
UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento di
Ingegneria Gestionale, dell'Informazione e della Produzione

Corso di laurea in
Ingegneria Informatica

Classe n. L-8 - Classe delle lauree in Ingegneria dell'informazione

# Implementazione delle policy in Kubernetes con OPA

Candidato:
*Enrico Perani*

Matricola n.
1066174

Relatore:
*Chiar.mo Prof. Stefano Paraboschi*

Anno Accademico
2022/2023

**Abstract:**

Negli ultimi anni molte aziende hanno adottato tecnologie innovative e flessibili per garantire scalabilità ed efficienza dei propri sistemi IT. Tra queste tecnologie, Kubernetes è diventato uno degli strumenti di orchestrazione dei container più diffusi e utilizzati, grazie alla sua capacità di semplificare la gestione e la distribuzione delle applicazioni in ambienti complessi.

Gestire le politiche di sicurezza e di accesso ai dati in ambienti eterogenei può rappresentare una sfida complessa e impegnativa, per questo motivo, l'utilizzo di Open Policy Agent (OPA) come strumento unificato per la gestione e il testing delle policy rappresenta una soluzione vantaggiosa e innovativa per le aziende.

In questa tesi si analizzerà la validità di OPA come strumento di gestione delle policy e si illustrerà come quest'ultimo possa essere utilizzato come metodo unico di attuazione, gestione e test delle policy per diverse tecnologie, tra cui Kubernetes.

In particolare si mostrerà come scrivere e applicare le policy per Kubernetes utilizzando OPA e si illustreranno alcuni esempi concreti; saranno inoltre presentati i risultati attesi e le potenziali applicazioni future dell'integrazione di OPA e Kubernetes.

# Indice

# Elenco delle figure

# Elenco dei listati

# Capitolo 1

# Introduzione

## 1.1 Contesto generale

L'aumento della dipendenza dalle tecnologie digitali ha reso la sicurezza informatica un aspetto essenziale per garantire la riservatezza, l'integrità e la disponibilità dei dati e delle risorse informatiche.

La sicurezza informatica si riferisce alle politiche, alle misure e alle procedure adottate al fine di prevenire e mitigare le minacce informatiche. Si tratta di un campo multidisciplinare che comprende aspetti tecnici, organizzativi e comportamentali.

In questa ottica la gestione efficace dell'ambiente IT assume un ruolo fondamentale; pertanto è indispensabile adottare politiche atte a garantire il corretto funzionamento dei sistemi informativi.

La gestione di queste politiche richiede l'adozione di strategie, strumenti e processi adeguati. Open Policy Agent (OPA) sta emergendo come una soluzione sempre più diffusa e apprezzata per la gestione e il testing delle policy, soprattutto grazie alla sua architettura modulare e altamente personalizzabile.

## 1.2 Motivazioni e obiettivi della tesi

Le motivazioni che hanno portato alla scelta di questo tema riguardano principalmente l'importanza crescente che la gestione degli ambienti IT sta assumendo in vari settori. In un mondo sempre più interconnesso, dove le reti e i sistemi informatici svolgono un ruolo

centrale, la sicurezza e l'affidabilità di tali ambienti sono cruciali per garantire il corretto funzionamento delle organizzazioni e la protezione delle informazioni sensibili.

L'obiettivo di questa tesi è analizzare l'utilizzo di Open Policy Agent come strumento per migliorare la sicurezza e la gestione degli ambienti IT. Verrà esplorata l'applicazione di OPA concentrandosi in particolare sull'integrazione con Kubernetes, una delle principali piattaforme di orchestrazione dei container. Attraverso l'implementazione pratica delle policy, si valuterà l'efficacia di questa soluzione.

# Capitolo 2

# Open Policy Agent

## 2.1 Introduzione

OPA [1] è un progetto open-source creato originariamente da Styra e pubblicato nel 2016 [46]. OPA è un controllore e attuatore di politiche di sicurezza che disaccoppia le policy dal codice dell'applicazione. In questo modo, le policy possono essere aggiunte, modificate o cancellate senza dover riadattare il codice del software, rendendo quindi l'intero processo di sviluppo, gestione e test delle policy il più flessibile e dinamico possibile.

Le policy sono implementate mediante un linguaggio specifico, sviluppato appositamente per OPA, chiamato Rego (Regular Expression Generator for OPA). Rego è un linguaggio dichiarativo basato su JSON (JavaScript Object Notation) che permette di definire le policy in modo strutturato e leggibile per gli sviluppatori.

Utilizzando OPA è quindi possibile definire policy altamente personalizzabili in più ambienti. A oggi, Open Policy Agent presenta un ecosistema d'utilizzo estremamente vasto, si può infatti utilizzare OPA per definire policy in Kubernetes, Spring, AWS, Flask, Docker e moltissime altre tecnologie.

## 2.2 Policy

### 2.2.1 Introduzione al concetto di policy

Una policy, o politica, rappresenta 'un insieme di regole e direttive che definiscono come un sistema, un'applicazione o un'organizzazione debbano comportarsi in una determinata situazione [5]'. Le policy sono utilizzate per stabilire le regole d'utilizzo di risorse e informazioni all'interno di un'organizzazione, garantendo conformità alle normative, best practice o semplicemente alle regole interne di un'organizzazione.

In ambito informatico, le policy possono essere definite in vari livelli, dal livello di sistema al livello di applicazione, e possono includere policy di sicurezza, di accesso, di backup, di password, di sicurezza delle reti e molte altre.

Le policy devono essere scritte in modo chiaro e conciso, inoltre devono essere regolarmente aggiornate e documentate per garantire che siano sempre efficaci e che rispondano alle mutevoli necessità dell'impresa e degli utilizzatori. La definizione e l'implementazione di policy efficaci sono fondamentali per garantire la sicurezza e la protezione delle informazioni aziendali, dei clienti e degli utenti finali.

### 2.2.2 Best practice

Le best practice per la scrittura delle policy ricordano sotto molti aspetti le principali linee guida per la scrittura di codice utilizzando i 'classici' linguaggi di programmazione: questo rafforza ancora di più l'idea che OPA permetta la scrittura delle policy *as code*. Non si possono definire universalmente e con precisione queste linee guida ma è possibile citarne alcune:

- **utilizzare funzioni**: le funzioni possono semplificare la scrittura delle policy, specialmente per le condizioni ripetitive;

- **mantenere la semplicità**: le policy devono essere scritte in modo chiaro e semplice per facilitarne la comprensione e la manutenzione, evitando l'utilizzo di condizioni troppo complesse;

- **utilizzare nomi descrittivi**: nomi descrittivi aiutano a capire lo scopo della policy e semplificano la manutenzione;

- **testare**: tutte le policy dovrebbero essere regolarmente testate tenendo conto di tutte le casistiche possibili, per assicurarsi che funzionino come previsto e che non vi siano conflitti;

- **versioning**: si dovrebbe utilizzare un sistema di versionamento, come GitHub, per poter tenere traccia dello storico, facilitando, se necessario, eventuali rollback;

- **documentare**: le policy devono essere documentate per fornire informazioni sugli obiettivi e funzionamento;

- **monitorare**: si deve monitorare le policy per individuare eventuali violazioni o errori di configurazione.

### 2.2.3 Disaccoppiamento delle policy

Riprendendo la definizione di OPA (Sezione 2.1), si può notare come il disaccoppiamento delle policy dal codice sia estremamente rilevante e comporti numerosi vantaggi:

1. le policy possono essere condivise tra le applicazioni, indipendentemente dal linguaggio o dal framework utilizzato, portando una maggiore scalabilità anche in ambienti eterogenei;

2. il ciclo di vita della policy può essere tenuto separato rispetto a quello dell'applicazione, consentendo la distribuzione degli aggiornamenti senza dover ricostruire e ridistribuire l'applicazione o parte di essa;

3. le policy possono essere testate isolatamente. Poiché la politica è un codice, dovrebbe essere testata come qualsiasi altro software, aumentando la fiducia nella correttezza delle policy e risoluzione, rilevazioni di eventuali bug;

4. la comprensione delle policy da parte degli sviluppatori e degli utilizzatori risulta notevolmente semplificata.

## 2.3 Architettura di OPA

OPA è formato da diversi componenti che cooperano per fornire le funzionalità di valutazione delle policy. La sua architettura, come evidenziato nella Figura 2.1 si basa su tre componenti principali:

1. **regole** (policy rules): definiscono cosa è consentito e cosa non lo è all'interno del sistema;

2. **dati di input** (input data): sono le informazioni che vengono utilizzate per valutare le regole di politica. Questi dati possono provenire da diverse fonti e solitamente corrispondono a una richiesta di un utente o di un'applicazione esterna. OPA è in grado di accettare come input qualsiasi elemento in formato JSON;

3. **valutazione** (evaluation): è il processo in cui OPA utilizza le regole di politica e i dati di input per determinare se l'azione richiesta è consentita o meno. Se la richiesta viene accettata, OPA restituisce una risposta positiva 'allow'; in caso contrario viene restituita una risposta non definita 'undefined'.

### 2.3.1 Valutazione delle policy in OPA

OPA valuta le policy [2] utilizzando un processo noto come decision evaluation, che è un flusso mediante il quale restituisce un valore finale. Il processo può essere diviso nelle seguenti fasi (Figura 2.2):

1. **caricamento**: nella prima fase, OPA carica le policy;

2. **parsing**: analizza il documento di input e lo converte in una struttura dati interna;

3. **valutazione delle regole**: OPA valuta le regole in base a un ordine prestabilito;

4. **generazione del risultato**: risolve eventuali variabili utilizzate nella decisione e restituisce il risultato finale.

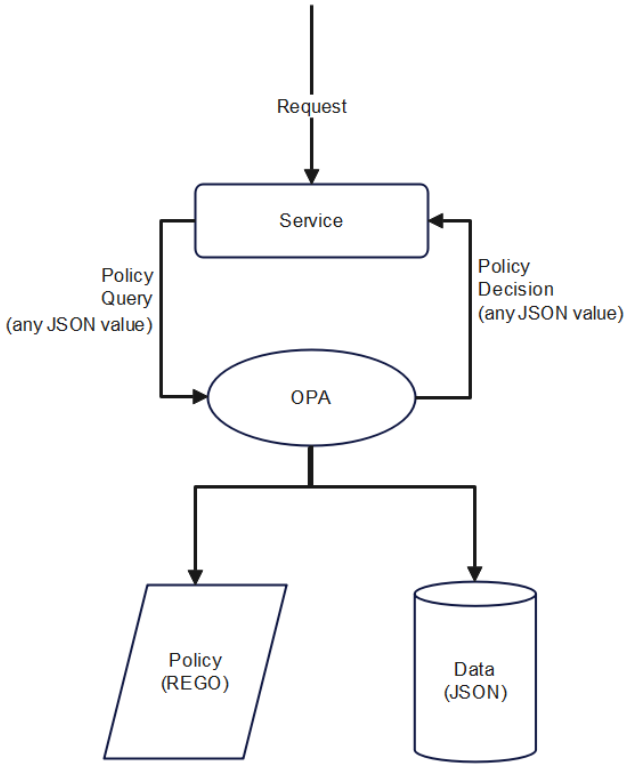


Figura 2.1: Architettura OPA [1].

### 2.3.2 Integrazione

Esistono diverse opzioni per integrare OPA con l'applicazione:

- se l'applicazione è stata creata utilizzando il linguaggio Go, è possibile usare direttamente OPA come libreria nell'applicazione;
- OPA può essere eseguito come un servizio autonomo (daemon);
- se l'applicazione è distribuita su Kubernetes, il servizio OPA può essere eseguito come 'container side-car' insieme ai servizi dell'applicazione. Questo riduce al minimo la latenza di comunicazione tra OPA e l'applicazione ed evita possibili problemi di comunicazione.

Sebbene esistano diverse possibilità per integrare OPA, la modalità side-car è ritenuta la più performante ed è diventata la modalità di integrazione più diffusa e utilizzata.

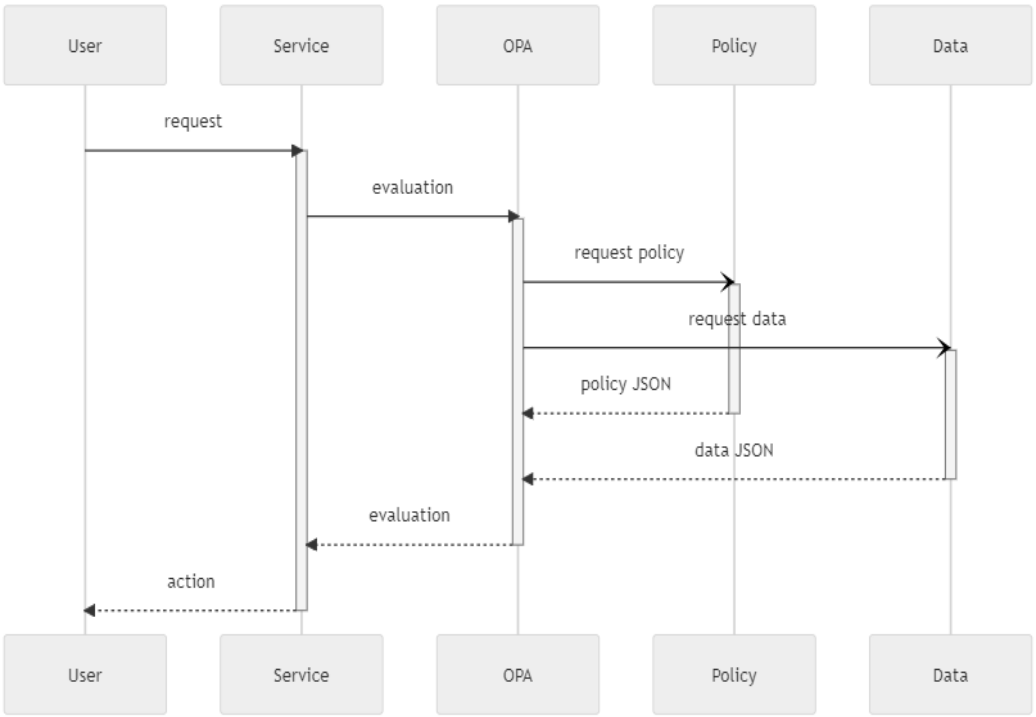


Figura 2.2: OPA Sequence Diagram.

## 2.4 Rego

### 2.4.1 Introduzione al linguaggio Rego

Rego [47] è un linguaggio dichiarativo di alto livello progettato per implementare regole rispetto dati strutturati come JSON. Rego consente di scrivere policy facilmente scalabili per svariati tipi di servizi. È possibile quindi valutare dati forniti come input e prendere decisioni di conseguenza.

A differenza di altri linguaggi di programmazione, Rego è un linguaggio dichiarativo per la scrittura di regole, simile a un linguaggio di query come SQL. Ciò significa che anziché scrivere codice per specificare come eseguire un'azione, si descrive semplicemente cosa deve essere fatto, consentendo agli utenti di concentrarsi sulla logica delle regole piuttosto che sulla loro implementazione.

Rego è altamente espressivo e consente la composizione modulare di regole complesse

semplificando la scrittura di policy di sicurezza personalizzate.

### 2.4.2 Struttura e sintassi di Rego

Essendo Rego un linguaggio dichiarativo di alto livello, le regole vengono scritte come funzioni che accettano input e restituiscono un valore specificato dal programmatore.

Un file Rego ha una struttura molto semplice: consiste in una serie di regole, definite come funzioni, ognuna delle quali può accettare uno o più input di diversi formati, tra cui stringhe, numeri, booleani od oggetti JSON. La struttura di base di una regola è composta da due parti principali:

- il test definisce le condizioni sotto cui la policy deve essere applicata;
- l'azione specifica l'operazione da intraprendere in caso in cui il test venga superato.

La sintassi di Rego può essere schematicamente così riassunta:

- **assegnamento**: è possibile assegnare a una variabile valori quali interi, stringhe, booleani, array, tipi object ecc. L'operatore utilizzato per l'assegnamento è **:=**;
- **riferimenti**: vengono utilizzati per accedere a documenti nidificati e sono in genere scritti utilizzando lo stile dot-access. La forma canonica **[]** può sostituire lo stile dot-access, ma il suo utilizzo è sconsigliato soprattutto per questioni di leggibilità;
- **funzioni**: sono l'equivalente delle subroutine nei linguaggi di programmazione. Possono essere chiamate con zero o più argomenti e restituire qualsiasi valore;
- **regole**: sono il cuore di una policy Rego. Sono costituite da una 'testa' e un 'corpo': la testa specifica nome e parametri della regola, mentre il corpo contiene le condizioni da soddisfare affinché la regola sia valida;
- **operatori**: sono utilizzati per combinare le condizioni delle regole e possono essere aritmetici, di confronto o logici (`and`, `or`, `not`);
- **costrutti di controllo di flusso** come `if/else`, `for`, `with`, `some`, `every`, `default`.

### 2.4.3 Esempio

Supponiamo ora, a fine di esempio, di avere un'azienda con tre categorie di lavoratori: manager, capi area e operai. Si vuole che ogni categoria abbia determinati livelli di accesso ai dati:

1. I manager devono avere il pieno accesso a tutti i dati aziendali. Questo significa che possono accedere a qualsiasi informazione e hanno il permesso di eseguire tutte le operazioni, inclusa la cancellazione dei dati.

2. I capi area, invece, hanno accesso limitato ai dati e possono visualizzare solo le informazioni relative alla loro area di competenza. Non hanno la possibilità di eseguire operazioni di cancellazione dei dati.

3. Gli operai, infine, possono consultare solo i dati relativi alla loro area di competenza, ma non possono modificarli né cancellarli. L'accesso è limitato alla lettura dei dati.

La policy che permette di realizzare ciò è visibile nel listato 2.1.

```
1  default allow = false

3  allow {
4      input.user.role == "manager"
5  }
6  allow {
7      input.user.role == "capo-area"
8      not input.user.method == "DELETE"
9      some i
10     input.user.areas[i] == input.data.area

12 }
13 allow {
14     input.user.role == "operaio"
15     input.user.method == "GET"
16   some i
17     input.user.areas[i] == input.data.area
18 }
```

Listato 2.1: Esempio policy in Rego

# Capitolo 3

# Kubernetes

## 3.1 Introduzione, storia e successo

### 3.1.1 Introduzione

Kubernetes [27] (o k8s) è una piattaforma portatile, estensibile e open-source per la gestione di carichi di lavoro e servizi containerizzati, in grado di facilitare la configurazione e l'automazione. Il suo scopo principale è semplificare la gestione di applicazioni containerizzate in ambienti distribuiti, consentendo agli sviluppatori di creare, distribuire e gestire le loro applicazioni in modo efficiente e scalabile. Un cluster Kubernetes deve essere:

- **sicuro**: deve rispettare le best practice più aggiornate in ambito di sicurezza;
- **facile da usare**: deve poter funzionare con pochi e semplici comandi;
- **estendibile**: deve essere personalizzabile a partire da un file di configurazione.

Grazie a Kubernetes, gli sviluppatori possono concentrarsi sulla creazione di applicazioni, mentre la piattaforma si occupa della gestione delle risorse e delle dipendenze di cui le applicazioni necessitano.

### 3.1.2 Storia e motivi del suo successo

Kubernetes è stato originariamente sviluppato da Google nel 2014 sulla base del loro sistema interno di gestione dei Container, noto come Borg [50], dal 2015 il progetto è stato reso open-source.

Il successo di Kubernetes è attribuibile principalmente alla sua architettura modulare e altamente scalabile, che consente di gestire in modo efficiente applicazioni containerizzate su diverse infrastrutture, come ambienti cloud, on-premises e ibridi.

La vasta comunità che si è formata attorno a Kubernetes ha contribuito allo sviluppo di numerosi strumenti e plug-in, rendendolo la piattaforma di gestione dei container più diffusa a livello mondiale. Inoltre, la crescente adozione dei container come tecnologia di virtualizzazione ha favorito la diffusione di Kubernetes, poiché consente di isolare le applicazioni in modo efficiente e di eseguirle su qualsiasi infrastruttura disponibile.

### 3.1.3 Principali funzionalità

Tra le principali funzionalità di Kubernetes [29] possiamo trovare:

- **ottimizzazione dei carichi**: Kubernetes consente di specificare le risorse necessarie per l'esecuzione di ogni Container e alloca di conseguenza i Container sui nodi per ottimizzare l'uso delle risorse;
- **scoperta dei servizi e bilanciamento del carico**: con Kubernetes è possibile esporre un Container mediante il suo IP o tramite un nome DNS. Kubernetes distribuisce il traffico su più Container per garantire la stabilità del servizio;
- **orchestrazione dello storage**: Kubernetes può montare automaticamente un sistema di storage scelto dall'utente;
- **rollout e rollback automatizzati**: durante la messa in produzione del Container, Kubernetes può raggiungere lo stato desiderato in modo automatico; inoltre, è possibile ripristinare versioni precedenti in caso di malfunzionamento;
- **sicurezza**: Kubernetes offre molteplici funzionalità di sicurezza, tra cui la gestione dei certificati TLS, l'autenticazione e l'autorizzazione basate su ruoli e la gestione centralizzata delle credenziali;

- **self-healing**: nel caso in cui un Container si blocchi o non risponda più, Kubernetes si occuperà di riavviarlo, sostituirlo o terminarlo;
- **volume management**: Kubernetes supporta diversi tipi di volumi per gestire i dati dell'applicazione in modo affidabile e scalabile.

## 3.2 Architettura

L'architettura di Kubernetes è composta da un set di componenti interconnessi tra loro [30], ciascuno dei quali svolge un ruolo specifico nella gestione dei Container. Come illustrato nella Figura 3.1, possiamo dividere questi componenti in due categorie:

1. **control plane**: il componente responsabile della gestione e dell'orchestrazione dei nodi del cluster;
2. **node**: fornisce risorse computazionali per l'esecuzione delle applicazioni.

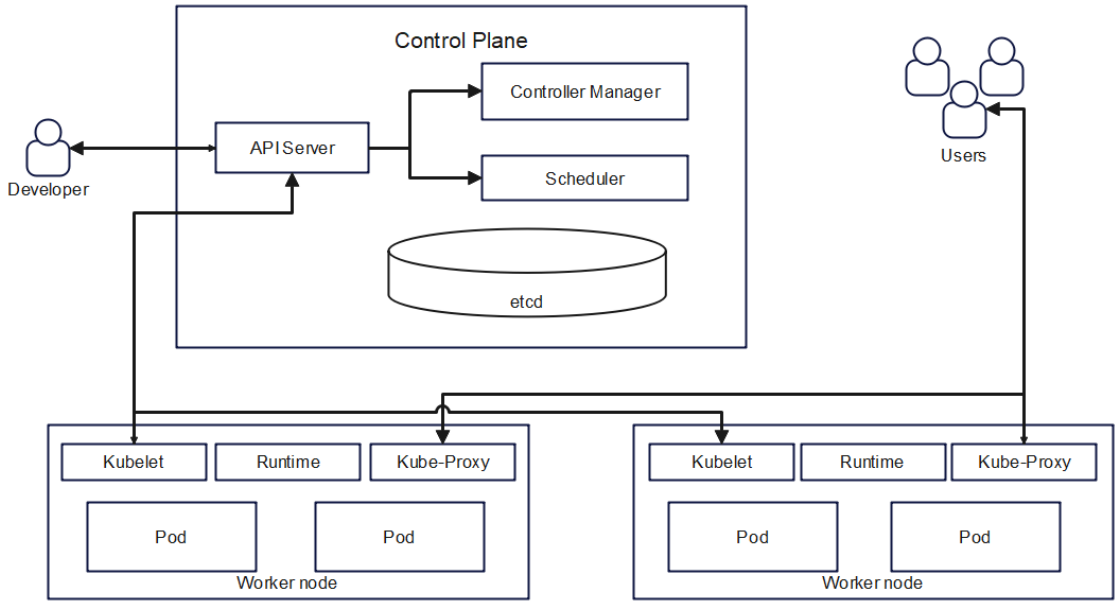


Figura 3.1: Architettura Kubernetes [30].

Il control plane di Kubernetes è il componente centrale che controlla l'intero cluster e gestisce le operazioni di orchestrazione del carico di lavoro ed è responsabile di corretto

mantenimento dello stato del cluster. Il control plane può essere eseguito su uno o più node, che prendono il nome di Master node mentre i node che forniscono risorse computazionali vengono chiamati nodi worker. Una buona pratica è di eseguire il control plane su più Master node per garantire un'alta disponibilità e resistenza agli errori [26].

Il control plane è formato da diversi componenti:

1. **kube-apiserver**: è il componente che espone l'API di Kubernetes. Tutti componenti del cluster, per poter comunicare, devono connettersi con l'API server;

2. **kube-scheduler**: è il componente che decide su quale node avviare i nuovi carichi di lavoro. Basandosi sulle richieste di risorse e sulle preferenze specificate, sceglie il nodo worker ottimale su cui avviare il lavoro. Se un node è occupato o ha la maggior parte delle risorse già occupate, invierà la richiesta a un node con più risorse disponibili;

3. **kube-controller-manager**: gestisce i controlli di background del cluster, garantendo che lo stato del cluster sia sempre ottimale. Ad esempio, se un node 'muore', il controller manager deve individuarlo e provvedere al ripristino nel minor tempo possibile;

4. **etcd**: consiste in un datastore distribuito utilizzato per memorizzare lo stato del cluster. Tutti i componenti del control plane utilizzano etcd per memorizzare lo stato del cluster e le informazioni sulle risorse, tracciando così di ogni modifica, anche colposa, del cluster;

5. **cloud-controller-manager**: è una componente opzionale che permette di collegare il cluster al cloud e separa i componenti che interagiscono internamente al cluster dai componenti che interagiscono con la piattaforma.

Un node è una macchina virtuale o un server fisico che fornisce risorse computazionali (CPU, memoria, storage) per poter eseguire i Container delle applicazioni. I node sono le entità che svolgono il lavoro richiesto dal cluster, per questa ragione sono conosciuti anche come 'worker node'.

Un node è identificato da un nome univoco all'interno del cluster e viene registrato presso il control plane permettendo la comunicazione. Ciascun node compie una serie di

processi e servizi per gestire i Container e le applicazioni. A tale scopo, è necessario che nel node siano presenti tre componenti:

1. **Runtime**: è il software che esegue i Container, come Docker;
2. **Kubelet**: è un agente eseguito su ogni node che, comunicando con il control plane, assicura la corretta configurazione ed esecuzione dei Container;
3. **kube-proxy**: è un servizio di rete che, mantenendo le regole di rete sul node, permette la comunicazione dei node e dei suoi componenti interna o esterna al cluster.

## 3.3 Kubernetes Objects

### 3.3.1 Introduzione agli oggetti

Gli oggetti [33] in Kubernetes sono delle entità persistenti nel cluster che rappresentano le risorse fondamentali utilizzate per definire, configurare e gestire l'intera infrastruttura. Un oggetto Kubernetes ha un proprio schema definito che ne illustra il comportamento e le proprietà.

Gli oggetti Kubernetes sono dichiarativi: il che significa che gli utenti non hanno bisogno di conoscere i dettagli tecnici del loro funzionamento, ma possono semplicemente specificare l'oggetto e le proprietà desiderate e il control plane di Kubernetes si occuperà di gestire l'implementazione e la configurazione. I principali oggetti sono:

- **Pod** [35]: sono l'unità di base dell'orchestrazione dei Container. Un Pod è immutabile, il che significa che non è possibile modificarne lo stato o le caratteristiche una volta creato. Tuttavia, possono invece essere sostituiti con nuove istanze create o distrutte in modo dinamico;
- **Namespace** [32]: è un'unità logica di isolamento utilizzata per creare più partizioni virtuali all'interno di un cluster (Figura 3.2). Ogni Namespace fornisce una visibilità isolata all'interno del cluster. In Kubernetes, esistono quattro Namespace predefiniti: default, kube-node-lease, kube-public e kube-system, quest'ultimo contiene gli oggetti di sistema;

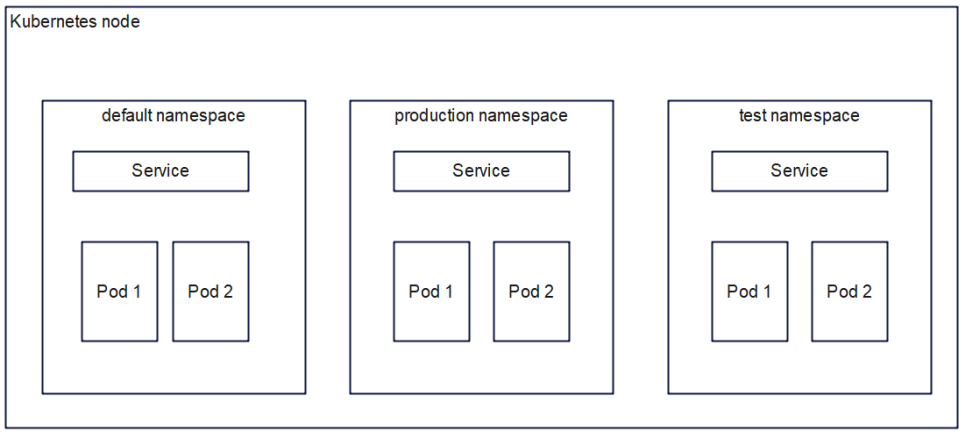


Figura 3.2: Namespace Kubernetes.

- **Deployment** [28]: è un oggetto che definisce come i Pod devono essere distribuiti e aggiornati all'interno del cluster. Fornisce un'astrazione per la gestione delle repliche, consentendo di specificare uno stato desiderato e gestire facilmente il processo di aggiornamento dell'applicazione.

  Kubernetes fornisce anche la possibilità di eseguire il rollback degli aggiornamenti del Deployment in caso di errori, ripristinando facilmente l'applicazione alla versione precedente garantendo disponibilità e continuità del servizio;

- **Service** [38]: è un oggetto che definisce un insieme di Pod e una politica di accesso a essi. Come illustrato dalla Figura 3.3, il Service viene utilizzato per fornire un'interfaccia di rete stabile per i Pod. In altre parole, espone i Pod come un set di endpoint unificati, consentendo agli altri componenti di interagire in modo affidabile;

- **ConfigMap** [25]: è un oggetto che consente di memorizzare e gestire in maniera centralizzata la configurazione degli oggetti. Il valore principale di un ConfigMap è una mappa di coppie chiave-valore che possono essere utilizzate dalle applicazioni;

- **Secret** [37]: è una risorsa utilizzata per gestire in maniera sicura le informazioni sensibili, come le password o le chiavi di autenticazione. Secret è simile a un ConfigMap, ma invece di una mappa di coppie chiave-valore, contiene coppie di dati crittografati;

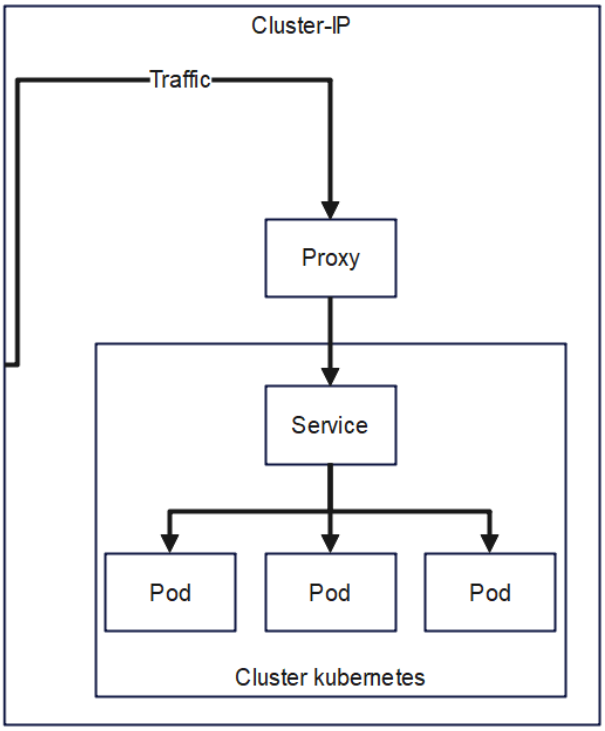


Figura 3.3: Service Kubernetes.

- **Volume** [39]: sono oggetti che consentono la memorizzazione dei dati. Possono essere visti come un'unità di storage in cui i Container possono scrivere e leggere i dati. I volumi consentono ai dati di persistere durante il ciclo di vita di un Pod, anche quando si verificano arresti o riavvii.

### 3.3.2 Gestione degli oggetti

La gestione degli oggetti in Kubernetes comporta diverse operazioni che si riassumono nell'acronimo CRUD (Create Read Update Delete); l'utente può utilizzare l'interfaccia della riga di comando kubectl per eseguire queste operazioni.

Per definire e gestire gli oggetti si utilizza il formato YAML [51] (YAML Ain't Markup Language): un formato di serializzazione di dati leggibile dall'uomo che viene comunemente utilizzato per la scrittura dei file di configurazione.

La struttura di YAML (listato 3.1) è determinata dall'indentazione per definire le strutture gerarchiche dei dati, ovvero zero o più caratteri spazio all'inizio di una riga.

Le indentazioni vengono quindi utilizzate per definire la relazione tra i campi.

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: my-pod
5 spec:
6   containers:
7   - name: my-container
8     image: nginx
```

Listato 3.1: Esempio file YAML

Per creare un Deployment chiamato 'myapp-deployment' si può utilizzare il seguente comando:

```
kubectl apply -f myapp-deployment.yaml
```

Per modificare un oggetto in Kubernetes si utilizza lo stesso comando, ma il file YAML deve contenere le modifiche che si vogliono apportare. Per eliminare il Deployment precedentemente creato si utilizza il seguente comando:

```
kubectl delete -f myapp-deployment.yaml
```

In generale, è consigliabile verificare lo stato degli oggetti presenti nel cluster prima di modificarli per evitare eventuali problemi. L'eliminazione di un oggetto in Kubernetes comporta anche l'eliminazione di tutti gli oggetti dipendenti da esso. Ad esempio, se si elimina un Deployment, saranno eliminati anche i Pod associati ad esso.

## 3.4 Best practice e policy

### 3.4.1 Best practice

Similmente a OPA, come descritto nella Sezione 2.2, anche in Kubernetes è fondamentale adottare una combinazione efficace di best practice e policy. Infatti, la gestione di un'infrastruttura Kubernetes richiede attenzione e competenze, poiché errori di configurazione possono causare gravi conseguenze sull'intero sistema. Certificare la sicurezza delle applicazioni e proteggere le risorse del cluster sono solo alcune delle principali preoccupazioni per gli amministratori di Kubernetes. Per garantire un cluster Kubernetes ben configurato e performante, è fondamentale seguire alcune best practice [23, 3, 43]:

1. aggiornare regolarmente le versioni di Kubernetes e delle immagini nei Container, per poter beneficiare delle nuove funzionalità e correzioni di bug;

2. utilizzare le immagini Docker ufficiali o controllate per evitare problemi di sicurezza, utilizzare inoltre strumenti di scansione delle vulnerabilità per verificare la sicurezza delle immagini;

3. evitare di utilizzare privilegi elevati nelle applicazioni o nei Container, utilizzare invece il principio del modello minimo di privilegi per ridurre i rischi di sicurezza;

4. limitare le risorse disponibili per ogni Pod;

5. utilizzare i Namespace per isolare correttamente le risorse Kubernetes;

6. utilizzare l'autenticazione e l'autorizzazione per controllare e gestire l'accesso alle risorse del cluster.

### 3.4.2 Esempi

#### 3.4.2.1 Sovra utilizzo delle risorse del cluster

Un problema comune in Kubernetes è il sovra utilizzo delle risorse del cluster (best practice n. 4).

Alcune applicazioni possono monopolizzare una quantità sproporzionata di CPU e memoria causando prestazioni lente, instabilità del cluster e una cattiva distribuzione delle risorse tra le applicazioni.

**Best practice** Una best practice per affrontare questo problema è l'imposizione di limiti di risorse sui Pod utilizzando le LimitRange [31]. Questo consente di specificare i limiti di CPU e memoria che ogni Pod può utilizzare, evitando che le applicazioni consumino risorse in modo eccessivo.

**Policy** Un'implementazione personalizzata di questa best practice potrebbe includere l'adozione di una policy specifica per l'allocazione delle risorse. Ad esempio, potrebbe essere definita una policy aziendale che stabilisce limiti di CPU e memoria specifici per i diversi gruppi o progetti. In questo modo si può evitare il sovra utilizzo delle risorse.

#### 3.4.2.2 Gestione del versionamento delle immagini

Il secondo problema tratta la gestione delle versioni delle immagini utilizzate per i Container (best practice n. 1).

Quando le immagini non vengono aggiornate regolarmente si possono verificare problemi di sicurezza, mancata correzione di bug. Inoltre, l'utilizzo di immagini obsolete può impedire di beneficiare delle nuove funzionalità offerte dalle versioni più recenti.

**Best practice** Una best practice standardizzata per affrontare questo problema è mantenere le immagini dei Container aggiornate regolarmente. Ciò implica di monitorare le versioni delle immagini utilizzate, verificare la presenza di aggiornamenti e applicare le nuove versioni in modo tempestivo. È possibile utilizzare strumenti di automazione per semplificare il processo di aggiornamento delle immagini e garantire che il cluster Kubernetes utilizzi sempre le versioni più recenti e sicure delle immagini.

**Policy** Una policy personalizzata potrebbe prevedere l'adozione di regole specifiche per la gestione delle versioni delle immagini. Ad esempio, potrebbe essere definita una policy che richiede l'aggiornamento delle immagini entro un determinato periodo di tempo, come ad esempio ogni mese o a ogni nuova release. Oppure potrebbe essere creata una policy che imponga un determinato metodo di versionamento delle immagini attraverso l'utilizzo di specifici tag o impedendo l'uso di tag generici.

Questo assicura che la gestione delle versioni delle immagini sia disciplinata e che le applicazioni utilizzino sempre le versioni più recenti e sicure.

# Capitolo 4

# Tecnologie utilizzate

Il successo nella creazione di un progetto basato sul cluster Kubernetes dipende in gran parte dalle tecnologie utilizzate per implementarlo. In questo capitolo verranno introdotte le tecnologie presenti nel caso di studio e verrà illustrato il loro ruolo all'interno del sistema.

## 4.1 Docker

Docker [7] è una tecnologia di containerizzazione che ha rivoluzionato il modo in cui le applicazioni vengono sviluppate, distribuite e gestite. Docker è stato rilasciato nel 2013 [45] come progetto open-source e da allora ha avuto un'ampia diffusione in tutto il mondo.

Il concetto di containerizzazione non era nuovo ma Docker ha reso l'uso dei Container più facile e accessibile fornendo un'interfaccia semplice e intuitiva per la creazione, l'avvio e la gestione dei Container e semplificando il processo di distribuzione delle applicazioni su più ambienti.

### 4.1.1 Architettura

Docker utilizza un'architettura client-server per consentire la creazione e la gestione dei Container. Come evidenziato dalla Figura 4.1, l'architettura di Docker è formata da tre componenti principali:

- **Docker daemon**: è un processo in esecuzione sul sistema host che gestisce gli oggetti Docker. Il daemon ascolta le richieste API e si occupa della creazione, dell'avvio e della terminazione degli oggetti come Container, immagini, volumi e reti;
- **Client Docker**: è l'interfaccia tramite la quale gli utenti interagiscono con Docker. Il Client invia comandi al daemon utilizzando l'API Docker e può essere eseguito sulla stessa macchina in cui è in esecuzione il daemon o su una macchina remota;
- **Docker Registry**: è un registro centralizzato in cui vengono archiviate le immagini Docker. Docker è configurato per cercare le immagini su Docker Hub, che è un registro pubblico, ma è anche possibile avere un proprio repository privato.

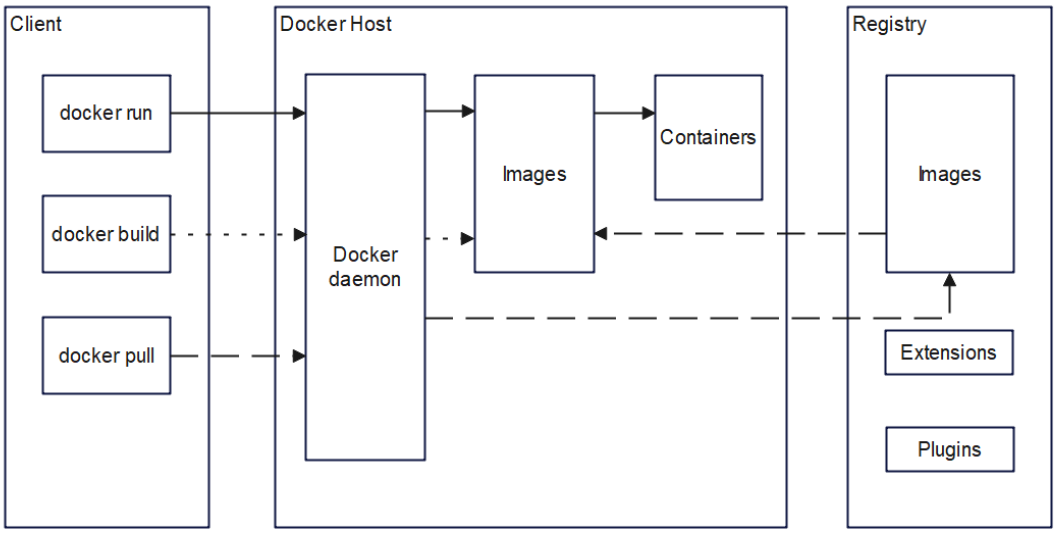


Figura 4.1: Architettura Docker [7].

### 4.1.2 Oggetti Docker

Docker offre diversi oggetti che consentono di gestire in modo efficiente le applicazioni containerizzate. I principali sono:

- **Image** [7]: è un modello (template) immutabile utilizzato per la creazione dei Container. Una Image contiene tutte le informazioni e le dipendenze necessarie per

eseguire un Container comprese librerie, codice sorgente, dipendenze e configurazioni;

- **Container** [7]: è un'istanza in esecuzione di Image Docker. Un Container (Figura 4.2) diversamente dalle Image utilizza le risorse di sistema e rappresenta un'entità isolata che può essere avviata, fermata, spostata o eliminata in modo indipendente;

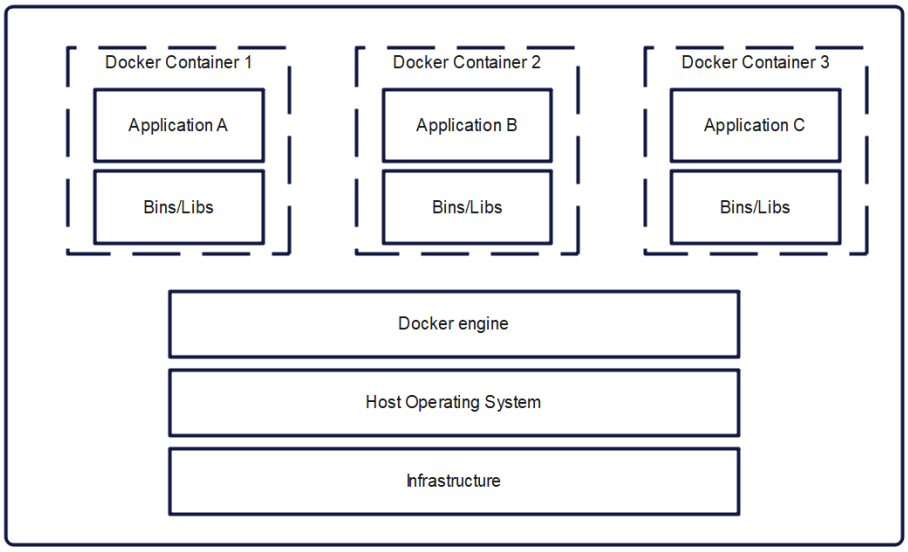


Figura 4.2: Docker Containers.

- **Volume** [11]: sono meccanismi per la gestione dei dati persistenti utilizzati dai Container. Un Volume è uno spazio di archiviazione che può essere condiviso tra i Container o persistere anche dopo la rimozione del Container stesso. I Volume consentono di separare i dati dall'ambiente di esecuzione facilitando il backup, il ripristino e lo scambio dei dati;

- **Network** [10]: consentono la comunicazione tra i Container e l'ambiente esterno. Docker fornisce una gestione semplice e flessibile dei Network infatti è possibile creare reti personalizzate e definire le politiche di comunicazione.

### 4.1.3 Dockerfile

Un Dockerfile [6] è un file, in formato testo, contenente una serie di istruzioni utilizzate per costruire una Image. Le istruzioni (listato 4.1) specificano le azioni da eseguire durante

la creazione della Image come l'installazione di pacchetti, la configurazione dell'ambiente e l'esecuzione di comandi.

```
1  FROM python:3
2  ENV PYTHONUNBUFFERED 1
3  RUN mkdir /app
4  WORKDIR /app
5  COPY requirements.txt /app
6  RUN pip install --upgrade pip
7  RUN pip install -r requirements.txt
8  COPY . /app
9  EXPOSE 5000
10 CMD [ "python", "app.py" ]
```

Listato 4.1: Esempio di Dockerfile.

Le istruzioni principali [9] sono:

- **FROM**: è un'istruzione obbligatoria e dovrà anche essere la prima istruzione del Dockerfile. Specifica l'immagine di base da cui partire per costruire un'immagine personalizzata;
- **ENV**: permette di poter definire delle variabili d'ambiente che verranno utilizzate all'interno del Container;
- **RUN**: esegue un'istruzione o un comando durante la fase di creazione dell'Image Docker;
- **COPY**: permette di poter copiare dei file locali all'interno dell'immagine in una posizione che dovrà essere specificata dallo sviluppatore;
- **EXPOSE**: permette di dichiarare quali porte sono raggiungibili dall'esterno;
- **CMD**: permette l'esecuzione dei comandi di shell subito dopo che il Container viene avviato.

Una volta creato il Dockerfile è possibile creare una Image Docker utilizzando il comando:

```
docker build "dockerfile-name"
```

### 4.1.4 Vantaggi

Docker è diventato uno degli strumenti più importanti per i team di sviluppo software, i responsabili delle operazioni IT e i sistemisti, poiché offre numerosi vantaggi [49], tra cui:

1. **portabilità**: la tecnologia Docker permette di creare e distribuire applicazioni e poterle eseguire su qualsiasi dispositivo;

2. **isolamento**: ogni Container ha risorse isolate, questo permette di aggiungere, modificare o rimuovere le applicazioni senza problemi causati da dipendenze. Se non è più necessaria un'applicazione è sufficiente eliminare il suo Container con la certezza che non verranno lasciati file temporanei o di configurazione nel sistema operativo;

3. **scalabilità**: i Container sono altamente scalabili e possono essere espansi in modo relativamente semplice;

4. **flessibilità**: Docker consente di utilizzare qualsiasi linguaggio di programmazione, eliminando il problema dell'incompatibilità tra librerie e tecnologie diverse;

5. **basso consumo di risorse**: rispetto alle macchine virtuali, i Container sono molto più efficienti in termini di utilizzo risorse perché non è necessario installare un sistema operativo completo su ogni Container.

## 4.2 Cluster locale

La gestione di un cluster Kubernetes può risultare molto complessa soprattutto a causa dei molteplici componenti e delle interazioni tra loro. Per disporre di un ambiente semplice è possibile utilizzare un cluster locale [48].

Un cluster locale è un'istanza di Kubernetes eseguita all'interno della propria macchina senza richiedere l'accesso a una piattaforma cloud o un server dedicato. Dunque è possibile creare un ambiente di sviluppo e test simile a quello che verrà utilizzato in produzione, senza dover affrontare costi elevati o complessi requisiti di configurazione.

Il cluster locale presenta alcune differenze rispetto a un cluster Kubernetes normale in quanto solitamente è composto da un singolo nodo che esegue tutti i servizi di controllo e di lavoro. Non è quindi possibile distribuire i carichi di lavoro su nodi diversi, il che può comportare problemi di prestazioni in caso di carichi di lavoro elevati.

Esistono diverse opzioni per creare un cluster Kubernetes locale, tra la moltitudine di tecnologie spiccano due soluzioni:

- **minikube**
- **kind**

### 4.2.1 minikube

`minikube` [40] è lo strumento più utilizzato per creare un cluster Kubernetes locale su una singola macchina. Permette di sperimentare e testare le applicazioni Kubernetes in un ambiente sicuro e isolato.

`minikube` è stato progettato per essere utilizzato come macchina virtuale sul computer dell'utente e distribuisce al suo interno un cluster a nodo singolo, è un'opzione eccellente per avere un semplice cluster Kubernetes attivo e funzionante su localhost. Si possono eseguire tutte le operazioni di gestione del cluster come la creazione di oggetti Kubernetes e la configurazione del networking direttamente dalla macchina locale.

### 4.2.2 kind

`kind` [20] è un'altra valida opzione per creare un cluster Kubernetes locale, uno dei principali vantaggi di `kind` rispetto a `minikube` è che è più veloce e leggero. `kind` utilizza le immagini Docker per creare i nodi del cluster ma, al contrario di `minikube`, non supporta l'esecuzione di Container non Docker.

Una funzionalità molto interessante di `kind` è la possibilità di poter caricare le immagini locali, presenti in docker-hub, direttamente nel cluster. Questa procedura consente di evitare diversi passaggi, con un semplice comando:

```
kind load docker-image my-app:latest
```

L'immagine 'my-app' è disponibile per l'uso nel cluster. `kind` inoltre si differenzia dai competitor per l'architettura del cluster multi-nodo.

In questo progetto di tesi si è scelto di utilizzare `kind` per la creazione del cluster locale poiché la maggior parte del lavoro è incentrato sulla creazione di Constraint e Template Constraint. `kind` ha dimostrato di essere una soluzione più veloce e leggera rispetto a `minikube`.

## 4.3 OPA Gatekeeper

### 4.3.1 Introduzione

OPA Gatekeeper [12, 42] (o Gatekeeper) è uno strumento open-source che fornisce un controllo di conformità su oggetti Kubernetes (Figura 4.3). Gatekeeper ha l'obiettivo di semplificare la gestione delle policy garantendo che gli oggetti Kubernetes presenti nel cluster siano conformi alle politiche di sicurezza, alle best practice e alle normative dell'organizzazione. Gatekeeper si occupa della conformità del cluster permettendo a sviluppatori e amministratori di concentrarsi sulla creazione di applicazioni e sulla gestione dell'infrastruttura.

Il progetto OPA Gatekeeper è stato avviato nel 2018 da Styra con il proposito di creare uno strumento per applicare le policy in ambienti Kubernetes. Inizialmente il progetto si basava su OPA, successivamente il progetto è cresciuto e sono stati introdotti diversi miglioramenti e nuove funzionalità, a oggi è diventato un progetto Cloud Native Computing Foundation (CNCF), riconoscendo di fatto la sua importanza per la comunità Kubernetes.

Gatekeeper, come appena anticipato, ha subito diverse modifiche nel tempo [34]. Gli stadi intermedi possono essere così riassunti:

1. **Gatekeeper v1.0**: utilizza OPA come Admission Controller con sidecar kube-mgmt che applica criteri basati su ConfigMap, fornisce funzioni di convalida e modifica;

2. **Gatekeeper v2.0**: simile al suo predecessore, oltre che fornire funzioni di convalida e modifica, aggiunge la funzionalità di audit;

3. **Gatekeeper v3.0**: l'Admission Controller è integrato nel Framework di OPA Constraint, fornisce funzionalità di convalida e, eventualmente, del controllo di ammis-

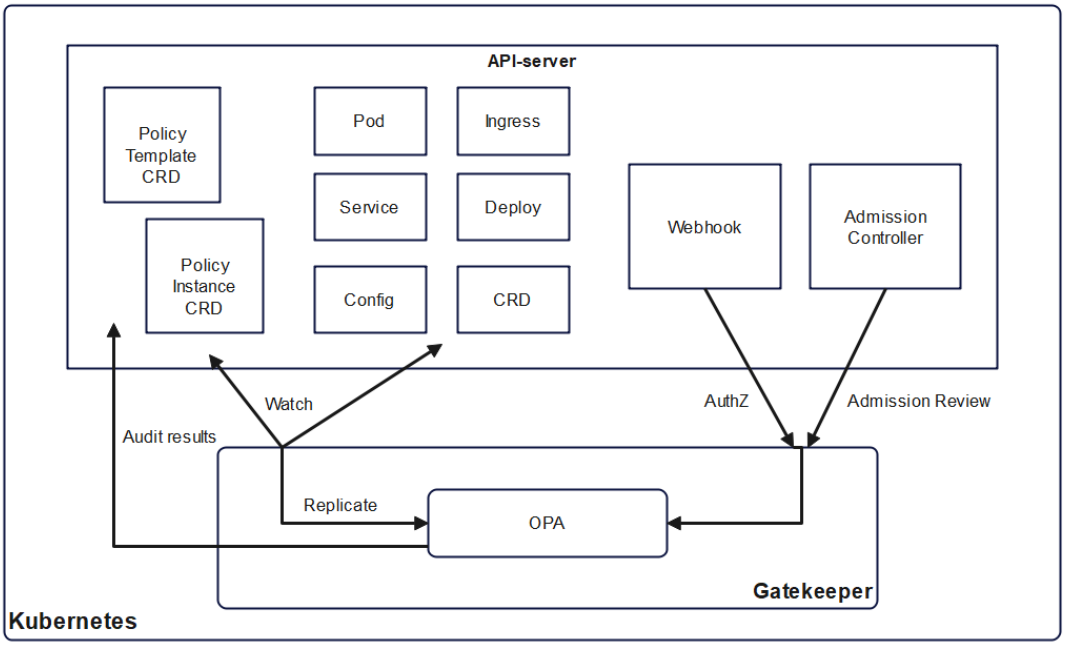


Figura 4.3: OPA in Kubernetes [34].

sione e della funzionalità di audit. Consente una notevole personalizzazione delle politiche mediante il linguaggio Rego; il progetto è nato da una collaborazione tra Google, Microsoft, Red Hat e Styra.

### 4.3.2 Admission Controller

In Kubernetes, gli Admission Controller [22] sono meccanismi di controllo utilizzati per modificare il comportamento predefinito di Kubernetes API Server. Sono impiegati per validare o respingere le richieste fatte ai server API di Kubernetes.

OPA Gatekeeper include un Admission Controller personalizzabile (Figura 4.4) che può essere utilizzato per applicare policy su qualsiasi oggetto Kubernetes. Gatekeeper garantisce che tutti gli oggetti rispettino le policy effettuando un controllo preventivo durante la loro creazione, migliorando significativamente la sicurezza dell'ambiente Kubernetes.

L'Admission Controller di OPA Gatekeeper è integrato direttamente nel ciclo di vita di Kubernetes. Ogni volta che un utente crea, modifica o elimina un oggetto la richiesta viene inviata al Controller di Gatekeeper. Per valutare le richieste l'Admission Controller interroga OPA il quale esegue le verifiche e valuta le policy utilizzando l'oggetto Kuber-

netes come input; infine, OPA comunica il risultato ottenuto al Controller che provvederà ad accettare o rifiutare la richiesta.

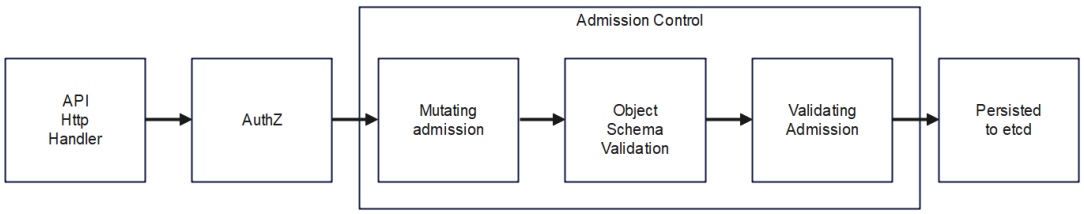


Figura 4.4: OPA Gatekeeper Admission Controller.

### 4.3.3 Constraint Template e Constraint

Per poter valutare le policy in OPA utilizzando un oggetto Kubernetes come input, come illustrato nella Figura 4.5, Gatekeeper introduce due Custom Resource Definition (CRD) [16], ovvero due risorse:

- Constraint Template;
- Constraint.

**Constraint Template** Prima di poter definire un Constraint, è necessario prima definire un Constraint Template [13, 14]. Il Constraint Template, o modello di vincolo, permette la creazione di modelli generici che definiscono come una policy deve essere applicata agli oggetti Kubernetes.

I Constraint Template sono scritti in formato YAML ma la logica al loro interno è scritta in Rego. Esistono svariati modelli predefiniti disponibili nella libreria Gatekeeper [17] che coprono molte delle regole comuni per la sicurezza e la conformità, l'utilizzo di questi Template standardizzati è altamente raccomandato soprattutto a causa della loro eccessiva verbosità.

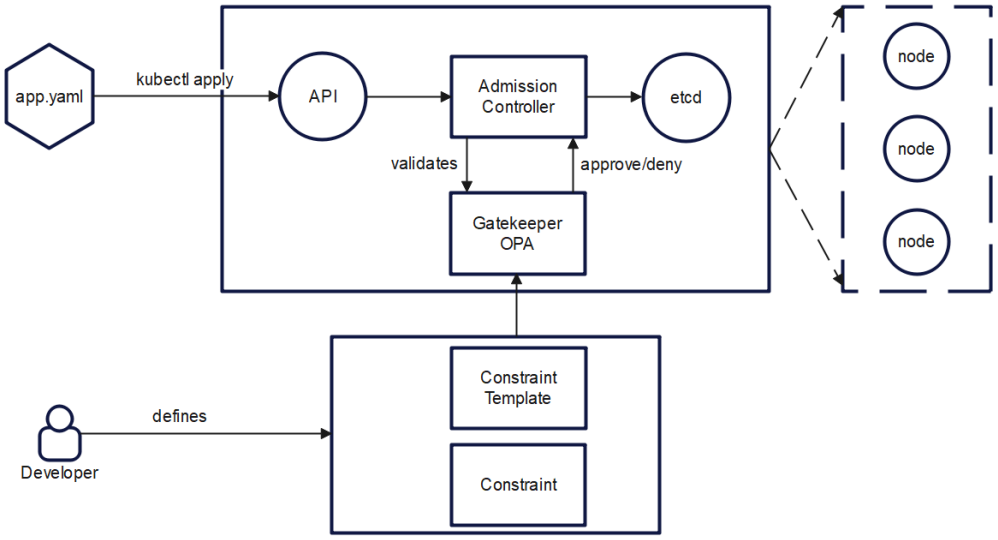


Figura 4.5: OPA Gatekeeper architecture.

**Constraint** Un Constraint, o vincolo, è un'istanza specifica di un Constraint Template e rappresenta una policy per un determinato oggetto Kubernetes. Il Constraint viene applicato agli oggetti per garantire che soddisfino determinate regole e requisiti definiti nel Constraint Template corrispondente. Ogni Constraint è scritto in formato YAML ed è costituito da un nome univoco e da un insieme di parametri che definiscono le configurazioni specifiche da applicare all'oggetto Kubernetes.

### 4.3.4 Esempio

Supponiamo ora di voler implementare una policy con lo scopo di rendere obbligatorio la presenza di un'etichetta (labels) chiamata risorse per tutti i Namespace presenti nel cluster.

Il Template (listato 4.2) deve essere costruito in maniera tale da poter essere applicato a qualsiasi oggetto Kubernetes. In questo modo è possibile astrarre il Template e lasciare che sia il Constraint a specificare quali risorse debbano rispettare il vincolo.

```
1  apiVersion: templates.gatekeeper.sh/v1beta1
2  kind: ConstraintTemplate
3  metadata:
4    name: clusterrequiredlabels
5    annotations:
6     description: "labels required for objects"
7  spec:
8    crd:
9      spec:
10       names:
11         kind: ClusterRequiredLabels
12       validation:

14         openAPIV3Schema:
15           properties:
16             labels:
17               type: array
18               items:
19                 type: string
20   targets:
21     - target: admission.k8s.gatekeeper.sh
22       rego: |
23         package clusterrequiredlabels

25         violation[{"msg": msg, "details": {"missing_labels": missing}}] {
26           provided := {label | input.review.object.metadata.labels[label]}
27           required := {label | label := input.parameters.labels[_]}
28           missing := required - provided
29           count(missing) > 0
30           msg := sprintf('devi provvedere ad specificare le seguenti
     etichette:%v', [missing])
31         }
```

Listato 4.2: Constraint Template: labels required

Il Template sopra definito è composto da tre parti [4]:

1. la prima sezione (righe 1-2) specifica la CRD Kubernetes, è un componente obbligatorio;

2. la seconda parte (righe 7-19) è opzionale e definisce l'input; in questo esempio, l'input è un array di stringhe che rappresentano le etichette;

3. l'ultima sezione (righe 20-31) definisce la policy scritta in linguaggio Rego. Viene definita una violazione (riga 25) attivata quando il numero delle etichette richieste

(definite come parametri nell'input, riga 27) è maggiore del numero delle etichette fornite (riga 26). Il messaggio di violazione viene definito come una stringa di testo (riga 31) contenente le etichette mancanti.

Passiamo ora a definire un Constraint che applichi la policy ai Namespace:

```
1  apiVersion: constraints.gatekeeper.sh/v1beta1
2  kind: ClusterRequiredLabels
3  metadata:
4    name: costcenterlabelrequired
5    annotations:
6      description: "I vincoli necessitano etichette o labels"
7    match:
8      kinds:
9        - apiGroups: [""]
10         kinds: ["Namespace"]
11   parameters:
12     labels: ["risorse"]
```

Listato 4.3: Constraint: tutti i Namespace devono avere un'etichetta risorse.

Il vincolo cui sopra è composto da tre parti principali:

1. la sezione `metadata` (righe 3-7) contiene informazioni descrittive del Constraint, come nome (riga 4) e annotazioni (riga 5). Le annotazioni forniscono informazioni aggiuntive come descrizioni o note sul Constraint;

2. la sezione `match` (righe 7-11) specifica i tipi di oggetti Kubernetes a cui si applica il Constraint. In particolare, 'apiGroups' (riga 9) specifica i gruppi API degli oggetti mentre 'kinds' (riga 10) elenca il tipo specifico di oggetto. In questo caso, il vincolo si applica solo ai Namespace;

3. la sezione `parameters` (righe 11-12) definisce i parametri per questo Constraint. I parametri consentono di personalizzare il comportamento del Constraint per ogni istanza specifica. Il parametro 'labels' (riga 12), richiede che siano presenti le etichette specificate nell'elenco che, nel nostro caso, contiene un'unica etichetta chiamata 'risorse'.

### 4.3.5 Vantaggi e svantaggi

Dopo aver illustrato Gatekeeper è chiaro che il suo utilizzo comporta numerosi vantaggi [21], tra cui:

- **maggiore sicurezza**: Gatekeeper permette di verificare che gli oggetti Kubernetes rispettino le policy evitando che vengano eseguite configurazioni potenzialmente dannose o semplicemente non conformi;
- **controllo centralizzato**: è possibile controllare e gestire le policy in modo centralizzato facilitando la gestione dell'infrastruttura Kubernetes;
- **automazione**: Gatekeeper consente di automatizzare l'applicazione delle policy, semplificando e velocizzando il processo di verifica e controllo;
- **flessibilità**: Gatekeeper offre la possibilità di definire policy di personalizzate adattabili alle specifiche esigenze dell'organizzazione.

Tuttavia, l'utilizzo di OPA Gatekeeper comporta anche alcuni svantaggi [41], tra cui:

- **complessità**: l'introduzione di un nuovo componente all'interno dell'infrastruttura Kubernetes comporta una certa complessità nell'implementazione e nella gestione, oltre all'utilizzo di tempo e risorse per comprenderne il corretto funzionamento;
- **overhead**: l'utilizzo di OPA Gatekeeper può comportare un certo overhead prestazionale, generato dalla presenza di policy molto complesse o dai controlli di routine sulle risorse presenti in Kubernetes.

# Capitolo 5

# Progetto

## 5.1 Fasi del progetto

Questo progetto di tesi consiste nell'implementazione di policy per assicurare la conformità e l'adeguatezza di un cluster Kubernetes. Il progetto è stato suddiviso in quattro fasi distinte:

1. **analisi dei requisiti**: in questa fase sono stati identificati i requisiti specifici del progetto, compresi gli obiettivi di sicurezza e le policy da definire;

2. **configurazione del cluster**: è stato creato un cluster Kubernetes locale con `kind`;

3. **definizione delle policy**: sono state definite e implementate le policy utilizzando i Template Constraint e i Constraint di OPA Gatekeeper;

4. **test delle policy**: sono stati eseguiti test per verificare l'efficacia delle policy.

## 5.2 Analisi dei requisiti

L'analisi dei requisiti è un processo di pianificazione rivolto alla definizione delle informazioni necessarie per ottimizzare la configurazione del cluster [43, 44]. Questo processo consente di creare una base solida per la progettazione e lo sviluppo del sistema, garantendo che il risultato finale soddisfi gli obiettivi prefissati.

Al fine di ottenere un cluster in linea con alcune delle principali best practice in Kubernetes, introdotte nella Sezione 3.4.1, sono stati identificati cinque requisiti:

1. **impostare limiti di memoria e di utilizzo CPU**: è necessario definire dei limiti di utilizzo della memoria e della CPU per i Container. Definire questi limiti è fondamentale perché consente al sistema di gestire le risorse in modo efficiente evitando situazioni in cui i Container monopolizzino le risorse;

2. **utilizzare tag specifici per le immagini**: il tag permette l'identificazione di una specifica versione di una Image [8]. È obbligatorio l'uso di tag per garantire il versionamento evitando l'utilizzo di versioni obsolete o non attendibili;

3. **non consentire la creazione di oggetti con privilegi**: un oggetto con privilegi ha accesso ai privilegi di sistema estesi il che potrebbe rappresentare un rischio per la sicurezza del cluster. Non consentire l'utilizzo di privilegi significa che gli oggetti operano con privilegi minimi, riducendo così la superficie di attacco e mitigando potenziali vulnerabilità [18];

4. **utilizzare unicamente repository autorizzati**: un repository di immagini autorizzato è un archivio attendibile e affidabile che ospita le Image dei Container. L'utilizzo di repository autorizzati riduce il rischio di introdurre Image non sicure o compromesse;

5. **introdurre specifiche etichette per i Namespace**: le etichette sono coppie chiave-valore che possono essere applicate a qualsiasi oggetto Kubernetes facilitandone la gestione, configurazione e la risoluzione dei problemi. L'utilizzo di etichette specifiche nei Namespace facilita l'applicazione di politiche di sicurezza e di accesso.

## 5.3 Configurazione cluster

Come introdotto nella Sezione 4.2.2, viene utilizzato `kind` per creare il cluster locale.

La creazione di un cluster locale risulta molto intuitiva grazie alla documentazione fornita [19]. Una volta completata l'installazione di `kind`, è possibile procedere con la creazione del cluster locale. Utilizzando il comando:

```
kind create cluster
```

Viene avviato il processo di creazione del cluster locale identificato con il nome di default 'kind'. È possibile specificare un nome personalizzato per il cluster con il comando:

```
kind create cluster -f "nome del cluster"
```

## 5.4 Definizione delle policy

Una volta che il cluster locale è stato creato con successo possiamo procedere con la definizione delle policy mediante OPA Gatekeeper. Prima di procedere, è importante assicurarsi che OPA Gatekeeper sia correttamente installato nel cluster [15].

Come introdotto nella Sezione 4.3.3, ogni policy sarà formata da Template Constraint e Constraint. I Template non citati sono consultabili nella libreria gatekeeper[1] o nel repository del progetto[2].

### 5.4.1 Limiti di memoria e di utilizzo CPU dei Container

Per poter definire la policy in modo efficiente, è fondamentale capire come venga misurato l'utilizzo della memoria e CPU [36]:

- i limiti e le richieste di utilizzo della memoria sono misurati in byte, è possibile esprimerli attraverso multipli (kilo, mega, giga, etc.) o multipli in base 2 (kibi, mebi, gibi, etc.);
- i limiti e le richieste di utilizzo della CPU vengono espressi in unità CPU. Un'unità CPU corrisponde a una CPU fisica o virtuale a seconda del tipo di nodo. Sono consentite richieste frazionate, ad esempio, la richiesta '0.2' richiede un quinto del tempo CPU, corrispondente a '200 millicpu'.

[1] https://open-policy-agent.github.io/gatekeeper-library/website/
[2] https://github.com/enr3per/Progetto_tesi_Perani_Enrico

Per definire la policy è stato utilizzato il Constraint Template presente nella libreria ufficiale[3], mentre il Constraint è stato definito nel seguente modo:

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sContainerLimits
metadata:
  name: container-must-have-limits
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
  parameters:
    cpu: "200m"
    memory: "1Gi"
```

Listato 5.1: Constraint limiti di memoria e di utilizzo CPU

Nel constraint (listato 5.1), il campo `spec` contiene i dettagli e le specifiche sulle risorse che devono essere valutate. Il campo `match` specifica quali tipi di risorse devono essere considerate dalla policy, in questo caso, la policy si applica ai Pod. La chiave `parameters` contiene i valori massi di utilizzo CPU e memoria, con un limite di 200 millicpu e 1 GiB come utilizzo massimo di memoria. Applicando il Constraint al Pod vengono definiti i limiti delle risorse che vengono condivise dai Container al suo interno.

### 5.4.2 Tag specifici per le Image

Il requisito impone che tutte le immagini del cluster abbiano un tag che soddisfi determinati requisiti. In dettaglio:

- è obbligatorio l'uso del tag;
- non si può utilizzare un tag contenete la stringa 'latest'.

Se una o entrambe le condizioni non sono soddisfatte, la risorsa non è accettata nel cluster.

[3] https://open-policy-agent.github.io/gatekeeper-library/website/validation/containerlimits/

```
1  apiVersion: templates.gatekeeper.sh/v1
2  kind: ConstraintTemplate
3  metadata:
4    name: k8simagetagvalid
5  spec:
6    crd:
7      spec:
8        names:
9          kind: K8sImageTagValid
10   targets:
11     - target: admission.k8s.gatekeeper.sh
12       rego: |
13         package k8simagetagvalid

15         violation[{"msg": msg, "details":{}}] {
16           image := input.review.object.spec.template.spec.containers[_].image
17           not count(split(image, ":")) == 2
18           msg := sprintf("immagine '%v' utilizza non utilizza tag ", [image])
19         }

21         violation[{"msg": msg, "details":{}}] {
22           image := input.review.object.spec.template.spec.containers[_].image
23           tag := split(image, ":")[1]
24           contains(tag, "latest")
25           msg := sprintf("l'immagine utilizza un tag che contiene 'latest'",
     [image])
```

Listato 5.2: Template Image tag valido

Come evidenziato dalla logica del Template (listato 5.2) viene generata una violazione in due casi:

1. la prima violazione (riga 15) ha luogo se l'Image non presenta alcun tag. Per definire una Image correttamente è necessario che sia formata da: 'nomeImmagine:versione'. Il comando `not count(split(":")) == 2` verifica che sono presenti sia in nome dell'Image che la sua versione;

2. la seconda violazione (riga 21) avviene quando il tag utilizzato è o contiene la stringa 'latest'. Il controllo viene effettuato utilizzando il comando `contains`.

Il Constraint (listato 5.3), applica la policy ai Pod nel Namespace `default`.

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sImageTagValid
metadata:
  name: valid-image-tag
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
    namespaces:
      - "default"
```

Listato 5.3: Constraint Image tag valido

### 5.4.3 Tag privilegiati non consentiti

La policy impedisce l'utilizzo di Container con il flag 'privileged'. Viene utilizzato il Template proveniente dalla libreria ufficiale[4].

```
apiVersion: constraints.gatekeeper.sh/v1beta1
  kind: K8sPSPPrivilegedContainer
    metadata:
      name: psp-privileged-container
    spec:
      match:
        excludedNamespaces:
        - kube-system
        kinds:
        - apiGroups:
         - ""
         kinds:
         - Pod
```

Listato 5.4: Constraint flag privileged non ammessa

Il Constraint (listato 5.4) applica la policy a tutti i Pod, ad eccezione dei Pod nel Namespace `kube-system` il quale contiene gli oggetti creati dal sistema Kubernetes [32].

[4] https://open-policy-agent.github.io/gatekeeper-library/website/validation/privileged-containers

### 5.4.4 Utilizzo di repository autorizzati

Il codice Rego nel Template preso dalla libreria ufficiale[5] esegue i controlli sulla conformità dei repository presenti nel Pod: se il repository non corrisponde ad quanto specificato nel Constraint, viene generata una violazione.

```
1  apiVersion: constraints.gatekeeper.sh/v1beta1
2  kind: K8sAllowedRepos
3  metadata:
4    name: repo-is-perrea
5  spec:
6    match:
7      kinds:
8        - apiGroups: [""]
9          kinds: ["Pod"]
10     namespaces:
11       - "default"
12   parameters:
13     repos:
14       - "perrea/"
```

Listato 5.5: Constraint repository ammesse

Il Constraint (listato 5.5) applica la policy ai Pod nel Namespace `default`. La chiave `repos` specifica l'unico repository consentito, nel caso specifico `perrea`.

### 5.4.5 Etichette specifiche nei Namespace

L'ultima policy utilizza il Template della libreria ufficiale[6]. Il Constraint (listato 5.6) applica la policy a tutti i Namespace e attiva una violazione se non è presente l'etichetta `owner` o se l'etichetta fornita non rispetta il pattern `nomeUtente.test`, il valore dell'etichetta deve essere composto da un identificativo `nomeUtente`, costituito da una combinazione di lettere minuscole e/o maiuscole, seguito dalla stringa `.test`.

[5] https://open-policy-agent.github.io/gatekeeper-library/website/validation/allowedrepos

[6] https://open-policy-agent.github.io/gatekeeper-library/website/validation/requiredlabels

```yaml
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sRequiredLabels
metadata:
  name: all-must-have-owner
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Namespace"]
  parameters:
    message: "All namespaces must have an `owner` label that points to your
    company username"
    labels:
      - key: owner
        allowedRegex: "^[a-zA-Z]+ .test$"
```

Listato 5.6: Constraint etichette obbligatorie

## 5.5 Test

Dopo aver definito le policy, è importante eseguire dei test per verificare l'efficacia di OPA Gatekeeper nel prevenire l'accesso di risorse non conformi nel cluster. I test sono stati effettuati in ambiente Windows utilizzando lo strumento d'interfaccia a riga `kubectl` [24] e consistono nella creazione di oggetti Kubernetes, rappresentati da file YAML. Ogni policy viene testata utilizzando almeno due risorse, una conforme alla policy (test positivo) e una non conforme (test negativo).

### 5.5.1 Test limiti di memoria e utilizzo CPU dei Container

Per testare la policy (Sezione 5.4.1) sono state create le seguenti risorse:

- per i test negativi, la prima risorsa[7] richiede un utilizzo di CPU pari a 300 millicpu, superiore al limite di 200 millicpu imposto dalla policy. La seconda risorsa[8] richiede un utilizzo massimo di memoria pari a 2 GiB superiore a quanto concesso.

[7] https://github.com/enr3per/Progetto_tesi_Perani_Enrico/blob/main/policy_test/limited_container/disallowed-cpu.yaml

[8] https://github.com/enr3per/Progetto_tesi_Perani_Enrico/blob/main/policy_test/limited_container/disallowed-memory.yaml

In entrambi i casi OPA Gatekeeper impedisce l'accesso delle risorse nel cluster e restituisce un messaggio d'errore come illustrato nei listati 5.7 e 5.8;

```
kubectl apply -f disallowed-cpu.yaml
Error from server (Forbidden): error when creating "disallowed-cpu.yaml":
    admission webhook "validation.gatekeeper.sh" denied the request:
    [container-must-have-limits] container <mongo> cpu limit <300m> is
    higher than the maximum allowed of <200m>
```

Listato 5.7: Output test risorsa con utilizzo CPU superiore ai limiti

```
kubectl apply -f disallowed-memory.yaml
Error from server (Forbidden): error when creating
    "disallowed-memory.yaml": admission webhook "validation.gatekeeper.sh"
    denied the request: [container-must-have-limits] container <mongo>
    memory limit <2Gi> is higher than the maximum allowed of <1Gi>
```

Listato 5.8: Output test risorsa con utilizzo memoria superiore ai limiti

- la terza risorsa[9] rispetta i vincoli di utilizzo di memoria e CPU imposti, viene quindi creata correttamente (listato 5.9).

```
kubectl apply -f is-allowed.yaml
pod/container-limit-allowed created
```

Listato 5.9: Output test risorsa conforme

Eseguendo il comando:

```
kubectl get all
```

Il Container creato è visibile nel cluster.

```
 NAME                            READY          STATUS
pod/container-limit-allowed   0/1              ContainerCreating
```

[9] https://github.com/enr3per/Progetto_tesi_Perani_Enrico/blob/main/policy_test/limited_container/is-allowed.yaml

### 5.5.2 Test tag specifici per le Image

Al fine di testare la policy (Sezione 5.4.2) sono state create le seguente risorse:

- la prima risorsa[10] non utilizza alcun tag e quindi non è conforme (listato 5.10).

  La seconda risorsa[11] ha come tag `v1.0-latest.0` per cui, contenendo la parola latest, non viene ammessa nel cluster (listato 5.11);

```
kubectl apply -f not-allawod-tag-leatest.yaml
Error from server (Forbidden): error when creating
    "not-allawod-tag-leatest.yaml": admission webhook
    "validation.gatekeeper.sh" denied the request: [valid-image-tag]
    immagine non utilizza un tag%!(EXTRA string=perrea/mongo)
```

Listato 5.10: Output test risorsa senza tag

```
kubectl apply -f not-alloved-tag-2.yaml
Error from server (Forbidden): error when applying patch:
for: "not-alloved-tag-2.yaml": error when patching
    "not-alloved-tag-2.yaml": admission webhook "validation.gatekeeper.sh"
    denied the request: [valid-image-tag] l'immagine utilizza un tag che
    contiene 'latest'%!(EXTRA string=perrea/mongo:v1.0-latest)
```

Listato 5.11: Output test risorsa con tag contenente stringa latest

- la terza risorsa[12] utilizza il tag `5.0`, la risorsa è conforme alla policy (listato 5.12).

```
kubectl apply -f is-allowed.yaml
deployment.apps/mongo created
```

Listato 5.12: Output test risorsa conforme

[10] https://github.com/enr3per/Progetto_tesi_Perani_Enrico/blob/main/policy_test/image_tag/not-allawod-tag-leatest.yaml

[11] https://github.com/enr3per/Progetto_tesi_Perani_Enrico/blob/main/policy_test/image_tag/not-alloved-tag-2.yaml

[12] https://github.com/enr3per/Progetto_tesi_Perani_Enrico/blob/main/policy_test/image_tag/is-allowed.yaml

È possibile verificare l'avvenuta creazione del Deployment eseguendo il comando:

```
 kubectl get deployments
NAME          READY           UP-TO-DATE
mongo         0/1              1
```

### 5.5.3 Test tag privilegiati non consentiti

La policy (Sezione 5.4.3) è stata testata mediante l'utilizzo di due risorse:

- il test negativo viene effettuato mediante una risorsa[13] con il flag `privileged: true`. Gatekeeper impedisce l'accesso della risorsa nel cluster e restituisce un messaggio d'errore (listato 5.13).

```
kubectl apply -f not-allowed-1.yaml
Error from server (Forbidden): error when creating "not-allowed-1.yaml":
    admission webhook "validation.gatekeeper.sh" denied the request:
    [psp-privileged-container-sample] Privileged container is not allowed:
    nginx, securityContext: {"privileged": true}
```

Listato 5.13: Output test risorsa con privilegi

- il test positivo consiste nella creazione di una risorsa[14] con il flag `privileged: false`;

È possibile verificare l'avvenuta creazione dell'oggetto con il comando:

```
 kubectl get all
NAME                            READY     STATUS
pod/nginx-privileged-allowed    0/1       ContainerCreating
```

### 5.5.4 Test utilizzo di repository autorizzati

Per testare la policy (Sezione 5.4.4) sono state create le seguenti risorse:

[13] https://github.com/enr3per/Progetto_tesi_Perani_Enrico/blob/main/policy_test/privileges_deprecated/not-allowed-1.yaml

[14] https://github.com/enr3per/Progetto_tesi_Perani_Enrico/blob/main/policy_test/privileges_deprecated/is-allowed.yaml

- per i test negativi sono state utilizzate due risorse. La prima risorsa [15] non specifica nessun repository e viene restituito un messaggio d'errore (listato 5.14). La seconda risorsa[16] utilizza il repository `personal` che non è consentito (listato 5.15).

```
kubectl apply -f not-allow-miss-repository.yaml
Error from server (Forbidden): error when creating
    "not-allow-miss-repository.yaml": admission webhook
    "validation.gatekeeper.sh" denied the request: [repo-is-perrea] il container
    <mongo> ha un'errata repository <momngodb>, le reporisories ammesse sono:
    ["perrea/"]
```

Listato 5.14: Output test risorsa senza repository

```
kubectl apply -f not-allowed-wrong-repository.yaml
Error from server (Forbidden): error when creating
    "not-allowed-wrong-repository.yaml": admission webhook
    "validation.gatekeeper.sh" denied the request: [repo-is-perrea] il
    container <mongo> ha un'errata repository <vineguard/mongo:5.0>, le
    reporisories ammesse sono: ["perrea/"]
```

Listato 5.15: Output test risorsa con repository non ammessa

- per il test positivo, viene utilizzata una risorsa[17] che specifica il repository `/perrea`, la risorsa viene creata correttamente.

È possibile verificare l'avvenuta creazione della risorsa eseguendo il comando:

```
kubectl get pods
NAME                              READY          STATUS
container-trusted-repository      0/1            ContainerCreating
```

### 5.5.5 Test etichette richieste

per verificare che i Namespace abbiano un'etichetta `owner` conforme alla policy (Sezione 5.4.5) sono state create le seguenti risorse:

[15] https://github.com/enr3per/Progetto_tesi_Perani_Enrico/blob/main/policy_test/repositories_allowed/not-allow-miss-repository.yaml

[16] https://github.com/enr3per/Progetto_tesi_Perani_Enrico/blob/main/policy_test/repositories_allowed/not-allowed-wrong-repository.yaml

[17] https://github.com/enr3per/Progetto_tesi_Perani_Enrico/blob/main/policy_test/repositories_allowed/is-allowed.yaml

- per i test negativi sono state create due risorse. La prima risorsa[18] non specifica nessuna etichetta `owner`. La seconda risorsa[19] ha l'etichetta `owner: production` non conforme al pattern consentito. In entrambi i casi Gatekeeper nega l'accesso al cluster delle risorse (listati 5.16 e 5.17);

```
kubectl apply -f label-not-allowed.yaml
Error from server (Forbidden): error when creating "label-not-allowed.yaml":
    admission webhook "validation.gatekeeper.sh" denied the request:
    [all-must-have-owner] All namespaces must have an `owner` label that points
    to your company username
```

Listato 5.16: Output test risorsa senza etichetta

```
kubectl apply -f label-not-allowed-bis.yaml
Error from server (Forbidden): error when creating "label-not-allowed-bis.yaml":
    admission webhook "validation.gatekeeper.sh" denied the request:
    [all-must-have-owner] All namespaces must have an `owner` label that points
    to your company username
```

Listato 5.17: Output test risorsa con etichetta errata

- per il test positivo[20] è stata creato l'oggetto Namespace `my-namespace` con l'etichetta `owner: rossi.test`, la risorsa viene creata correttamente;

[18] https://github.com/enr3per/Progetto_tesi_Perani_Enrico/blob/main/policy_test/labels_required/label-not-allowed.yaml

[19] https://github.com/enr3per/Progetto_tesi_Perani_Enrico/blob/main/policy_test/labels_required/label-not-allowed-bis.yaml

[20] https://github.com/enr3per/Progetto_tesi_Perani_Enrico/blob/main/policy_test/labels_required/label-is-allowed.yaml

Con il comando:

```
kubectl get namespaces
```

```
NAME                 STATUS   AGE
default              Active   2m51s
gatekeeper-system    Active   2m9s
kube-node-lease      Active   2m53s
kube-public          Active   2m53s
kube-system          Active   2m53s
local-path-storage   Active   2m43s
my-namespace         Active   31s
```

Il Namespace `my-namespace` è visibile nel cluster locale.

# Capitolo 6

# Conclusioni

## 6.1 Riassunto della tesi

In questa tesi è stato presentato un approccio per l'utilizzo di Open Policy Agent (OPA) per la definizione di policy *as code* all'interno di un cluster Kubernetes. Si è illustrato il funzionamento di OPA e del suo integratore con Kubernetes: Gatekeeper. Successivamente, è stata illustrata l'implementazione delle policy mediante l'utilizzo di Template e Constraint, e sono stati descritti i test per verificare la conformità delle risorse alle policy definite.

## 6.2 Limiti e sviluppi futuri

Uno dei principali limiti del progetto è stato l'utilizzo di un cluster locale, il quale non ha permesso di valutare l'efficacia delle policy definite in un ambiente di produzione reale.

Per superare questo limite e valutare l'applicabilità delle policy in un ambiente di produzione, sarebbe auspicabile implementare una fase di sperimentazione su un cluster Kubernetes di produzione. Ciò consentirebbe di valutare l'impatto delle politiche sulle risorse effettivamente utilizzate dagli utenti e di raccogliere feedback sulle eventuali criticità o miglioramenti necessari.

Inoltre, sarebbe interessante esplorare l'integrazione di OPA e Gatekeeper con strumenti di monitoraggio e reportistica come OPA Gatekeeper dashboard e Prometheus metrics per ottenere una visione più approfondita sulla conformità delle risorse alle policy.

Questo potrebbe includere la generazione di report automatici sullo stato delle policy e l'identificazione di eventuali violazioni o potenziali problemi di sicurezza.

Infine, considerando la natura dinamica degli ambienti di produzione, sarebbe opportuno implementare un meccanismo per il versionamento delle politiche, ciò consentirebbe di gestire le modifiche delle policy nel tempo, garantendo la tracciabilità e la gestione controllata delle revisioni.

## 6.3 Conclusioni finali

L'utilizzo di OPA Gatekeeper per definire policy all'interno di un cluster Kubernetes si è rivelato un approccio efficace per garantire la sicurezza e la conformità delle risorse. Nonostante i limiti riscontrati nell'ambiente locale, l'applicazione di policy ha dimostrato di essere uno strumento prezioso per fare in modo che le risorse siano coerenti con le best practice e le politiche aziendali.

La continua evoluzione degli ambienti di produzione richiede un approccio agile e flessibile per garantire la sicurezza delle applicazioni e delle infrastrutture e l'utilizzo di OPA come parte di un processo più ampio di gestione delle politiche di sicurezza può contribuire a creare un ambiente adatto a ogni esigenza.

È importante sottolineare che l'implementazione di politiche all'interno di un cluster Kubernetes rappresenta solo uno degli aspetti della sicurezza complessiva: è necessario adottare un approccio globale che consideri il sistema nella sua interezza e che tenga conto delle varie interconnessioni fra le parti che lo compongono e quindi che includa anche la protezione dei dati, la gestione degli accessi, il monitoraggio delle minacce e altre pratiche di sicurezza. Affrontare la sicurezza in modo completo significa considerare ogni aspetto e interconnessione del sistema, in modo da affrontare le sfide emergenti e proteggere adeguatamente le risorse aziendali.

# Bibliografia


[1] Open Policy Agent. *Introduction to OPA*. Documentazione. URL: https://www.openpolicyagent.org/docs/latest/.

[2] Mohamed Ahmed. *Introducing Policy As Code: The Open Policy Agent*. URL: https://www.cncf.io/blog/2020/08/13/introducing-policy-as-code-the-open-policy-agent-opa/.

[3] Kohgadai Ajmal. *12 Kubernetes Configuration Best Practices*. URL: https://cloud.redhat.com/blog/12-kubernetes-configuration-best-practices.

[4] B. Burns et al. *Kubernetes Best Practices: Blueprints for Building Successful Applications on Kubernetes*. O'Reilly Media, 2019. ISBN: 9781492056478. URL: https://learning.oreilly.com/library/view/kubernetes-best-practices/9781098142155/.

[5] Cambridge dictionary. *policy*. URL: https://dictionary.cambridge.org/dictionary/english/policy.

[6] Docker. *Containerize an application*. URL: https://docs.docker.com/get-started/02_our_app/.

[7] Docker. *Docker documentation*. Documentazione. URL: https://docs.docker.com/get-started/overview/.

[8] Docker. *Docker tag*. URL: https://docs.docker.com/engine/reference/commandline/tag/.

[9] Docker. *Dockerfile reference*. URL: https://docs.docker.com/engine/reference/builder/.

[10] Docker. *Networks*. URL: https://docs.docker.com/network/.

[11] Docker. *Volumes*. URL: https://docs.docker.com/storage/volumes/.

[12] Cloud Native Computing Foundation. *Gatekeeper*. Documentazione. URL: https://open-policy-agent.github.io/gatekeeper/website/docs/.

[13] Gatekeeper. *Constraint Templates Policy Comparison*. Documentazione. URL: https://open-policy-agent.github.io/gatekeeper/website/docs/constrainttemplates/#v1-constraint-template.

[14] Gatekeeper. *Constraint Templates: How to use Gatekeeper*. Documentazione. URL: https://open-policy-agent.github.io/gatekeeper/website/docs/howto/#constraint-templates.

[15] Gatekeeper. *Installation*. Documentazione. URL: https://open-policy-agent.github.io/gatekeeper/website/docs/install/.

[16] OPA Gatekeeper. *OPA Constraint Framework*. Documentazione. URL: https://github.com/open-policy-agent/frameworks/tree/master/constraint.

[17] OPA Gatekeeper. *OPA Gatekeeper Library*. Documentazione. URL: https://open-policy-agent.github.io/gatekeeper-library/website/.

[18] Or Kamara. *Kubernetes – privileged pods*. blog. URL: https://www.cncf.io/blog/2020/10/16/hack-my-mis-configured-kubernetes-privileged-pods/.

[19] kind. *Installation*. Quick Start installation. URL: https://kind.sigs.k8s.io/docs/user/quick-start/.

[20] kind. *kind documentation*. Documentazione. URL: https://kind.sigs.k8s.io/.

[21] Kubernetes. *A Guide to Kubernetes Admission Controllers*. Documentazione. URL: https://kubernetes.io/blog/2019/03/21/a-guide-to-kubernetes-admission-controllers/.

[22] Kubernetes. *Admission Controllers Reference*. Documentazione. URL: https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/.

[23] Kubernetes. *best practice overview*. Documentazione. URL: https://kubernetes.io/docs/concepts/configuration/overview/.

[24] Kubernetes. *Command line tool (kubectl)*. URL: https://kubernetes.io/docs/reference/kubectl/.

[25] Kubernetes. *ConfigMaps*. URL: https://kubernetes.io/docs/concepts/configuration/configmap/.

[26] Kubernetes. *Creating Highly Available Clusters with kubeadm*. URL: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/.

[27] Kubernetes. *definition*. Documentazione. URL: https://kubernetes.io/it/docs/concepts/overview/what-is-kubernetes/.

[28] Kubernetes. *Deployments*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

[29] Kubernetes. *features*. Documentazione. URL: https://kubernetes.io/.

[30] Kubernetes. *Kubernetes Components*. URL: https://kubernetes.io/docs/concepts/overview/components/.

[31] Kubernetes. *Limit Ranges*. URL: https://kubernetes.io/docs/concepts/policy/limit-range/.

[32] Kubernetes. *Namespace*. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.

[33] Kubernetes. *Objects In Kubernetes*. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/.

[34] Kubernetes. *OPA Gatekeeper: Policy and Governance for Kubernetes*. Documentazione. URL: https://kubernetes.io/blog/2019/08/06/opa-gatekeeper-policy-and-governance-for-kubernetes/.

[35] Kubernetes. *Pods*. URL: https://kubernetes.io/docs/concepts/workloads/pods/.

[36] Kubernetes. *Resource Management for Pods and Containers*. Documentazione. URL: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/.

[37] Kubernetes. *Secrets*. URL: https://kubernetes.io/docs/concepts/configuration/secret/.

[38] Kubernetes. *Service*. URL: https://kubernetes.io/docs/concepts/services-networking/service/.

[39] Kubernetes. *Volumes*. URL: https://kubernetes.io/docs/concepts/storage/volumes/.

[40] minikube. *minikube documentation*. Documentazione. URL: https://minikube.sigs.k8s.io/docs/start/.

[41] NeonMirros. *Kubernetes Policy Comparison*. article. URL: https://neonmirrors.net/post/2021-02/kubernetes-policy-comparison-opa-gatekeeper-vs-kyverno/.

[42] OPA. *Overview and Architecture*. Documentazione. URL: https://www.openpolicyagent.org/docs/latest/kubernetes-introduction/.

[43] Md Shazibul Islam Shamim, Farzana Ahamed Bhuiyan e Akond Rahman. «Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices». In: *2020 IEEE Secure Development (SecDev)* (2020). URL: https://arxiv.org/pdf/2006.15275.pdf.

[44] Shazibul Islam Shamim. «Mitigating security attacks in kubernetes manifests for security best practices violation». In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2021). URL: https://dl.acm.org/doi/pdf/10.1145/3468264.3473495.

[45] Muñoz Stefani. *The history of Docker's climb in the container management market*. 2019. URL: https://www.techtarget.com/searchitoperations/feature/The-history-of-Dockers-climb-in-the-container-management-market.

[46] Styra. *Open Policy Agent 101: A Beginner's Guide*. URL: https://www.styra.com/blog/open-policy-agent-101-a-beginners-guide/.

[47] Styra. *Policy Language*. Documentazione. URL: https://www.openpolicyagent.org/docs/latest/policy-language/.

[48] Taylor Twain. *Setting Up A Local Kubernetes Cluster*. 2022. URL: https://techgenix.com/local-kubernetes-cluster-setup/.

[49] Tuomas Vase. «Advantages of Docker». In: 2015, pp. 15–16. URL: https://jyx.jyu.fi/handle/123456789/48029.

[50] Abhishek Verma et al. «Large-scale cluster management at Google with Borg». In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015. URL: https://dl.acm.org/doi/abs/10.1145/2741948.2741964.

[51] YAML. *YAML Aint Markup Language*. URL: https://yaml.org/spec/1.2.2/.