker o rkt, che consente l'esecuzione dei container. I nodi sono gestiti dal Control Plane, ma anche in caso di perdita temporanea di comunicazione con il master, un nodo può mantenere attivi i pod già in esecuzione, garantendo una certa continuità operativa. Questo è possibile grazie all'architettura decentralizzata del cluster Kubernetes, che distribuisce il carico di lavoro in modo efficiente tra i nodi.

I componenti principali eseguiti su ogni nodo includono il kubelet, che assicura che i pod siano in esecuzione e funzionino correttamente, e il kube-proxy, che gestisce il networking tra i pod e il resto del cluster. Questa struttura distribuita consente a Kubernetes di orchestrare i carichi di lavoro su più nodi, mantenendo alta disponibilità e flessibilità nella gestione delle risorse del cluster.

Kubelet

Ogni nodo all'interno di un cluster Kubernetes include un componente fondamentale chiamato kubelet. Questo piccolo agente è responsabile di garantire che i container siano eseguiti correttamente all'interno dei pod. Quando il Control Plane del cluster deve eseguire un'operazione su un nodo, si affida alla kubelet per gestire questa azione. In pratica, la kubelet riceve istruzioni, o PodSpecs, che specificano come devono essere eseguiti i container, e si assicura che vengano rispettati.

Il ruolo della kubelet è quindi quello di mantenere una comunicazione continua con il master node e di assicurarsi che lo stato del nodo sia sempre allineato con lo stato desiderato dal sistema, ossia il cosiddetto desired state. Ad esempio, se un container smette di funzionare o viene richiesto un cambiamento, la kubelet interviene per correggere la situazione e garantire che tutto funzioni come dovrebbe.

Un aspetto interessante della kubelet è che non gestisce i container che non sono stati creati direttamente da Kubernetes. Questo significa che la sua responsabilità si limita ai container definiti tramite il sistema Kubernetes, assicurandosi che siano sempre attivi e in salute all'interno del pod.

Kube-proxy

Il kube-proxy funge da proxy di rete. Il suo compito principale è gestire il traffico di rete all'interno del cluster, sia tra i nodi che verso l'esterno. Questo componente è essenziale per garantire che le comunicazioni tra i vari pod e servizi avvengano in modo corretto, configurando le regole di rete e assicurando che i pacchetti di dati siano instradati correttamente.

Il kube-proxy utilizza, quando possibile, le funzionalità di rete del sistema operativo per ottimizzare il filtraggio dei pacchetti. In caso contrario, gestisce il traffico direttamente, permettendo comunque il corretto instradamento delle richieste. In sostanza, il kube-proxy si occupa di tutto ciò che riguarda la rete: dall'accesso ai service Kubernetes fino alla comunicazione tra i nodi del cluster e verso l'esterno.

Grazie al kube-proxy, la gestione delle comunicazioni di rete diventa trasparente, poiché si occupa automaticamente di creare le regole necessarie per far sì che i servizi siano sempre raggiungibili, indipendentemente da dove si trovano i pod all'interno del cluster. In questo modo, i Kubernetes Services possono distribuire il traffico in modo efficiente e affidabile, senza che gli utenti o le applicazioni debbano preoccuparsi della complessità del networking sottostante.

2.3 Oggetti Kubernetes

In Kubernetes, le risorse (o oggetti) rappresentano delle unità che descrivono lo stato desiderato del cluster. Quando un amministratore crea una risorsa, sta definendo come dovrebbe essere configurato il sistema, come ad esempio quali applicazioni containerizzate devono essere eseguite e su quali nodi, quali risorse devono essere allocate, o quali policy devono essere applicate per gestire l'esecuzione e la resilienza delle applicazioni. Una volta che queste specifiche sono impostate, Kubernetes lavora continuamente per mantenere questo stato desiderato, cercando di far coincidere la configurazione reale con quella definita.

Un modo semplice per capire le risorse di Kubernetes è pensare a loro come a delle "istanze" di un concetto più ampio. Ogni risorsa è un esempio di un tipo specifico, come un pod, un deployment o un servizio, che segue una struttura predefinita. Questa struttura indica al kube-apiserver come gestire le risorse e comunicare con gli altri componenti del cluster.

Kubernetes utilizza le risorse per mantenere in modo persistente lo stato del cluster. Queste entità definiscono non solo quali container devono essere eseguiti, ma anche come gestire gli aggiornamenti, i riavvii automatici e la tolleranza agli

errori. L'idea alla base delle risorse è quella di rappresentare un "record di intenti": una volta creata una risorsa, Kubernetes si assicura che essa sia sempre conforme a quanto definito dall'amministratore. Tutte le risorse in Kubernetes hanno la stesso formato. Sono rappresentati nell'API mediante il formato `.yaml`. Eccone un esempio:

Listing 2.1: Esempio generico di un oggetto Kubernetes

```yaml
apiVersion: VERSIONE_API
kind: TIPO
metadata:
  name: NOME
  labels:
    chiave1: valore1
    chiave2: valore2
spec:
  replicas: NUMERO_REPLICHE
  selector:
    matchLabels:
      chiave_selector: valore_selector
  template:
    metadata:
      labels:
        chiave_pod: valore_pod
    spec:
      containers:
      - name: NOME_CONTAINER
        image: IMMAGINE_CONTAINER:VERSIONE
        ports:
        - containerPort: PORTA
```

L'esempio presentato mostra la configurazione generica di un oggetto Kubernetes utilizzando il formato `YAML`. Di seguito sono descritti i campi principali:

- **apiVersion**: questo campo specifica la versione dell'API Kubernetes da utilizzare per l'oggetto. La versione può variare a seconda del tipo di risorsa e delle sue specifiche (ad esempio, `apps/v1`).
- **kind**: indica il tipo di oggetto che si vuole creare o gestire nel cluster. Alcuni esempi comuni includono `Deployment`, `Service` e `Pod`.
- **metadata**: contiene informazioni descrittive sull'oggetto. Include:
  - **name**: il nome dell'oggetto, utilizzato per identificarlo univocamente all'interno del namespace.
  - **labels**: una serie di etichette (coppie `chiave: valore`) utilizzate per classificare e raggruppare le risorse. Le etichette facilitano la selezione delle risorse per scopi di gestione e monitoraggio.

- **spec**: descrive lo stato desiderato dell'oggetto, cioè come dovrebbe essere configurato e comportarsi nel cluster. Questo campo può includere:
  - **replicas**: specifica il numero di istanze del pod che devono essere eseguite. Ad esempio, impostando `replicas: 3`, Kubernetes manterrà sempre tre repliche del pod in esecuzione.
  - **template**: definisce il modello dei pod da creare, specificando la configurazione dei container al suo interno.
    - **metadata**: le etichette da applicare ai pod generati.
    - **containers**: una lista di container che devono essere eseguiti all'interno di ciascun pod. Include:
      - **name**: il nome del container.
      - **image**: l'immagine del container da utilizzare, spesso specificata con il nome dell'immagine e la versione (ad esempio, `nginx:1.14.2`).
      - **ports**: la porta su cui il container è in ascolto, utile per esporre i servizi del container.

Questa configurazione permette a Kubernetes di gestire automaticamente lo stato desiderato dell'applicazione. Ad esempio, se un pod viene eliminato o smette di funzionare, Kubernetes utilizzerà le informazioni presenti nel `spec` per creare un nuovo pod e ripristinare il numero di repliche configurato.

2.3.1 Pods

Un pod è l'elemento di base per l'esecuzione delle applicazioni all'interno di un cluster Kubernetes. Esso raggruppa uno o più container che condividono risorse come rete e storage e che vengono eseguiti insieme sullo stesso nodo. Questo permette ai container all'interno di un pod di comunicare tra loro facilmente e di operare in un contesto comune. Grazie ai pod, Kubernetes può gestire e spostare le applicazioni tra diversi nodi, facilitando la scalabilità orizzontale, soprattutto per applicazioni costruite con un'architettura a microservizi.

I pod sono pensati per essere entità temporanee: possono essere eliminati e ricreati in caso di malfunzionamenti o di modifiche nel carico di lavoro. Se un pod o il nodo su cui viene eseguito smette di funzionare, Kubernetes crea automaticamente una nuova istanza del pod su un altro nodo disponibile, garantendo la continuità operativa dell'applicazione.

Come mostrato in figura 2.3, i pod raggruppano i container all'interno di ciascun nodo del cluster: nella maggior parte dei casi ne contiene solo uno, ma in scenari più avanzati può includere più container che lavorano strettamente insieme.

Inoltre, i pod possono ospitare dei init containers, che vengono eseguiti prima dei container principali per preparare l'ambiente, o ephemeral containers, che possono essere utilizzati temporaneamente per operazioni di debugging.

I pod agiscono come un tramite tra i container e l'infrastruttura di Kubernetes. Offrono ai container l'ambiente necessario per funzionare, gestendo l'accesso alle risorse di rete, lo storage, e altre configurazioni richieste per l'esecuzione. Mentre un container racchiude tutto ciò che serve per eseguire un'applicazione, un pod crea un "contenitore" che permette a Kubernetes di gestire facilmente i container stessi, garantendo che abbiano tutto ciò di cui hanno bisogno per operare in modo efficiente nel cluster.

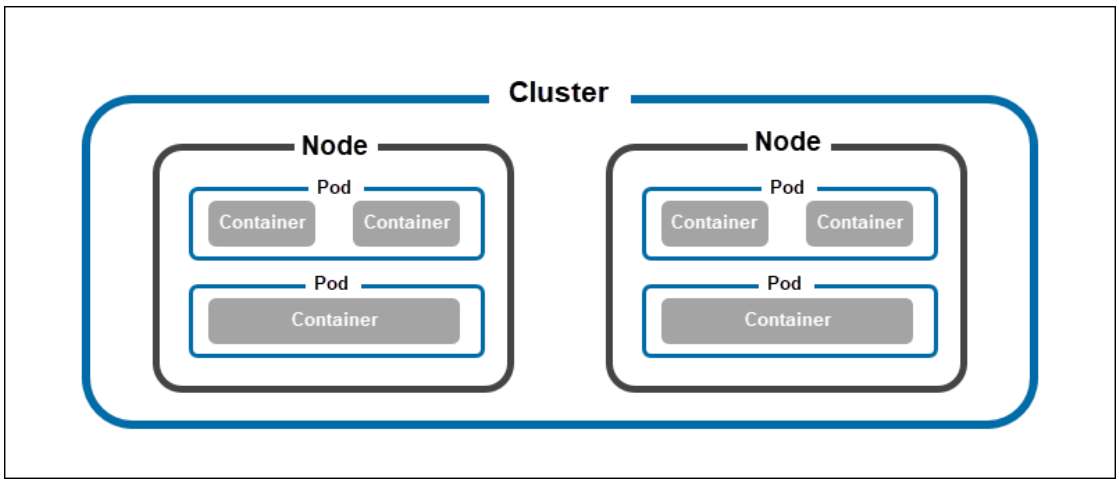

Figura 2.3: Cluster Kubernetes con nodi e pod [5].

2.3.2 Service

Come specificato nella documentazione ufficiale di Kubernetes [6] un Service è un'astrazione che consente di esporre un gruppo di pod come un unico punto di accesso di rete, facilitando la comunicazione sia interna che esterna al cluster. Questo è utile perché, nel contesto di un cluster, i pod hanno un ciclo di vita dinamico: possono essere creati, distrutti e ricreati su nodi diversi, cambiando il loro indirizzo IP. Senza un service, la gestione degli indirizzi IP di ogni singolo pod diventerebbe molto complessa.

Un Service assegna un indirizzo IP stabile e un nome DNS, rendendo più semplice l'accesso ai pod associati. Esistono diversi tipi di service in base all'esposizione della rete:

- ClusterIP: è il tipo predefinito e rende il servizio accessibile solo all'interno del cluster. Questo è utile per la comunicazione interna tra i vari componenti.
- NodePort: consente di esporre un servizio su una porta specifica di ogni nodo, rendendolo accessibile dall'esterno del cluster.
- LoadBalancer: integra un bilanciamento del carico (load balancing) e rende il servizio accessibile tramite un singolo IP esterno. È spesso utilizzato nei cloud provider per bilanciare il traffico in ingresso tra i pod.

Il Service utilizza un sistema di etichette per associare i pod al servizio stesso. Grazie alle etichette, un service può monitorare dinamicamente i pod che devono essere inclusi nel gruppo di backend, permettendo al traffico di essere reindirizzato anche quando i pod cambiano.

2.3.3 Deployment

In un contesto reale i pod non sono creati manualmente, piuttosto vengono definiti oggetti di alto livello, come i Deployment che si occupano di gestire automaticamente la creazione e il ciclo di vita dei pod. Un Kubernetes Deployment è un tipo di risorsa che consente di gestire aggiornamenti alle applicazioni in modo dichiarativo. Con un deployment, è possibile definire come deve essere gestita la vita di un'applicazione, specificando, ad esempio, quali immagini container utilizzare, quanti pod devono essere presenti e come devono essere eseguiti gli aggiornamenti.

Quando si crea un oggetto in Kubernetes, si forniscono indicazioni su come deve essere configurato il carico di lavoro nel cluster. Kubernetes si occupa poi di mantenere tale configurazione, garantendo che lo stato desiderato venga sempre rispettato.

Aggiornare manualmente le applicazioni containerizzate può essere complesso e richiedere molto tempo. Per passare a una nuova versione, si devono avviare i nuovi pod, arrestare quelli vecchi, verificare che i nuovi siano operativi e, in caso di problemi, ripristinare la versione precedente. Questo processo può comportare errori umani e richiedere un notevole sforzo per automatizzare correttamente ogni fase.

Un Kubernetes Deployment automatizza e rende ripetibile questo processo. Essi garantiscono che il numero desiderato di pod sia sempre attivo e disponibile. Inoltre, ogni aggiornamento è tracciato e versionato, permettendo di sospendere, continuare o ripristinare a una versione precedente, se necessario. Consentono quindi, di scalare rapidamente il numero di repliche dei pod e mantenere lo stato desiderato del cluster. Tutto questo avviene in modo automatizzato senza che l'utente debba intervenire manualmente.

Questo esempio illustra come un Deployment gestisca le repliche dei pod, specificando l'immagine del container, il numero di repliche desiderate e altre configurazioni.

Listing 2.2: Esempio di Deployment in Kubernetes

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: NOME_DEPLOYMENT
  labels:
    app: NOME_APP
spec:
  replicas: NUMERO_REPLICHE
  selector:
    matchLabels:
      app: NOME_APP
  template:
    metadata:
      labels:
        app: NOME_APP
    spec:
      containers:
      - name: NOME_CONTAINER
        image: IMMAGINE_CONTAINER:TAG
        ports:
        - containerPort: PORTA_CONTAINER
```

Capitolo 3

Gestione dei permessi in Kubernetes

3.1 Controllo degli accessi in Kubernetes

Il controllo degli accessi è un meccanismo fondamentale che garantisce la sicurezza delle risorse all'interno di un cluster. Quando una richiesta da parte di utenti o servizi raggiunge l'API Server attraversa diverse fasi illustrate nella seguente figura, tratta dal sito ufficiale di Kubernetes [7].

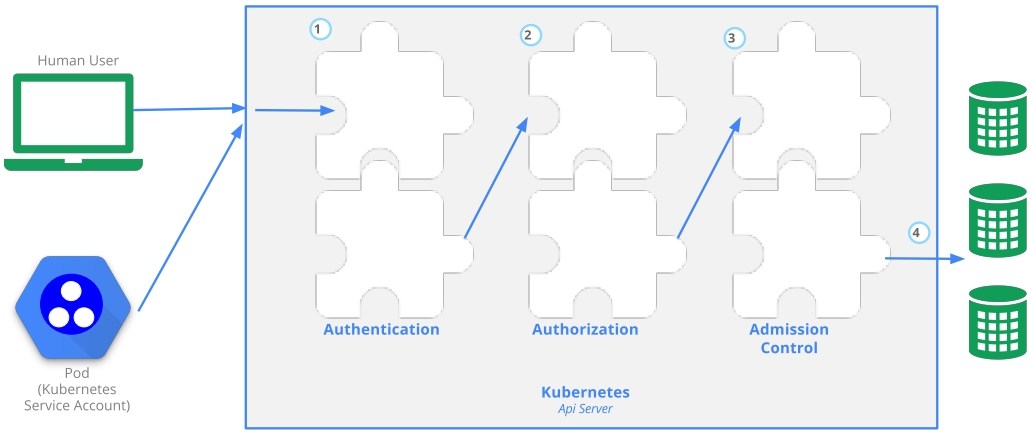

Figura 3.1: Overview del controllo degli accessi in Kubernetes[7].

La prima fase consiste nel verificare l'identità di chi invia la richiesta. L'autenticazione è basata su uno o più moduli *Authenticator*. Essi includono certificati client, password, token etc. Nel caso fossero specificati più moduli, ognuno di questi viene provato in sequenza, finchè uno di questi non autorizza la richiesta. Se l'autorizzazione non viene concessa, viene rifiutata con uno stato di errore.

Dopo aver autenticato l'utente, l'API Server valuta se esso ha i permessi necessari per eseguire l'azione richiesta su una specifica risorsa. La documentazione ufficiale di Kubernetes sottolinea l'importanza di includere nelle richieste il nome dell'utente, l'azione desiderata e l'oggetto interessato. Questo approccio garantisce che le autorizzazioni siano assegnate correttamente in base alle policy esistenti.[7]. Come l'autenticazione anche l'autorizzazione ammette più moduli di autorizzazione. Se sono presenti più moduli vengono controllati secondo la logica *OR*, se almeno un modulo autorizza la richiesta allora può procedere, se tutti i moduli rifiutano la richiesta, allora viene rifiutata.

La terza fase, il controllo di ammissione ha lo scopo di modificare la richiesta (es. aggiungendo informazioni) o validarla. Superate tutte le fasi, la richiesta viene applicata al cluster.

I prossimi paragrafi analizzeranno in dettaglio i moduli utilizzati per la gestione dell'autorizzazione in Kubernetes, con particolare attenzione a RBAC, ABAC, Node Authorizer, Kyverno, OPA e WebHook.

3.2 RBAC (Role-Based Access Control)

Il controllo degli accessi basato sui ruoli è un metodo per regolamentare l'accesso alle risorse in base ai ruoli assegnati agli utenti. Kubernetes RBAC utilizza il gruppo API *rbac.authorization.k8s.io*, abilitati attraverso il flag–*authorization-mode=RBAC*, per guidare le decisioni di autorizzazione [8]. Con la crescente complessità degli ambienti cloud, RBAC è anche diventato lo standard per la gestione delle policy di accesso.

Componenti e funzionamento

Questa modalità di controllo degli accessi si basa su quattro componenti principali: **Role, ClusterRole, RoleBinding, ClusterRoleBinding**.

Role e RoleBinding

Il **Role** definisce l'insieme dei permessi che possono essere assegnati agli utenti, specificano cosa può fare, su quali risorse all'interno del cluster (come pod, servizi o nodi) e in quale ambito. I permessi sono additivi, pertanto, non esistono regole per negare esplicitamente l'accesso ed inoltre, sono limitati al solo namespace in cui il Role viene creato. Questo semplifica la gestione dei permessi, in quanto basta assegnare all'utente il minimo necessario per svolgere i suoi compiti.

Il **RoleBinding** associa i permessi definiti in un Role ad uno o più soggetti (utenti, gruppi o ServiceAccount) all'interno di un namespace specifico.

Per esempio, supponiamo che il team di sviluppo necessiti di visualizzare lo stato dei pod nel namespace *test*, il Role chiamato *pod-reader* potrebbe essere configurato come segue:

Listing 3.1: Esempio Role

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: test
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
```

Questo Role permette solo la lettura dei pod e limita le azioni agli utenti che devono semplicemente monitorare le risorse.

Mentre se volessimo configurare il RoleBinding in modo che l'utente *Jane* ottenga i permessi definiti nel Role *pod-reader* nel solo namespace *test*, allora scriveremmo:

Listing 3.2: Esempio RoleBinding

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: example-rolebinding
  namespace: test
subjects:
- kind: User
  name: Jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
```

```
13   apiGroup: rbac.authorization.k8s.io
```

ClusterRole e ClusterRoleBinding

Come il Role anche Il **ClusterRole** definisce un insieme di permessi che possono essere assegnati agli utenti, ma a differenza loro specificano le azioni consentite su risorse a livello di cluster o su risorse in tutti i namespace. Quindi, i ClusterRole non sono limitati a un solo namespace e possono includere permessi globali. Questo li rende particolarmente utili per ruoli di amministrazione o di sistema che devono interagire con risorse dell'intero cluster.

Un ClusterRole può essere utilizzato sia a livello di cluster (ad esempio per accedere a risorse globali come i nodi) sia, tramite un RoleBinding, per assegnare permessi limitati a un singolo namespace.

Il **ClusterRoleBinding** è utilizzato per associare un ClusterRole a uno o più soggetti su tutto il cluster, permettendo l'accesso a tutte le risorse definite nel ClusterRole.

Se, ad esempio, un team ha bisogno di accedere in lettura ai nodi su tutto il cluster, il ClusterRole chiamato *node-reader* potrebbe avere la seguente configurazione:

Listing 3.3: Esempio ClusterRole

```
1 apiVersion: rbac.authorization.k8s.io/v1
2 kind: ClusterRole
3 metadata:
4   name: node-reader
5 rules:
6 - apiGroups: [""]
7   resources: ["nodes"]
8   verbs: ["get", "list", "watch"]
```

Questo ClusterRole consente solo la lettura delle risorse 'nodes' e permette agli utenti o alle applicazioni di monitorare i nodi del cluster senza poterli modificare.

Per assegnare i permessi definiti nel ClusterRole *node-reader* all'utente *admin* su tutto il cluster, possiamo creare un ClusterRoleBinding come segue:

Listing 3.4: Esempio ClusterRoleBinding

```
1 apiVersion: rbac.authorization.k8s.io/v1
2 kind: ClusterRoleBinding
3 metadata:
4   name: node-reader-binding
5 subjects:
6 - kind: User
```

```
7     name: admin
8     apiGroup: rbac.authorization.k8s.io
9   roleRef:
10    kind: ClusterRole
11    name: node-reader
12    apiGroup: rbac.authorization.k8s.io
```

Limitazioni di RBAC

Sebbene RBAC offra un controllo granulare e sia ben integrato in Kubernetes, presenta alcune limitazioni:

- **Complesso in ambienti molto grandi:** la gestione delle policy può diventare molto complicata in ambienti complessi, come quelli con migliaia di utenti e ruoli.
- **Nessuna condizione dinamica:** non è possibile creare regole di accesso che si basano su condizioni dinamiche come l'orario del giorno o la posizione geografica.
- **Nessuna negazione esplicita:** RBAC non supporta la definizione esplicita di azioni negate, quindi non è possibile definire esplicitamente un'azione vietata.

Le decisioni di accesso basate su condizioni temporali o attributi dinamici non sono supportate da RBAC, il che ne limita la flessibilità. Assenza di Regole di Negazione: Non esiste un meccanismo esplicito per negare l'accesso; tutte le policy sono additive.

3.3 ABAC (Attribute-Based Access Control)

ABAC è un metodo di autorizzazione in cui le decisioni di accesso si basano su attributi legati a utenti, risorse, azioni e contesto. A differenza del controllo degli accessi basato su ruoli, che assegna permessi statici utilizzando solo i ruoli, ABAC consente una gestione più dinamica e dettagliata dei permessi. Le prime versioni di Kubernetes supportavano ABAC, ma questa modalità è stata deprecata perché difficile da gestire e poco utilizzata [9].

Componenti e funzionamento

ABAC consente agli amministratori di definire policy di accesso utilizzando un file JSON che specifica le condizioni basate su attributi per concedere o negare l'accesso. Gli attributi considerati includono:

- **Attributi dell'utente:** nome utente, gruppi a cui l'utente appartiene.
- **Attributi della risorsa:** tipo di risorsa (es. pod, servizi).
- **Attributi ambientali:** condizioni ambientali come ora, posizione, IP di origine della richiesta.

Questo esempio mostra una policy di accesso basata su più condizioni. L'utente *admin* può accedere alle risorse in sola lettura soltanto se vengono soddisfatte determinate condizioni: l'accesso è consentito solo dalle 09:00 alle 17:00, l'utente si trova negli Stati Uniti e la risorsa appartiene all'ambiente di produzione (*env: production*). Questo tipo di policy evidenzia la flessibilità di ABAC rispetto ad esempio a RBAC, consentendo un controllo molto più granulare.

Listing 3.5: Esempio di Policy ABAC avanzata

```
{
  "apiVersion": "abac.authorization.kubernetes.io/v1beta1",
  "kind": "Policy",
  "spec": [
    {
      "user": "admin",
      "namespace": "*",
      "resource": "*",
      "readonly": true,
      "conditions": {
        "timeOfDay": {
          "start": "09:00",
          "end": "17:00"
        },
```

```
        "location": {
          "country": "US"
        },
        "resourceAttributes": {
          "labels": {
            "env": "production"
          }
        }
      }
    }
  ]
}
```

Limitazioni di ABAC

Nonostante i suoi vantaggi in termini di flessibilità, ABAC presenta alcune limitazioni significative:

- **Complesso da configurare:** le policy devono essere definite e mantenute manualmente, il che può diventare laborioso e soggetto a errori.
- **Gestione complessa:** la gestione delle policy ABAC può diventare molto complessa, specialmente in ambienti con molti attributi e condizioni.

3.4 Node Authorizer

Il Node Authorizer è una modalità di autorizzazione sviluppata per Kubernetes, progettata per migliorare la sicurezza dei nodi nel cluster. Consente ai nodi di accedere solo alle risorse necessarie per eseguire i pod assegnati. Questa funzionalità è stata introdotta per ridurre la superficie di attacco di un nodo compromesso, impedendo l'accesso non autorizzato alle risorse di altri nodi o a risorse non necessarie. Il Node Authorizer lavora in collaborazione con il NodeRestriction Admission Controller, garantendo che i nodi possano eseguire solo operazioni autorizzate sulle risorse critiche del cluster [10].

Componenti e funzionamento

Il Node Authorizer opera attraverso due componenti principali:

- **Nodi Kubernetes:** sono i nodi che eseguono i carichi di lavoro nel cluster. Ogni nodo possiede un'identità unica che determina a quali risorse può accedere.

- **API Server:** l'API Server riceve tutte le richieste dai nodi e applica le politiche del Node Authorizer per decidere se una richiesta deve essere approvata o rifiutata.

Questo meccanismo di autorizzazione si basa su regole predefinite che stabiliscono a quali risorse un nodo può accedere e quali operazioni può eseguire. In pratica, ogni nodo ha accesso solo ai pod assegnati e alle risorse correlate, come Secret e ConfigMap, necessari per il loro funzionamento.

Le restrizioni si applicano anche a livello di namespace: un nodo può interagire solo con le risorse del namespace che ospita i suoi pod. Questa segregazione riduce il rischio di accesso non autorizzato a risorse di altri namespace.

Inoltre, i nodi sono solitamente limitati a operazioni di sola lettura su risorse globali e non possono eseguire operazioni che modificano risorse critiche del cluster. Quindi viene aasicurato che i nodi non possano alterare lo stato del cluster o compromettere le operazioni di altri nodi.

Limitazioni di Node Authorizer

Il Node Authorizer presenta alcune limitazioni significative che possono influire sulla flessibilità e l'efficacia in determinati scenari:

- **Applicabile solo ai nodi:** non si applica agli utenti o account di servizio, è specifico per le richieste provenienti dai nodi stessi.
- **Permessi limitati:** il Node Authorizer gestisce esclusivamente i permessi relativi all'accesso alle risorse solo da parte dei nodi.

3.5 Kyverno

Kyverno è un motore di gestione delle policy nativo per Kubernetes, progettato per facilitare la definizione, l'applicazione e il controllo delle policy di sicurezza.

Una delle caratteristiche chiave di Kyverno è l'uso delle Custom Resource Definitions (CRD). Le CRD sono un meccanismo che consente di estendere l'API di Kubernetes, permettendo la creazione di nuovi tipi di risorse personalizzate oltre a quelle standard come Pod, Service o Deployment. Kyverno utilizza le CRD per definire le Policy CRD, permettendo di gestire le policy come risorse native di Kubernetes. Questo approccio rende la gestione delle policy coerente con il

resto dell'infrastruttura e permette agli amministratori di utilizzare strumenti come *kubectl*.

Componenti e funzionamento

Un componente chiave in Kyverno sono le Policy CRD, ogni policy è una risorsa Kubernetes che può essere applicata a livello di cluster o namespace. Possono essere di tre tipi:

- **Mutating:** queste policy modificano automaticamente le risorse per far sì che rispettino le configurazioni desiderate.
- **Validating:** queste policy sono regole che controllano le risorse prima che vengano accettate nel cluster.
- **Generating:** queste policy automatizzano la creazione di risorse o configurazioni necessarie.

La figura 3.2 mostra il flusso di una richiesta. Kyverno intercetta le richieste API inviate al server Kubernetes. Ogni richiesta viene confrontata con le policy definite nel cluster. A seconda del tipo di policy, Kyverno può: modificare la richiesta per conformarsi alle regole (Mutating), Verificare richieste non conformi (Validating) oppure generare risorse aggiuntive per automatizzare configurazioni (Generating).

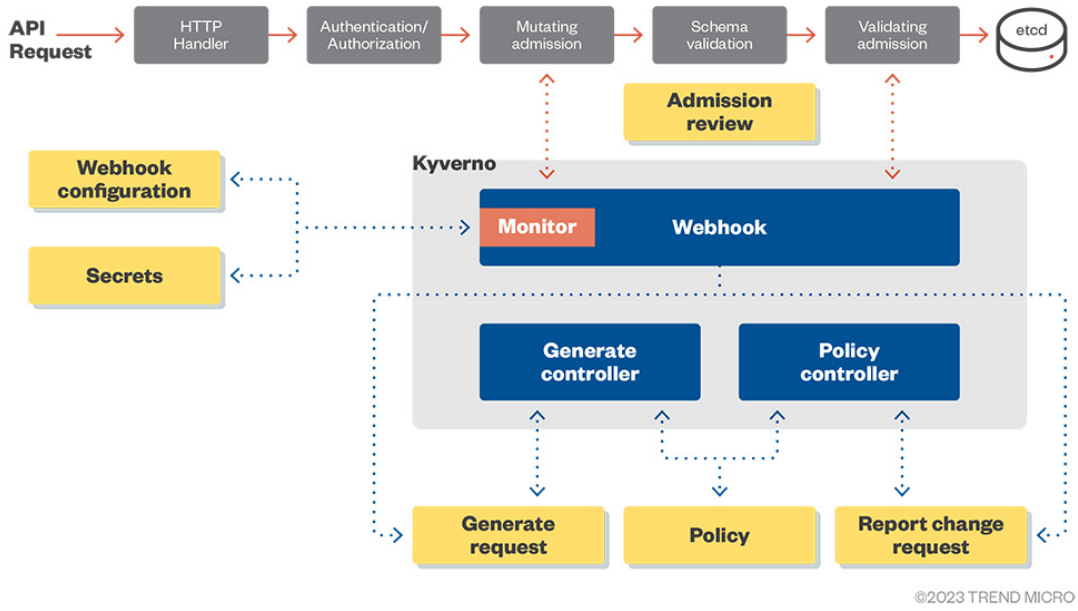

Figura 3.2: Architettura Kyverno [11].

Limitazioni di Kyverno

Di seguito vediamo le limitazioni di Kyverno:

- **Richiede Risorse di Calcolo Aggiuntive:** l'esecuzione dei controller Kyverno richiede risorse di calcolo aggiuntive, il che può aumentare l'overhead sul cluster, specialmente in ambienti ad alta scala.
- **Limitato alle Operazioni Supportate dalle CRD di Kyverno:** le policy sono limitate alle funzionalità e alle operazioni supportate dalle CRD di Kyverno. Qualsiasi operazione che non può essere espressa attraverso queste CRD non può essere gestita da Kyverno.

3.6 OPA (Open Policy Agent)

Open Policy Agent (OPA) è un motore di policy estremamente flessibile e versatile, utilizzato per applicare regole di sicurezza e governance in vari ambienti, incluso Kubernetes. Quando viene combinato con Gatekeeper, OPA diventa uno strumento potente per la gestione centralizzata delle policy nei cluster Kubernetes. Gatekeeper non si limita a sfruttare le capacità di OPA per valutare le policy, ma aggiunge funzionalità specifiche per Kubernetes, come l'integrazione nativa con l'API server e la gestione delle policy tramite risorse personalizzate.

Componenti e funzionamento

OPA e Gatekeeper collaborano attraverso una serie di componenti fondamentali:

- **Open Policy Agent (OPA):** è il cuore del sistema, un motore di policy che valuta le richieste basandosi su policy scritte in Rego, un linguaggio dichiarativo progettato per definire regole di policy.
- **Gatekeeper:** un controller Kubernetes che integra OPA con Kubernetes e fornisce funzionalità aggiuntive specifiche per Kubernetes.
- **ConstraintTemplates:** questi definiscono i tipi di vincoli che possono essere applicati. Ogni ConstraintTemplate include il codice Rego necessario per esprimere la logica della policy.
- **Constraints:** sono le istanze delle ConstraintTemplates che specificano regole concrete da applicare nel cluster. Ogni Constraint applica le regole definite a risorse specifiche o all'intero cluster.

Il processo mostrato in fig. 3.3 inizia con una richiesta API inviata al server API di Kubernetes. Questa richiesta viene intercettata dai webhook di Gatekeeper,

che la inoltrano a OPA. OPA valuta la richiesta utilizzando policy scritte in Rego, basandosi su dati forniti in formato JSON. Dopo l'elaborazione, OPA restituisce una decisione che Gatekeeper applica, consentendo o bloccando la richiesta in base alle regole definite.

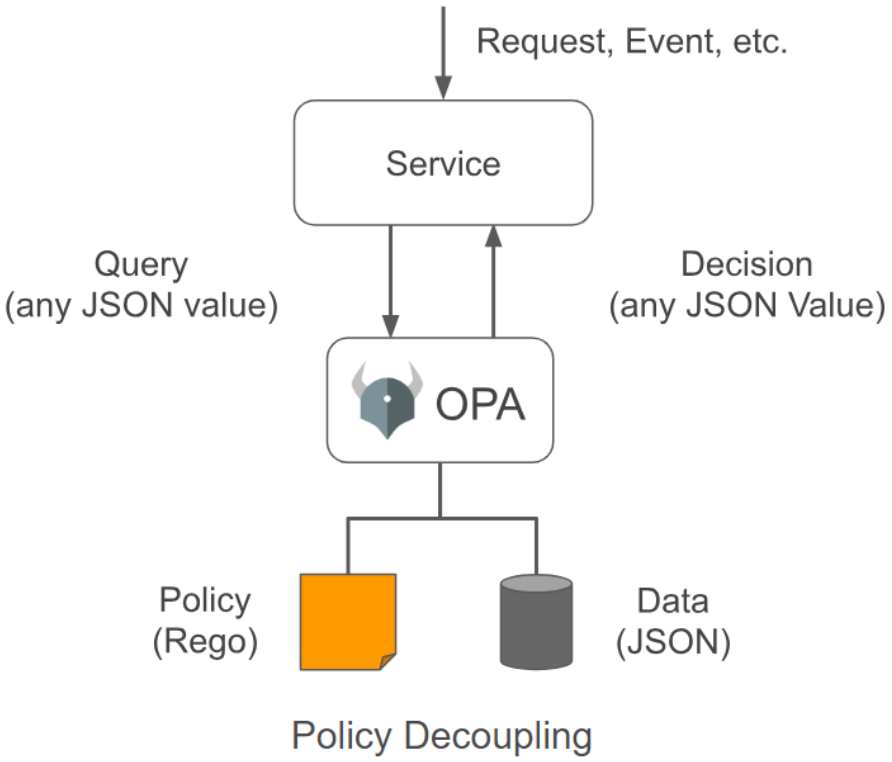

Figura 3.3: Architettura OPA/Gatekeeper [12].

Limitazioni di OPA/Gatekeeper

OPA e Gatekeeper rappresentano una combinazione potente per la gestione delle policy in Kubernetes, fornendo un controllo centralizzato e flessibile. Tuttavia ci sono alcune limitazioni legate al loro uso:

- **Dipendenza da Rego:** tutte le policy devono essere scritte in Rego.
- **Complessità operativa:** mantenere entrambi i componenti operativi, assicurandosi che le policy siano aggiornate e che l'infrastruttura supporti le valutazioni in tempo reale, può aumentare la complessità del sistema.

3.7 Webhook

Un ulteriore metodo di autenticazione degli accessi implementato da Kubernetes si basa sull'uso di un webhook HTTP. In generale, questo metodo viene adoperato per inviare notifiche asincrone da un server a un altro in risposta a eventi specifici. Infatti, tipicamente, queste richieste sono generate da un evento al quale l'applicazione web risponde inviando una richiesta HTTP all'Uniform Resource Locator (URL) assegnato al webhook.

Componenti e funzionamento

Nel caso specifico di Kubernetes, una richiesta POST HTTP viene utilizzata per notificare la presenza di un evento. In questo modo i privilegi dell'utente vengono determinati attraverso una valutazione esterna basata sulle informazioni fornite nella richiesta.

Per inizializzare questa modalità è necessario utilizzare un file di configurazione che rispetti il formato *kubeconfig*. Nello specifico, tramite questo file è possibile definire i parametri per gli *users*, ovvero gli API Server webhook, e per i *clusters*, i servizi resi accessibili. Inoltre, possono essere definiti altri campi come il *context* e l'ultima versione stabile supportate delle API. Un esempio di file di configurazione fornito dalla documentazione ufficiale di Kubernetes [13] è riportato in 3.6

Listing 3.6: Esempio file kubeconfig

```
# Kubernetes API version
apiVersion: v1
# kind of the API object
kind: Config
# clusters refers to the remote service.
clusters:
  - name: name-of-remote-authz-service
    cluster:
      # CA for verifying the remote service.
      certificate-authority: /path/to/ca.pem
      # URL of remote service to query. Must use 'https'. May not include parameters.
      server: https://authz.example.com/authorize

# users refers to the API Server's webhook configuration.
users:
  - name: name-of-api-server
    user:
      client-certificate: /path/to/cert.pem # cert for the webhook plugin to use
      client-key: /path/to/key.pem          # key matching the cert
```

```
20
21 # kubeconfig files require a context. Provide one for the API Server.
22 current-context: webhook
23 contexts:
24 - context:
25     cluster: name-of-remote-authz-service
26     user: name-of-api-server
27   name: webhook
```

In seguito alla configurazione, il webhook è pronto a ricevere le richieste POST. Nello specifico della concessione delle autorizzazioni la richiesta deve essere serializzata come oggetto *SubjectAccessReview*. Questo tipo di oggetto specifica al suo interno sia informazioni sull'utente che sta tentando di effettuare la richiesta sia informazioni sulla risorsa desiderata o gli attributi della richiesta.

Un esempio di payload è riportato in 3.7 dove è possibile evidenziare come sia l'utente, in questo esempio denominato *jane*, che gli attributi della risorsa sono definiti. Inoltre, la versione delle API viene nuovamente specificata, dal momento che gli oggetti appartenenti alle API webhook devono seguire le stesse norme di compatibilità che valgono per gli altri oggetti API di Kubernetes.

Listing 3.7: Esempio di payload SubjectAccessReview

```
1  {
2    "apiVersion": "authorization.k8s.io/v1beta1",
3    "kind": "SubjectAccessReview",
4    "spec": {
5      "resourceAttributes": {
6        "namespace": "kittensandponies",
7        "verb": "get",
8        "group": "unicorn.example.org",
9        "resource": "pods"
10     },
11     "user": "jane",
12     "group": [
13       "group1",
14       "group2"
15     ]
16   }
17 }
```

In risposta a questo oggetto, il servizio remoto è tenuto a permettere o rifiutate l'accesso alla risorsa utilizzando il campo status del pacchetto di risposta come indicato in 3.8

Listing 3.8: Esempio di payload risposta

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "status": {
    "allowed": true
  }
}

## OR ##

{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "status": {
    "allowed": false,
    "reason": "user does not have read access to the
   namespace"
  }
}
```

Una peculiarità di questo pacchetto è che la richiesta viene inizialmente rifiutata se il servizio non ha le informazioni necessarie per decidere se è permessa o meno. In seguito, altri servizi di autorizzazione possono permettere l'accesso alla risorsa. Se, invece, l'intento della risposta è quello di vietare non solo l'accesso alla risorsa, ma anche che altri servizi lo possano concedere, è possibile modificare la risposta come riportato in 3.9

Listing 3.9: Esempio di payload risposta negativa

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "status": {
    "allowed": false,
    "denied": true,
    "reason": "user does not have read access to the
   namespace"
  }
}
```

Limitazioni di Webhook

L'utilizzo di webhook offre significativi vantaggi sia in termini di personalizzazione che di capacità di validazione, dal momento che è possibile definire autonomamente

regole di validazioni complesse. Tuttavia, alcune sono presenti alcune limitazioni da considerare:

- **Complessità di Gestione e Configurazione:** la configurazione e la gestione dei webhook possono essere complesse, richiedendo una comprensione dettagliata del sistema e delle API di Kubernetes.
- **Dipendenza da Servizi Esterni:** i webhook dipendono da servizi esterni per eseguire la logica di validazione e mutazione, introducendo una dipendenza aggiuntiva.

Capitolo 4

Obiettivi della tesi

4.1 Introduzione al problema

Le configurazioni errate da parte degli amministratori rappresentano una delle criticità principali della sicurezza informatica moderna. Gli errori di configurazione comprendono permessi eccessivamente permissivi, regole di accesso non definite o obsolete, e configurazioni di sicurezza che non rispettano le politiche aziendali di controllo degli accessi. Tali errori derivano spesso dalla crescente complessità delle reti aziendali e dei sistemi distribuiti, che richiedono un numero sempre maggiore di configurazioni manuali per rispondere ai requisiti di sicurezza e operativi. Questi errori comportano rischi significativi per la sicurezza delle informazioni.

In un sistema come Kubernetes che gestisce ambienti containerizzati complessi e distribuiti la configurazione manuale dei controlli di accesso è particolarmente critica. Gli amministratori di rete devono definire chi può accedere a determinate risorse e con quali privilegi; tuttavia a causa di moltiplicarsi delle applicazioni e dei componenti, mantenere configurazioni coerenti è diventato sempre più difficile. Questi errori di configurazione possono portare a concedere permessi inutilmente ampi a utenti o servizi, aumentando i rischi sulla sicurezza come ad esempio accessi non autorizzati, compromissione di dati sensibili o interruzione dei servizi critici nell'infrastruttura.

La situazione è ulteriormente aggravata dalla mancanza di strumenti di verifica in grado di rilevare queste configurazioni errate in tempo reale. Con l'aumentare della complessità degli ambienti cloud e multi-tenant, la presenza di errori umani diventa quasi inevitabile, aggravata dalla frequenza con cui le configurazioni devono essere aggiornate per soddisfare i nuovi requisiti di sicurezza o i cambiamenti infrastrutturali. Per queste ragioni, le configurazioni errate sono ormai considerate una

delle principali fonti di vulnerabilità negli ambienti di rete attuali, rappresentando un rischio elevato per la resilienza e la sicurezza complessiva delle infrastrutture aziendali.

4.2 Scopo della Tesi

La tesi ha come scopo quello di creare un approccio automatizzato per tradurre e verificare le policy di accesso in Kubernetes, in modo che le configurazioni errate possano essere rilevate da software automatizzati e pertanto, ridurre la probabilità di errori umani, migliorando così la sicurezza complessiva del sistema. Ciò rende fondamentale la gestione delle policy in modo efficiente e sicuro in un ambiente dinamico come il cloud computing, dove le configurazioni di accesso cambiano continuamente per adattarsi a nuovi requisiti di sicurezza e a scenari operativi in evoluzione. Pertanto l'obiettivo principale della tesi è proprio quello di esplorare metodologie che possano portare all'automazione della definizione e della validazione delle policy, fornendo agli amministratori di rete il supporto necessario per configurazioni coerenti, sicure e adattabili ai cambiamenti.

Il lavoro sarà orientato verso due aspetti principali: una possibile modellazione architetturale di Kubernetes e la definizione di un linguaggio strutturato per la rappresentazione delle policy. Attraverso la modellazione dell'architettura di Kubernetes, il processo delle interazioni tra i componenti chiave del sistema, specialmente i flussi di richieste di accesso e di gestione delle autorizzazioni, verrà descritto in dettaglio.

In particolare, la definizione sarà completata con un linguaggio per la definizione delle policy basato su JSON, che fornirà un mezzo per rappresentare le configurazioni di sicurezza in modo chiaro e standardizzato, facilitando la loro traduzione in policy RBAC di basso livello. Questo lavoro intende offrire un contesto principalmente teorico non solo per la creazione di configurazioni coerenti e sicure, ma anche per il loro continuo miglioramento. Inoltre, verranno presentate scelte implementative e possibili casi d'uso per validare l'approccio proposto, spiegando come questo sistema di gestione delle policy possa contribuire a ridurre le vulnerabilità e a garantire il massimo livello di robustezza delle infrastrutture Kubernetes.

4.3 Fasi del lavoro di Tesi

Questo lavoro di tesi è articolato in quattro fasi:

- **Analisi del contesto cloud:** in questa fase si è analizzato come si sono evolute le modalità di deployment delle applicazioni, partendo dai server fisici dedicati, passando per la virtualizzazione, fino ad arrivare all'uso dei container. Questa analisi mette in luce le limitazioni delle soluzioni precedenti e i vantaggi introdotti dai container, che offrono isolamento, efficienza nell'uso delle risorse e portabilità. In questo contesto introduciamo Kubernetes, la piattaforma open-source per l'orchestrazione dei container che è diventata lo standard de facto nella gestione di ambienti distribuiti. Esaminiamo quindi come Kubernetes affronti sfide cruciali come il bilanciamento del carico, l'orchestrazione dello storage, l'ottimizzazione delle risorse e il self-healing, offrendo un framework resiliente e flessibile per eseguire e gestire sistemi distribuiti su larga scala.

- **Analisi delle modalità di controllo degli accessi in Kubernetes:** questa fase è incentrata sull'analisi delle principali modalità di controllo degli accessi in Kubernetes ovvero metodologie utilizzate per gestire le autorizzazioni all'interno di un cluster, con particolare attenzione al metodo RBAC, oggetto di approfondimento successivo.

- **Modellazione dell'architettura:** in questa fase viene costruito un modello architetturale formale per Kubernetes per descrivere le interazioni tra i componenti chiave del sistema, come l'API Server, etcd e i moduli di autorizzazione. Questo modello ci aiuta a comprendere nel dettaglio il flusso delle richieste di accesso e come vengono gestite le autorizzazioni, fornendo una base solida per implementare le policy in modo efficace.

- **Fase implementativa e di validazione:** in questa fase finale, si è sviluppato e validato un linguaggio specifico in formato JSON per definire le policy di accesso, progettato per essere tradotto in configurazioni RBAC di basso livello. Si progetta e implementa casi d'uso pratici che dimostrano come l'approccio proposto sia in grado di rilevare e correggere anomalie nelle configurazioni di accesso, migliorando la sicurezza complessiva del sistema e supportando gli amministratori di rete nella gestione quotidiana delle policy.

Capitolo 5

Design e Implementazione del Sistema di Gestione delle Policy in Kubernetes

In questo capitolo viene descritto il flusso di autorizzazione in Kubernetes, seguito dal design e dall'implementazione di un sistema di gestione delle policy di accesso. Il sistema introduce un approccio a più livelli che mira alla semplificazione della gestione delle policy. Successivamente, viene trattata l'implementazione del linguaggio per la definizione delle policy.

5.1 Flusso di autorizzazione attuale in Kubernetes

L'architettura di Kubernetes si basa su una serie di operazioni ben definite, che servono a garantire che solo richieste autorizzate possano accedere alle risorse del cluster. Questo processo comprende diverse fasi: l'autenticazione, l'autorizzazione e infine il controllo di ammissione (Adission Control).

La figura seguente mostra i componenti principali di un cluster Kubernetes, mettedno in evidenza il percorso seguito da una richiesta utente:

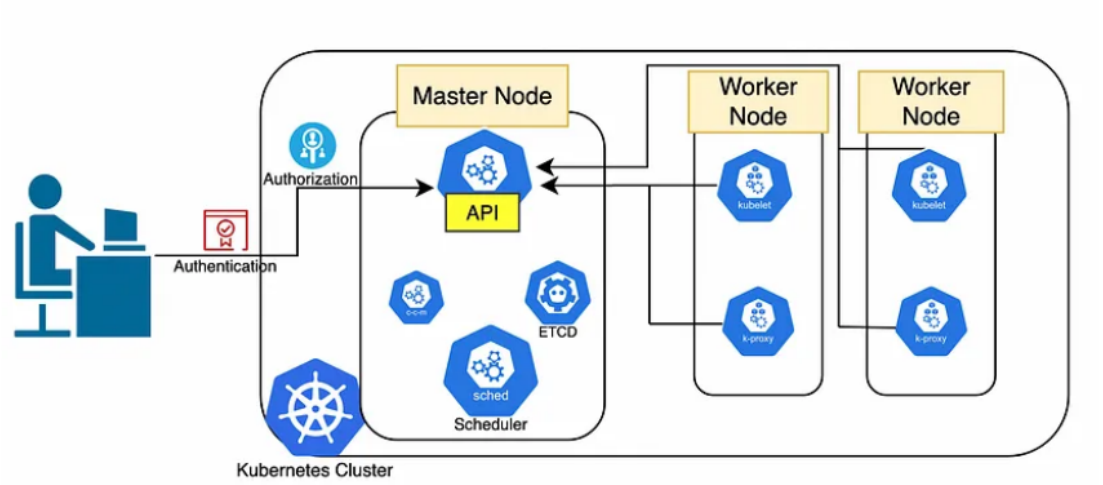

Figura 5.1: Flusso di autorizzazione [14].

Un client manda una richiesta all'API Server del cluster Kubernetes. Essa attraversa 3 fasi fondamentali, come riportato nel capitolo 3:

1. **Autenticazione**: l'API Server si assicura che la richiesta provenga da una fonte attendibile verificando l'identità del richiedente. Per questo, utilizza uno dei metodi di autenticazione supportati, come certificati client, token di accesso o integrazioni con provider esterni.

2. **Autorizzazione**: una volta autenticata l'identità, l'API Server invia la richiesta ai moduli di autorizzazione configurati, eseguendoli in sequenza. Ogni modulo valuta la richiesta in base alle policy definite e memorizzate in etcd, il datastore centrale di Kubernetes. La decisione di approvare o rifiutare la richiesta si basa sull'esito di questa valutazione.

3. **Admission Control**: se la richiesta viene autorizzata, passa attraverso i controller di ammissione, che possono applicare ulteriori controlli o modifiche prima che l'operazione venga effettivamente eseguita nel cluster.

Dettaglio del Processo di Autorizzazione

Il secondo passaggio, l'autorizzazione, è cruciale per garantire che solo le entità con i permessi appropriati possano interagire con le risorse del cluster. Kubernetes mette a disposizione diversi moduli di autorizzazione per soddisfare esigenze specifiche; tra i principali troviamo:

- **Role-Based Access Control (RBAC)**: il meccanismo di autorizzazione predefinito in Kubernetes basato sui ruoli.

- **Attribute-Based Access Control (ABAC)**: consente di definire policy basate su attributi delle richieste, come l'utente, la risorsa richiesta e l'azione desiderata.
- **Webhook Authorization**: delega le decisioni di autorizzazione a un servizio esterno tramite chiamate HTTP. Questo consente di implementare logiche altamente personalizzate.

Valutazione delle Policy di Autorizzazione

Quando una richiesta raggiunge la fase di autorizzazione, l'API Server la sottopone ai moduli di autorizzazione configurati, seguendo un ordine sequenziale. Ogni modulo applica le proprie policy per determinare se la richiesta deve essere approvata o rifiutata. La configurazione di più più moduli di autorizzazione consente un controllo più granulare sull'accesso alle risorse.

Processo di Valutazione

Il funzionamento del processo di valutazione è il seguente:

1. **Esecuzione sequenziale dei moduli:** l'API Server invia la richiesta al primo modulo di autorizzazione configurato.
2. **Decisione del Modulo**: il modulo valuta la richiesta basandosi sulle policy disponibili:
   - Se il modulo **approva** la richiesta, l'autorizzazione ha esito positivo e il processo termina.
   - Se il modulo **rifiuta** la richiesta, l'autorizzazione ha esito negativo e il processo termina.
   - Se il modulo **non è in grado di decidere**, la richiesta viene passata al modulo successivo.
3. **Esito Finale:** se tutti i moduli non riescono a decidere o rifiutano la richiesta, questa viene negata dall'API Server.

Policy Memorizzate in etcd

Le policy utilizzate dai moduli di autorizzazione sono generalmente memorizzate in etcd, il datastore distribuito di Kubernetes. Questo permette una gestione centralizzata e persistente delle policy, facilitando aggiornamenti e sincronizzazioni tra i componenti del cluster. Ciò assicura che le policy siano sempre aggiornate e coerenti in tutto il cluster, semplificando la gestione e la manutenzione.

5.2 Design dell'architettura e flusso di autorizzazione proposto

Al fine di migliorare la gestione attuale delle policy di accesso in Kubernetes, e' stata progettata un'architettura che introduce nuovi moduli di supporto, mirati a semplificare e rendere più sicuro il prcesso di creazione, traduzione e verifica delle policy. L'architettura proposta si basa su un flusso che automatizza la generazione delle configurazioni RBAC a partire dagli intenti di alto livello definiti dall'amministratore.

5.2.1 Descrizione dell'architettura

L'idea alla base è quella di integrare all'architettura Kubernetes già esiste 3 moduli distinti ma interconnessi:

1. **Modulo di Generazione delle Policy:** questo componente si occupa di trasformare gli intenti di alto livello dell'amministratore in policy formali rappresentate in formato JSON.

2. **Modulo di Traduzione delle Policy:** questo riceve le policy in formato JSON generate dal modulo precedente e le converte in configurazioni RBAC di basso livello.

3. **Modulo di Verifica delle Policy:** infine quest'ultimo componente si occupa di monitorare continuamente lo stato del cluster per assicurarsi che le configurazioni RBAC attuali corrispondano alle policy di alto livello definite. In caso di difformità, questo modulo segnala eventuali errori ed eventuali correzioni all'amministratore, il quale ha facoltà di decidere se applicare o meno le corrrezioni.

5.2.2 Design proposto

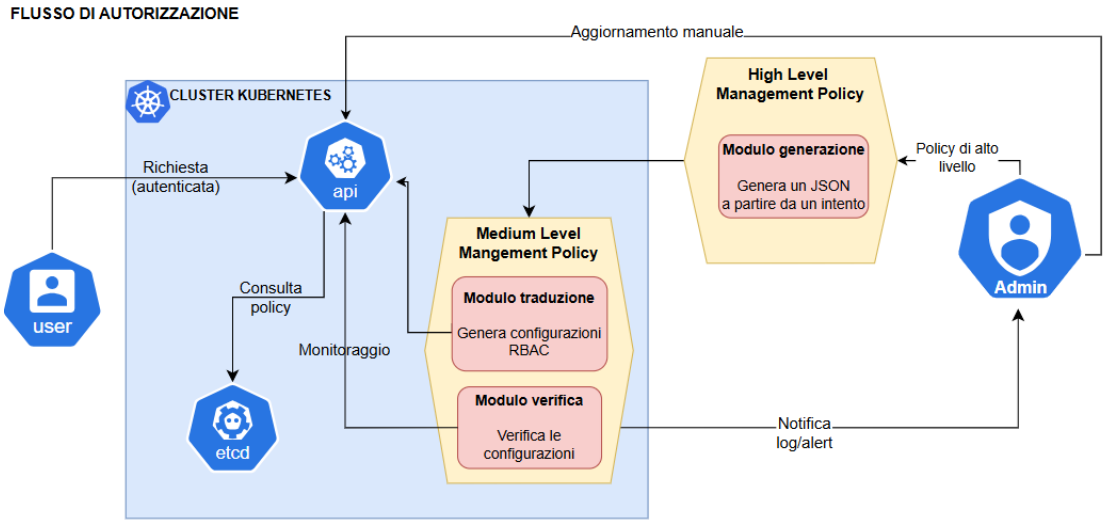

Figura 5.2: Architettura proposta

5.2.3 Flusso di una richiesta di autorizzazione

In questa sezione viene esaminato il flusso completo di una richiesta, dall'inserimento della policy da parte dell'amministratore fino all'esecuzione dell'azione da parte dell'utente.

Passaggi del flusso

1. **Definizione della policy da parte dell'amministratore:** l'amministratore definisce il suo intento e utilizza il modulo di generazione per definire le policy di accesso. Queste vengono formalizzate in formato JSON, per una successiva rielaborazione.

2. **Distribuzione delle policy al cluster:** le policy generate vengono inviate al cluster attraverso l'API REST.

3. **Ricezione delle policy dal modulo di traduzione interno:** all'interno del cluster, il modulo di traduzione riceve le nuove policy. Da qui inizia il processo di traduzione.

4. **Traduzione delle policy in configurazioni RBAC:** il modulo di traduzione converte le policy JSON in configurazioni RBAC di basso livello, creando o aggiornando le risorse necessarie come *Role, RoleBinding, ClusterRole* e *ClusterRoleBinding*.

5. **Applicazione delle configurazioni al cluster:** le configurazioni RBAC vengono ricevute dall'API Server che provvede ad applicarle al cluster salvandole in *etcd*.

6. **Verifica delle configurazioni:** il modulo di verifica interno controlla che le configurazioni attuali del cluster siano coerenti con le policy di alto livello. In caso di discrepanze vengono generate notifiche o alert.

7. **Esecuzione della richiesta da parte dell'utente:** L'utente esterno al cluster invia una richiesta all'API Server di Kubernetes per eseguire un'azione specifica.

8. **Fase di autenticazione:** l'API Server autentica l'identità del richiedente, verificando le credenziali fornite.

9. **Fase di autorizzazione:** l'API Server valuta la richiesta utilizzando le configurazioni RBAC aggiornate e decide se approvare o rifiutare la richiesta in base alle regole d'accesso.

10. **Fase di Admission Control:** se la richiesta è autorizzata, passa attraverso ulteriori controlli che possono applicare validazioni o modifiche aggiuntive.

11. **Esecuzione dell'azione richiesta:** la richiesta viene eseguita, e l'utente ottiene il risultato desiderato. In caso di rifiuto viene fornito un messaggio di errore.

Tabella riepilogativa

Ruolo	Posizione	Azione
Amministratore	Esterno al cluster	Definisce le policy di alto livello e riceve notifiche su eventuali discrepanze o suggerimenti dal sistema di verifica.
Modulo di Generazione	Esterno al cluster	Genera un file JSON basato sulle policy definite e lo invia al modulo di traduzione tramite l'API REST.
Modulo di Traduzione	Interno al cluster	Riceve le policy, le traduce in configurazioni RBAC e le applica al cluster.
Modulo di Verifica	Interno al cluster	Verifica che le configurazioni RBAC siano coerenti con le policy di alto livello e segnala eventuali discrepanze.
API Server	Interno al cluster	Gestisce le richieste degli utenti, applica le configurazioni RBAC per autorizzare o negare le azioni.
etcd	Interno al cluster	Memorizza le configurazioni RBAC e altre informazioni di stato del cluster.
Utente	Esterno al cluster	Invia richieste al cluster per eseguire azioni sulle risorse Kubernetes.

Tabella 5.1: Ruoli e azioni nel flusso della richiesta.

5.2.4 Conclusioni sulle scelte di design

Il design proposto ha come obiettivo quello del bilanciamento tra efficienza delle risorse, prestazioni e sicurezza. Si è scelto di posizionare il **modulo di generazione delle policy** al di fuori del cluster per ridurre il carico sulle risorse del cluster, evitando che la fase di generazione, spesso la più intesiva in termini di computazione, possa influenzare le performance delle applicazioni principali. Mentre i moduli di **traduzione** e **verifica** delle policy sono collocati all'interno del cluster, favorendo una riduzione della latenza interna. Grazie a questa disposizione possono accedere direttamente all'API Server, migliorando i tempi di applicazione e verifica delle policy.

In termini di sicurezza, mantenere i due moduli sopracitati all'interno del cluster riduce sensibilmente l'esposizone a potenziali attacchi esterni. Inoltre il **modulo di generazione** esterno offre il vantaggio di essere centralizzato, il che consente di gestire policy di più cluster.

Tuttavia questa architettura ha dei punti critici, tra cui la complessità di sincronizzazione tra il modulo esterno e quelli interni, che richiede una gestione affidabile della comunicazione per prevenire erorri. Inoltre, l'affidabilità dell'intero sistema dipende dal corretto funzionamento del modulo esterno, eventuali malfunzionamenti in questo modulo potrebbero limitare l'efficacia della gestione delle policy nel cluster.

5.3 Definizione del linguaggio per le policy di accesso

Per implementare un sistema di gestione delle policy di accesso efficace in Kubernetes è necessario definire un linguaggio in grado di esprimere in modo chiaro e definito i permessi di accesso. Si è scelto di utilizzare il formato JSON perchè offre una serie di vantaggi:

- **Compatibilità con REST API di Kubernets:** JSON è il formato nativo per la gestione delle risorse in Kubernetes, pertanto si integra più facilmente con l'infrastruttura di Kubernetes, riducendo la necessità di adattamenti o traduzioni successive.
- **Standardizzazione e portabilità:** JSON è uno standard ampiamente utilizzato e compatibile con la maggior parte delle tecnologie moderne, facilitando future integrazioni con altri sistemi di gestione delle policy.
- **Flessibilità e Manutenibilità:** JSON è particolarmente flessibile e permette di rappresentare strutture dati complesse. Consente di aggiungere successivamente nuovi campi e quindi garantisce che il sistema possa evolversi e adattarsi senza richiedere una revisione completa del formato delle policy.
- **Parsing e Validazione:** JSON è supportato da molti linguaggi di programmazione e dispone di librerie per il parsing e la validazione. Ad esempio librerie come **Kubernetes Python Client** o **Client-go** facilitano la trasformazione diretta di documenti JSON in configurazioni RBAC di basso livello, fornendo specifici metodi per la gestione dei *Role*, *RoleBinding*, *ClusterRole*, *ClusterRoleBinding* in Kubernetes.

- **Performance ed Efficienza:** JSON è un formato leggero e facile da serializzare e deserializzare risultando particolarmente efficiente in termini di prestazione.

5.3.1 Struttura di una policy

La struttura JSON è pensata per rappresentare in modo completo una policy di accesso. Include tutti gli elementi necessari per tradurre gli intenti di accesso definiti dall'amministratore in configurazioni RBAC di basso livello. Risponde perfettamente a queste domande:

- Chi è autorizzato ad accedere alle risorse
- Quali permessi specifici sono concessi
- Su quali risorse si applicano questi permessi
- Eventuali condizioni aggiuntive

Campo `role`

Il campo `role` rappresenta uno degli elementi fondamentali di una policy JSON in Kubernetes, in quanto definisce un insieme di permessi raggruppati sotto un ruolo specifico. Attraverso il `role`, è possibile assegnare tali permessi a utenti, gruppi o service account, semplificando così la gestione centralizzata delle autorizzazioni. In Kubernetes, un ruolo (`Role`) consente di specificare quali azioni possono essere eseguite su determinate risorse.

Un `Role`, quindi raggruppa una serie di permessi che autorizzano le azioni definite nel campo `verbs` su particolari risorse. Per essere effettivo, un `Role` deve essere creato all'interno del cluster prima di essere assegnato. Successivamente attraverso un `RoleBinding` si crea l'associazione tra uno specifico ruolo e i soggetti ai quali si desidera concedere tali permessi, rendendo effettiva la policy nel contesto del cluster.

Campo `subject`

Il campo `subject` definisce a chi vengono assegnati i permessi definiti nel `role`. In Kubernetes, il `subject` può rappresentare utenti, gruppi o service account e ciascun tipo di soggetto può essere associato a ruoli che ne regolano l'accesso alle risorse del cluster. Per identificare con precisione il `subject`, sono inclusi diversi sotto-campi:

- **`kind`**: indica il tipo di soggetto cui si applicano i permessi e può assumere valori come `User`, `Group` o `ServiceAccount`.
- **`name`**: specifica il nome del soggetto esatto (utente, gruppo o service account) cui il ruolo si riferisce.
- **`apiGroup`**: identifica il gruppo API a cui appartiene il soggetto. Per i soggetti standard come `User`, `Group` e `ServiceAccount`, questo campo è impostato su `rbac.authorization.k8s.io`.
- **`namespace`** (obbligatorio solo per `ServiceAccount`): definisce il namespace di appartenenza del service account. Questo campo non è necessario per `User` o `Group`, poiché non sono legati a un namespace specifico.

Campo `resource`

Il campo `resource` definisce la risorsa Kubernetes su cui si applicano i permessi specificati nella policy. L'amministratore con questo campo può limitare i permessi a risorse specifiche, indicando anche il contesto operativo (ad esempio, il namespace) se necessario. La struttura del campo `resource` include vari sotto-campi, che consentono di identificare con precisione il tipo di risorsa e i limiti di applicazione della policy.

- **`kind`**: indica il tipo di risorsa su cui si desidera applicare la policy. Può includere sia risorse standard di Kubernetes, come `pods`, `services` e `configmaps`, sia risorse personalizzate definite tramite Custom Resource Definitions (CRD).
- **`apiGroups`**: specifica il gruppo API a cui appartiene la risorsa. Per le risorse standard di Kubernetes (come `pods`, `services`, `configmaps`), questo campo è impostato sul gruppo API core e può essere lasciato vuoto (`apiGroups: [""]`). In caso di risorse personalizzate, è necessario specificare il gruppo API corrispondente (ad esempio, una risorsa `MyResource` definita nel gruppo `example.com` verrà rappresentata con `apiGroups: ["example.com"]`).
- **`namespace`** (facoltativo): definisce il namespace in cui è situata la risorsa. Questo sotto-campo è rilevante solo per le risorse legate a un singolo namespace e, in particolare, per i `Role`, che sono limitati a un singolo namespace. Tuttavia, se la policy è definita tramite un `ClusterRole` (valido a livello di cluster e non limitato ai singoli namespace), il campo `namespace` non è applicabile e viene ignorato.
- **`name`** (facoltativo): permette di specificare il nome della risorsa, limitando la policy a una singola istanza di quel tipo di risorsa all'interno del namespace. Tuttavia, nel sistema RBAC di Kubernetes, non supporta la selezione di una

singola risorsa per nome, e in molti casi il valore `name` viene impostato su `"*"`, indicando che i permessi si applicano a tutte le risorse di quel tipo nel namespace.

Campo `verbs`

Il campo `verbs` specifica le azioni che un'entità è autorizzata ad eseguire sulle risorse definite nella policy. Queste azioni rappresentano i permessi concreti concessi e sono fondamentali per definire il comportamento autorizzato nel cluster. In Kubernetes, i verbi descrivono operazioni CRUD (Create, Read, Update, Delete) e altre operazioni specifiche.

I verbi supportati dal sistema RBAC di Kubernetes includono:

- **`get`**: autorizza la lettura dei dettagli di una risorsa specifica.
- **`list`**: consente di elencare tutte le risorse di un determinato tipo (ad esempio, tutti i pod in un namespace).
- **`watch`**: permette di monitorare le risorse per rilevare modifiche.
- **`create`**: consente di creare nuove risorse.
- **`update`**: autorizza la modifica o l'aggiornamento di risorse esistenti.
- **`patch`**: permette di applicare modifiche parziali a una risorsa esistente.
- **`delete`**: concede il permesso di eliminare una risorsa specifica.
- **`deletecollection`**: autorizza l'eliminazione di un'intera collezione di risorse (ad esempio, tutti i pod in un namespace) in un'unica operazione.

Esempio completo di policy in JSON

Si riporta un esempio pratico di policy in formato JSON che aiuta a comprendere come gli intenti di alto livello definiti dall'amministratore possano essere tradotti in configurazioni concrete. Di seguito è riportata una policy che autorizza un utente specifico, appartenente al gruppo `developers`, ad accedere a tutte le risorse di tipo `pods` all'interno del namespace `development`. I permessi includono operazioni di lettura, come `get` e `list`, e operazioni di scrittura, come `update`.

Listing 5.1: Esempio di Policy in JSON

```
{
  "role": "developer",
  "subject": {
    "kind": "User",
    "name": "john.doe",
    "apiGroup": "rbac.authorization.k8s.io",
    "namespace": "development" // Solo per ServiceAccount
  },
  "resource": {
    "kind": "pods",
    "namespace": "development",
    "apiGroup": "",
    "name": "*"
  },
  "verbs": ["get", "list", "create", "update"]
}
```

5.4 Traduzione delle policy

La traduzione delle policy è un passaggio fondamentale nel processo di gestione automa delle configurazioni RBAC. Il modulo di traduzione riceve le policy definite ad livello alto in formato JSON, e le converte in configurazioni RBAC di basso livello applicabili direttamente al cluster Kubernetes. Questo processo automatizzato elimina la necessità di creare manualmente risorse come `Role`, `RoleBinding`, `ClusterRole`, e `ClusterRoleBinding`.

5.4.1 Processo di traduzione

Il processo di traduzione delle policy prevede l'esecuzione sequenziale dei seguenti passaggi:

1. **Ricezione della Policy**
   Il modulo di traduzione riceve una policy in formato JSON dal modulo di generazione, questa include tutti i dettagli necessari, come il ruolo, il soggetto, le risorse e i verbi associati.

2. **Analisi della Policy**
   La policy viene analizzata per estrarre i campi chiave:

   - `role`
   - `subject`
   - `resource`
   - `verbs`

3. **Creazione delle Configurazioni RBAC**
   Una volta analizzati i campi, il modulo di traduzione genera le seguenti risorse RBAC:

   - `Role` o `ClusterRole`: definiscono i permessi.
   - `RoleBinding` o `ClusterRoleBinding`: associano i ruoli ai soggetti definiti nella policy.

4. **Applicazione delle Configurazioni**
   Le configurazioni vengono inviate all'API Server di Kubernetes, dove vengono salvate in `etcd`. Da questo momento, le configurazioni sono attive e applicabili a tutte le richieste future.

5.4.2 Caso d'uso 1

Intento dell'amministratore

Consentire al gruppo `"sviluppatori"` di leggere i pod nel namespace `"sviluppo"`.

Policy Generata in JSON

Listing 5.2: Policy JSON caso d'uso 1

```
{
    "role": "lettura-pod-sviluppo",
    "subject": {
        "kind":
        "Group",
        "name": "sviluppatori",
        "apiGroup": "rbac.authorization.k8s.io"
    },
    "resource": {
        "kind": "pods",
        "apiGroups": [""],
        "namespace": "sviluppo",
        "name": "*"
    },
    "verbs": ["get", "list", "watch"]
}
```

Traduzione in Configurazioni RBAC

- Creazione del *Role*

Listing 5.3: Role caso d'uso 1

```
    apiVersion: rbac.authorization.k8s.io/v1
    kind: Role
    metadata:
        name: lettura-pod-sviluppo
        namespace: sviluppo
    rules:
        - apiGroups: [""]
        resources: ["pods"]
        verbs: ["get", "list", "watch"]

```

- Creazione del *RoleBinding*

Listing 5.4: RoleBinding caso d'uso 1

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
    name: binding-lettura-pod-sviluppatori
    namespace: sviluppo
subjects:
    - kind: Group
      name: sviluppatori
      apiGroup: rbac.authorization.k8s.io
roleRef:
    kind: Role
    name: lettura-pod-sviluppo
    apiGroup: rbac.authorization.k8s.io

```

Applicazione delle configurazioni

Una volta generate le risorse RBAC, queste vengono applicate al cluster utilizzando il comando `kubectl apply`, che invia le configurazioni al server API di Kubernetes:

Listing 5.5: Applicazione delle configurazioni RBAC

```
kubectl apply -f role.yaml
kubectl apply -f rolebinding.yaml
```

Il file `role.yaml` contiene la definizione del ruolo, mentre `rolebinding.yaml` associa il ruolo al soggetto definito nella policy JSON. Una volta applicati, questi file vengono elaborati dal server API e salvati in `etcd`, rendendo le configurazioni immediatamente operative.

5.4.3 Caso d'uso 2

Questo caso dimostra come autorizzare un'entità specifica (un ServiceAccount) a modificare risorse nel cluster.

Intento dell'amministratore

Consentire al ServiceAccount `"deployment-manager"` di aggiornare i Deployment nel namespace `"produzione"`.

Policy Generata in JSON

Listing 5.6: Policy JSON caso d'uso 2

```
{
    "role": "aggiornamento-deployment-produzione",
    "subject": {
        "kind": "ServiceAccount",
        "name": "deployment-manager",
        "namespace": "produzione"
    },
    "resource": {
        "kind": "deployments",
        "apiGroups": ["apps"],
        "namespace": "produzione",
        "name": "*"
    },
    "verbs": ["update"]
}
```

Traduzione in Configurazioni RBAC

- Creazione del *Role*

Listing 5.7: Role caso d'uso 2

```
    apiVersion: rbac.authorization.k8s.io/v1
    kind: Role
    metadata:
        name: aggiornamento-deployment-produzione
        namespace: produzione
    rules:
    - apiGroups: ["apps"]
      resources: ["deployments"]
      verbs: ["update"]

```

- Creazione del *RoleBinding*

Listing 5.8: RoleBinding caso d'uso 2

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
    name: binding-aggiornamento-deployment
    namespace: produzione
subjects:
- kind: ServiceAccount
  name: deployment-manager
  namespace: produzione
roleRef:
    kind: Role
    name: aggiornamento-deployment-produzione
    apiGroup: rbac.authorization.k8s.io

```

Applicazione delle configurazioni

Di seguito si riportano i già noti comandi per applicare le configurazioni:

Listing 5.9: Applicazione delle configurazioni RBAC

```
kubectl apply -f role.yaml
kubectl apply -f rolebinding.yaml
```

5.4.4 Caso d'uso 3

In questo caso vediamo una configurazione avanzata dove un intento viene tradotto in più file di configurazioni di basso livello.

Intento dell'amministratore

1. Consentire al gruppo `"devops-team"` di gestire i Deployment e ConfigMap nel namespace `"staging"`.
2. Assegnare a un membro senior del gruppo, `"alice"`, il ruolo aggiuntivo di supervisore a livello di cluster, con accesso in lettura a tutte le risorse.

Policy Generata in JSON

Listing 5.10: Policy caso d'uso 3

```
[
  {
    "role": "gestione-staging",
    "subject": {
      "kind": "Group",
      "name": "devops-team",
      "apiGroup": "rbac.authorization.k8s.io"
    },
    "resource": {
      "kind": ["deployments", "configmaps"],
      "apiGroups": ["apps", ""],
      "namespace": "staging",
      "name": "*"
    },
    "verbs": ["get", "create", "update", "delete"]
  },
  {
    "role": "supervisore-cluster",
    "subject": {
      "kind": "User",
      "name": "alice",
      "apiGroup": "rbac.authorization.k8s.io"
    },
    "resource": {
      "kind": "*",
      "apiGroups": ["*"],
      "namespace": "*",
      "name": "*"
```

```
    },
    "verbs": ["get", "list", "watch"]
  }
]
```

Traduzione in Configurazioni RBAC

- Creazione dei *Role*

Listing 5.11: Role per gestione del namespace staging

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: gestione-staging
  namespace: staging
rules:
  - apiGroups: ["apps"]
    resources: ["deployments"]
    verbs: ["get", "create", "update", "delete"]
  - apiGroups: [""]
    resources: ["configmaps"]
    verbs: ["get", "create", "update", "delete"]

```

Listing 5.12: ClusterRole per supervisione a livello di cluster

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: supervisore-cluster
rules:
  - apiGroups: ["*"]
    resources: ["*"]
    verbs: ["get", "list", "watch"]

```

- Creazione dei *RoleBinding*

Listing 5.13: RoleBinding per il gruppo devops-team

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: binding-gestione-staging
  namespace: staging
subjects:
  - kind: Group
    name: devops-team
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: gestione-staging
  apiGroup: rbac.authorization.k8s.io

```

Listing 5.14: ClusterRoleBinding per l'utente alice

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: binding-supervisore-cluster
subjects:
  - kind: User
    name: alice
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: supervisore-cluster
  apiGroup: rbac.authorization.k8s.io

```

Applicazione delle configurazioni

Le configurazioni vengono applicate al cluster utilizzando i seguenti comandi:

Listing 5.15: Applicazione delle configurazioni RBAC

```
kubectl apply -f role-gestione-staging.yaml
kubectl apply -f clusterrole-supervisore-cluster.yaml
kubectl apply -f rolebinding-gestione-staging.yaml
kubectl apply -f clusterrolebinding-supervisore-cluster.yaml
```

5.5 Verifica delle policy

Il modulo di verifica posizionato all'interno del cluster Kubernetes svolge un ruolo cruciale nel garantire la coerenza tra le policy di alto livello definite dall'amministratore e le configurazioni RBAC implementate nel cluster. Attraverso un'analisi strutturata, il modulo identifica e segnala eventuali discrepanze, suggerendo correzioni per mantenere il sistema conforme ai requisiti di accesso definiti.

In questa sezione analizzeremo in modo formale un possibile approccio per la verifica automatizzata delle policy basata sulla definizione proposta nel documento [15]. L'obiettivo è garantire che il sistema mantenga una coerenza tra gli intenti dell'amministratore e lo stato effettivo del cluster.

5.5.1 Modello di Specifica e Implementazione

La verifica si basa su due modelli fondamentali come descritto in [15]:

- **Modello di specifica delle policy** (S^+): rappresentano le policy di alto livello definite dall'amministratore ed indicano le azioni consentite per ciascun soggetto e risorsa.
- **Modello di implementazione delle policy** (I): eappresenta le configurazioni RBAC effettivamente applicate nel cluster, che determinano le azioni consentite.

Poiché RBAC non consente di definire esplicitamente azioni negate, il modello S^- non viene considerato.

La comparazione tra questi due modelli ha come obiettivo quella di rilevare discrepanze che possono compromettere la sicurezza o la funzionalità del sistema.

5.5.2 Identificazione delle anomalie

Il modulo di verifica confronta le specifiche e l'implementazione per individuare anomalie del tipo:

- **Permessi mancanti** ($\bar{S}^+ = S^+ \setminus I$): azioni richieste nelle specifiche ma non configurate nel cluster.
- **Permessi superflui** $I \setminus S^+$: azioni configurate nel cluster ma non richieste dalle specifiche.

La verifica quindi consiste nel controllare che ogni azione in S^+ sia effettivamente configuraza in I.

5.5.3 Processo di verifica

Il processo di verifica comprende i seguenti passaggi:

1. **Raccolta delle Policy di Alto Livello**: il modulo accede alle specifiche delle policy definite dall'amministratore (riceve le policy di alto livello dal modulo di generazione).

2. **Raccolta delle configurazioni RBAC**: le configurazioni effettive vengono richieste all'API Server, le quali sono salvate in `etcd`.

3. **Confronto tra Specifiche e Implementazione**: utilizzando algoritmi automatizzati, il modulo calcola i set di anomalie:

$$\bar{S}^+ = S^+ \setminus I \quad \text{e} \quad I \setminus S^+$$

4. **Identificazione delle Anomalie**: le discrepanze rilevate vengono classificate come permessi mancanti ($\bar{S}^+$) o permessi superflui ($I \setminus S^+$).

5. **Notifica e Suggerimenti**: il modulo genera notifiche dettagliate e suggerimenti per la correzione delle anomalie.

5.5.4 Suggerimenti per la correzione delle anomalie

In caso di discrepanze, il modulo di verifica fornisce suggerimenti specifici per correggere le configurazioni RBAC, ad esempio:

- **Aggiunta di permessi mancanti**: per avere coerenza tra le policy di alto livello e le configurazioni del cluster.

- **Rimozione di permessi non necessari**: ovvero rimuovere configurazioni che consentono più azioni del necessario.

Queste correzioni vengono presentate all'amministratore, che può approvarle o modificarle direttamente prima dell'applicazione.

5.5.5 Monitoraggio continuo

Il modulo di verifica opera in modalità continua, monitorando costantemente le modifiche alle configurazioni RBAC nel cluster. Questo perchè l'introduzione di eventuali anomalie può derivare da modifiche dinamiche o successive.

5.5.6 Integrazione con Kubernetes

Il modulo di verifica è integrato direttamente nel cluster Kubernetes e sfrutta l'API Server per accedere alle configurazioni RBAC. La sua posizione interna al cluster garantisce:

- **Riduzione della latenza**: grazie all'accesso diretto alle risorse del cluster.
- **Maggiore sicurezza**: minimizzando l'esposizione del modulo a potenziali attacchi esterni.

Capitolo 6

Conclusioni e Lavori Futuri

Il lavoro presentato in questa tesi rappresenta un primo passo verso un sistema più sicuro e automatizzato per la gestione delle policy di accesso in Kubernetes. I capitoli precedenti si sono concentrati sull'elaborazione di un modello per la gestione delle policy di accesso con la formalizzazione di un linguaggio specifico per la definizione delle policy. Grazie all'analisi delle caratteristiche di Kubernetes e delle sue modlaità di contorllo degli accessi è stato possibile proporre una soluzione a più moduli per migliorare la gestione degli accessi attraverso configurazioni RBAC.

Nonostante i risultati ottenuti sono emersi alcuni limiti che offrono opportunità per futuri studi e miglioramenti.

Estensione del Modello di Accesso

Sebbene la tesi si concentri esclusivamente su RBAC, una possibile estensione potrebbe includere altre modalità di accesso utilizzando strumenti avanzati come OPA. Questo può favorire una maggiore flessibilità e granularità nella definizione delle policy, rispondendo a requisiti più complessi.

Automazione della Correzione delle Anomalie

Attualmente, il sistema proposto segnala eventuali anomalie nelle configurazioni delle policy. Un'estensione interessante potrebbe essere l'implementazione di un modulo che automatizzi la correzione delle configurazioni errate, garantendo una coerenza continua senza interventi manuali.

Supporto per Ambienti Multi-Cluster

L'attuale implementazione è progettata per funzionare all'interno di un singolo cluster Kubernetes. Un'estensione futura potrebbe concentrarsi sul supporto di

ambienti multi-cluster, garantendo coerenza e sicurezza delle policy in infrastrutture distribuite.

Bibliografia

[1] Kubernetes Authors. *Kubernetes - Documentation*. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. Accessed: 2024-10-12. 2024 (cit. alle pp. 3, 5).

[2] David Bernstein. «Containers and Cloud: From LXC to Docker to Kubernetes». In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84. DOI: 10.1109/MCC.2014.51 (cit. a p. 3).

[3] Red Hat. *Kubernetes architecture*. https://www.redhat.com/it/topics/containers/kubernetes-architecture. Accessed: 12 October 2024. 2024 (cit. a p. 5).

[4] Wikipedia contributors. *Kubernetes*. https://it.wikipedia.org/wiki/Kubernetes. Accessed: 12 October 2024. 2024 (cit. a p. 5).

[5] phoenixNAP. *What is a Kubernetes Pod?* https://phoenixnap.com/kb/kubernetes-pod. Accessed: 19 October 2024. 2024 (cit. a p. 13).

[6] Kubernetes Authors. *Kubernetes Service - Documentation*. https://kubernetes.io/docs/concepts/services-networking/service/. Accessed: 2024-10-19. 2024 (cit. a p. 13).

[7] Kubernetes Documentation. *Controlling Access to the Kubernetes API*. Accessed: 2024-10-23. 2024. URL: https://kubernetes.io/docs/concepts/security/controlling-access/ (cit. alle pp. 16, 17).

[8] Kubernetes Documentation. *Role-Based Access Control (RBAC)*. Accessed: 2024-10-24. 2024. URL: https://kubernetes.io/docs/reference/access-authn-authz/rbac/ (cit. a p. 17).

[9] R. K. Das e D. K. Banerjee. *Real-Time Fine-Grained Access Control Policies for Kubernetes*. Accessed: 2024-10-24. 2020. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9298825 (cit. a p. 21).

[10] Kubernetes Documentation. *Node Authorizer*. Accessed: 2024-10-24. 2024. URL: https://kubernetes.io/docs/reference/access-authn-authz/node/ (cit. a p. 22).

[11] Red Hat. *Software Supply Chain Security on OpenShift with Kyverno and Cosign*. Accessed: 2024-10-24. 2024. URL: https://www.redhat.com/it/blog/software-supply-chain-security-on-openshift-with-kyverno-and-cosign (cit. a p. 24).

[12] Open Policy Agent. *Open Policy Agent Documentation*. Accessed: 2024-10-24. 2024. URL: https://www.openpolicyagent.org/docs/latest/ (cit. a p. 26).

[13] *Webhook Mode*. en. Section: docs. URL: https://kubernetes.io/docs/reference/access-authn-authz/webhook/ (visitato il giorno 26/10/2024) (cit. a p. 27).

[14] Badawekoo. *Make Kubernetes API Reachable from Anywhere*. Accessed: 2024-11-10. 2024. URL: https://medium.com/@badawekoo/make-kubernetes-api-reachable-from-anywhere-b7be128d8a0a (cit. a p. 35).

[15] Xueyuan Hu, Qiang Chen e Simon J.E. Taylor. «Automated Policy Refinement for Inter-Cloud Data Provenance». In: *Computers Security* 87 (2019). Accessed: 2024-11-19, p. 101588. URL: https://www.sciencedirect.com/science/article/abs/pii/S0167404818303870 (cit. a p. 54).