Politecnico di Torino

Politecnico di Torino

Computer Engineering

A.a. 2024/2025

Sessione di laurea Luglio 2025

Applicazioni Web in ambienti Cloud e Bare Metal

Ottimizzazioni, Sicurezza e Personalizzazione

Relatori:

Giovanni Malnati

Daniele Apiletti

Candidati:

Giuseppe Poma

Sommario

Negli ultimi anni, il paradigma dello sviluppo delle Applicazioni Web ha subito un'evoluzione radicale, passando da architetture monolitiche a modelli distribuiti basati su microservizi, container ed ambienti cloud-native. Questo cambiamento ha introdotto significativi vantaggi in termini di modularità, scalabilità, resilienza e rapidità di rilascio, ma ha anche reso più complesse le fasi di progettazione, distribuzione e gestione delle applicazioni. L'adozione su larga scala di tecnologie cloud comporta nuove sfide, tra cui l'orchestrazione di microservizi, l'ottimizzazione delle risorse per contenere i costi operativi, la gestione della sicurezza in ambienti distribuiti, la necessità di personalizzare le piattaforme cloud per soddisfare requisiti applicativi specifici e l'elaborazione efficiente di grandi quantità di dati eterogenei.

La presente tesi si propone di affrontare sistematicamente tali problematiche, offrendo un insieme di soluzioni tecnologiche ed architetturali che si pongono in continuità con lo stato dell'arte. Nella prima parte viene analizzato il ruolo fondamentale dell'orchestratore Kubernetes nella gestione automatizzata del ciclo di vita dei container e dei microservizi. Successivamente, si esplorano le tecniche di ottimizzazione delle performance applicative mediante l'adozione del framework Quarkus, concepito per ambienti cloud-native, e del sistema operativo minimalista Talos Linux, che riduce drasticamente l'overhead dei nodi di calcolo. Si prosegue con l'analisi delle problematiche di sicurezza, introducendo strumenti avanzati come Falco per la rilevazione di comportamenti anomali nel cluster e l'adozione di politiche di controllo degli accessi per limitare i privilegi operativi. Un ulteriore contributo è rappresentato dalla progettazione ed implementazione di un controller personalizzato in linguaggio Go, capace di convertire automaticamente risorse Kubernetes di tipo Ingress in regole di instradamento per Spring Cloud Gateway, dimostrando la possibilità di estendere dinamicamente le funzionalità del cluster secondo esigenze specifiche. Infine, si affronta il tema della gestione dei dati distribuiti, presentando Trino come motore SQL federato in grado di interrogare simultaneamente più fonti di dato eterogenee, migliorando l'accessibilità e l'efficienza delle analisi.

Nel loro insieme, i contributi sviluppati in questa tesi, evidenziano l'importanza di un approccio consapevole allo sviluppo cloud-native, che non si limiti all'utilizzo passivo delle tecnologie offerte, ma che ne comprenda le dinamiche interne per trarne vantaggio. L'obiettivo è quello di fornire una base metodologica ed operativa per la progettazione, realizzazione e messa in campo di Applicazioni Web moderne, efficienti, sicure e scalabili.

Ringraziamenti

Un grazie dal profondo del cuore al mio relatore, Giovanni Malnati, mi ha preso sotto la sua ala e mi ha insegnato a pensare in grande, a guardare avanti, oltre l'ostacolo, verso l'obbiettivo. Sono stati mesi intensi di formazione, ma la più importante lezione che mi porto dietro è: mai smettere di essere curiosi.

Ai miei eccezionali colleghi e formidabili amici del team TonicMinds. In particolare, un sincero grazie a Nicolò e Gabriele. Sono stati i miei mentori, ingegneri sopra le righe ed amici veri.

Alla mia famiglia, da sempre il mio supporto e sostegno. Senza di loro non sarei nessuno. Sono grato di avere dei genitori MERAVIGLIOSI, che con il loro amore smisurato e la loro solarità mi hanno sempre dato tutto quello che un ragazzo possa desiderare, permettendomi di vivere la vita sempre con il sorriso e la felicità nel cuore.

Alla mia ragazza, che ancora mi sopporta. Un rapporto speciale fatto di amore sincero. La persona con cui condivido le mie giornate, momenti, gioie e lamentele. Una gelato ed una passeggiata al porto saranno sempre il nostro modo per dirci che nonostante tutto non riusciamo a stare lontani.

Ai miei nonni, a chi c'è e a chi se n'è andato. GRAZIE.

Indice

Elenco delle tabelle VII

Elenco delle figure VIII

Glossario X

Acronimi XI

1 Introduzione 1
1.1 Obiettivi della Tesi . . . 1
1.2 Struttura della Tesi . . . 2

2 Il Cloud e Kubernetes 4
2.1 Evoluzione delle Applicazioni Web . . . 4
2.2 Rilascio delle Web Applications: La storia . . . 5
2.3 Docker . . . 8
2.3.1 Che cos'è Docker . . . 8
2.3.2 Docker: le limitazioni . . . 9
2.3.3 Docker Compose . . . 10
2.4 Kubernetes . . . 11
2.4.1 Che cos'è Kubernetes . . . 11
2.4.2 Gli aspetti fondamentali . . . 12
2.4.3 L'architettura . . . 13
2.4.4 I principali componenti . . . 14

3 Ottimizzazioni 21
3.1 Ottimizzazioni nel contesto delle Applicazioni Web in Cloud . . . 21
3.2 Ottimizzazione dei Microservizi . . . 22
3.2.1 SpringBoot . . . 22
3.2.2 Quarkus . . . 23
3.2.3 Comparativa SpringBoot - Quarkus . . . 25

3.3 Ottimizzazione del Sistema Operativo . . . 32
3.3.1 Talos Linux . . . 32

4 Sicurezza 34
4.1 Minacce esterne e monitoraggio del cluster con Falco . . . 35
4.1.1 Monitoraggio avanzato degli accessi SSH . . . 36
4.2 Minacce interne e controllo degli accessi con RBAC e Keycloak . . . 37
4.2.1 Gestione classica di RBAC . . . 37
4.2.2 Soluzione avanzata: RBAC integrato con Keycloak . . . 38
4.2.3 Differenze tra approccio diretto e federato . . . 38

5 Personalizzazione dell'ambiente Cloud 40
5.1 La personalizzazione nel contesto Cloud . . . 40
5.2 Ingress Controller ed API Gateway . . . 40
5.3 Ingress Controller Personalizzato per Spring Cloud Gateway . . . 41
5.3.1 Architettura e funzionamento . . . 41
5.3.2 Tecnologie e librerie utilizzate . . . 41
5.3.3 Client-go, Controller-runtime e meccanismi interni . . . 42
5.3.4 Flusso interno di funzionamento: interazione tra client-go, Informer e Controller . . . 43
5.3.5 Reconcile e Manager . . . 45
5.3.6 Client, Informer e Cache . . . 46
5.3.7 Meccanismo di riconciliazione . . . 47
5.3.8 Strategia di aggiornamento e idempotenza . . . 50
5.3.9 Risultati . . . 50

6 Motore SQL distibuito per le Applicazioni Web moderne 52
6.1 La necessità di un motore SQL . . . 52
6.2 Cos'è Trino . . . 53
6.3 SQL-on-Anything . . . 53
6.4 Query Federate . . . 54
6.5 Concetti fondamentali . . . 54
6.6 L'Architettura . . . 55
6.7 I connettori . . . 56
6.8 Esecuzione delle Query . . . 57
6.9 Predicate Pushdown e ottimizzazione basata su costi . . . 58
6.10 Limitazioni dei connettori e uso delle viste . . . 59
6.11 Implementazione in Kubernetes . . . 59
6.11.1 Analisi sperimentale delle performance . . . 60
6.11.2 Consumo di risorse . . . 64
6.12 Considerazioni . . . 65

7 Conclusioni 66

Bibliografia 67

Elenco delle tabelle

2.1 Comparativa tra l'architettura Monolitica e quella a Microservizi . . 6

3.1 Comparativa Performance: SpringBoot - Quarkus 26
3.2 Fattore di miglioramento delle versioni non *reactive* rispetto allo standard SpringBoot JVM 27
3.3 Comparativa Performance Reactive: SpringBoot - Quarkus 27
3.4 Comparativa Performance con un carico di 1000 richieste: SpringBoot - Quarkus 28
3.5 Comparativa Performance Reactive con un carico di 1000 richieste: SpringBoot - Quarkus 29
3.6 Confronto ISO: Talos Linux, Ubuntu Server ed Ubuntu Desktop . . 33
3.7 Confronto tra Talos Linux e Ubuntu Server nell'installazione di un cluster Kubernetes 33

Elenco delle figure

2.1 Architettura Monolitica comparata con l'architettura a Microservizi 5
2.2 Confronto tra VMs e Containers 7
2.3 Condivisione dei layers tra container Docker 10
2.4 I componenti di Kubernetes 14
2.5 ReplicaSets 15
2.6 ReplicaSet controllato dal Deployment 16
2.7 Uso di due diverse configMap per la configurazione di due ambienti diversi 20

3.1 Quarkus: le performance riportate sul sito ufficiale 25
3.2 Evoluzione dei tempi di esecuzione con un carico di 1000 richieste . 28
3.3 Distribuzione dei tempi di esecuzione con un carico di 1000 richieste 29
3.4 Evoluzione dei tempi di esecuzione con un carico di 1000 richieste (Reactive) 30
3.5 Distribuzione dei tempi di esecuzione con un carico di 1000 richieste (Reactive) 31

5.1 Architettura e funzionamento di un controller 44

6.1 L'architettura di Trino 55
6.2 Trino: Esecuzione dell'interrogazione 57
6.3 Comparativa: Trino - Nativo 62
6.4 Trino: 100 richieste di query federate 64

Glossario

Keycloak Keycloak è un prodotto software open source che consente il single sign on con gestione dell'identità e degli accessi, pensato per applicazioni e servizi moderni. [1]

MongoDB MongoDB (da "humongous", enorme) è un DBMS non relazionale, orientato ai documenti. Classificato come un database di tipo NoSQL, MongoDB si allontana dalla struttura tradizionale basata su tabelle dei database relazionali in favore di documenti in stile JSON con schema dinamico (MongoDB chiama il formato BSON), rendendo l'integrazione di dati di alcuni tipi di applicazioni più facile e veloce. [2]

OAuth 2.0 OAuth 2.0 è il protocollo standard di settore per l'autorizzazione. OAuth 2.0 si concentra sulla semplicità di utilizzo per gli sviluppatori client, fornendo al contempo flussi di autorizzazione specifici per applicazioni web, applicazioni desktop, telefoni cellulari e dispositivi domestici. Questa specifica e le sue estensioni sono in fase di sviluppo all'interno dell'IETF OAuth Working Group. [3]

PostgreSQL PostgreSQL è un potente sistema di database relazionale a oggetti open source che utilizza ed estende il linguaggio SQL, combinandolo con numerose funzionalità che consentono di archiviare e scalare in modo sicuro anche i carichi di lavoro più complessi. Le origini di PostgreSQL risalgono al 1986, nell'ambito del progetto POSTGRES presso l'Università della California a Berkeley, e vanta oltre 35 anni di sviluppo attivo sulla piattaforma principale. [4]

Acronimi

AOT

Ahead of Time

API

Application Programming Interface

CD

Continuous Delivery

CBO

Cost Based Optimization

CI

Continuous Integration

CLI

Command Line Interface

DB

Database

DDoS

Distributed Denial of Service

DNS

Domain Name System

FIFO

First In First Out

GB

Giga Byte, 1 GB = 0.93 GiB

GiB

Gibi Byte

IAM

Identity and Access Management

JDBC

Java Database Connectivity

JIT

Just in Time

JPA

Java Persistence API

JVM

Java Virtual Machine

mTLS

mutual TLS

NFS

Network File System

ODBC

Open Database Connectivity

OS

Sistema Operativo

RBAC

Role Based Access Control

REST

Representational State Transfer

SCG

Spring Cloud Gateway

SPA

Single Page Application

SQL

Structured Query Language

SSH

Secure Shell

TLS

Transport Layer Security

VM

Macchina Virtuale

Capitolo 1

Introduzione

Negli ultimi dieci anni, il panorama dello sviluppo di applicazioni web ha attraversato una trasformazione profonda e strutturale. Se fino agli inizi del 2010 era consuetudine adottare architetture monolitiche basate sul paradigma client-server, l'evoluzione tecnologica ha spinto il settore verso modelli architetturali più flessibili, resilienti e scalabili. In particolare, l'adozione massiva di microservizi, Single Page Applications (SPA), funzioni serverless e container, ha ridefinito le modalità con cui le applicazioni moderne vengono progettate, sviluppate e distribuite in ambienti cloud-native.

Questo passaggio ha introdotto significativi benefici, tra cui una maggiore modularità del codice, una migliorata disponibilità dei servizi, una risposta più rapida alle richieste dell'utente (*responsiveness*) e una semplificazione del ciclo di rilascio delle funzionalità. Tuttavia, tali vantaggi sono accompagnati da una crescente complessità architetturale, che comporta nuove sfide ingegneristiche e gestionali. Tra queste si annoverano la necessità di orchestrare efficacemente una moltitudine di servizi, l'adozione di strategie di ottimizzazione per ridurre i costi operativi, l'implementazione di misure di sicurezza robuste, la personalizzazione dell'ambiente cloud per esigenze applicative specifiche, nonché la gestione efficiente di grandi volumi di dati eterogenei distribuiti tra i diversi servizi.

La presente tesi si propone di analizzare in modo sistematico tali problematiche, evidenziandone le criticità tecniche e proponendo soluzioni concrete per supportare un approccio progettuale moderno, efficiente e sicuro nello sviluppo di Applicazioni Web in ambienti cloud.

1.1 Obiettivi della Tesi

L'obiettivo principale di questa tesi è l'analisi critica e l'approfondimento delle principali sfide legate allo sviluppo di applicazioni web distribuite in ambienti

cloud, proponendo soluzioni tecnologiche e metodologiche allineate con lo stato dell'arte. A tal fine, i contenuti della tesi sono organizzati in modo da affrontare le problematiche più rilevanti attraverso l'adozione di strumenti, framework e tecniche consolidati nel settore.

Nel Capitolo 2 viene introdotto l'orchestratore Kubernetes, evidenziandone il ruolo centrale come strato infrastrutturale per la gestione e l'automazione del ciclo di vita dei microservizi. Il Capitolo 3 è dedicato all'ottimizzazione delle performance e dei consumi delle applicazioni cloud-native, attraverso l'uso combinato del framework Quarkus e del sistema operativo minimalista Talos Linux, entrambi scelti per le loro caratteristiche di efficienza ed integrazione con i containers. Il Capitolo 4 esplora le problematiche di sicurezza in ambito cloud, proponendo l'adozione di soluzioni avanzate come Falco per il monitoraggio di eventi nel cluster e l'impiego di politiche di accesso RBAC limitate per la gestione dei permessi. Il Capitolo 5 si concentra invece sull'estensione delle funzionalità offerte da Kubernetes mediante la progettazione di un controller personalizzato in grado di convertire dinamicamente risorse Ingress in regole di routing per Spring Cloud Gateway, illustrando un caso d'uso concreto di personalizzazione dell'ambiente. Infine, nel Capitolo 6, viene affrontato il problema della frammentazione e dell'interrogazione di dati distribuiti, presentando Trino come motore SQL capace di eseguire query federate su fonti eterogenee.

In sintesi, la tesi intende dimostrare come l'approccio cloud-native richieda non soltanto la conoscenza degli strumenti offerti dalle piattaforme cloud, ma anche una piena comprensione delle dinamiche che li governano. Solo così è possibile implementare soluzioni realmente efficienti, sicure, personalizzabili e scalabili, in linea con i requisiti delle moderne Applicazioni Web.

1.2 Struttura della Tesi

La trattazione è suddivisa nei seguenti macrotemi:

- **Sviluppo in Cloud e Kubernetes**: introduzione all'architettura a microservizi e al ruolo dell'orchestrazione.
- **Ottimizzazione**: tecniche per migliorare prestazioni, tempi di avvio e consumo di risorse, riducendo i costi operativi.
- **Sicurezza**: strumenti e strategie per proteggere il cluster da minacce esterne ed interne.
- **Personalizzazione**: sviluppo di componenti custom per l'adattamento delle funzionalità standard del cloud alle esigenze applicative.

- **Gestione dei Dati**: soluzioni per interrogare ed elaborare dati distribuiti su fonti eterogenee in modo efficiente.

Capitolo 2

Il Cloud e Kubernetes

2.1 Evoluzione delle Applicazioni Web

Nel panorama dello sviluppo software, ed in particolare nel contesto delle applicazioni web, si è assistito negli ultimi anni ad una evoluzione significativa dei modelli architetturali adottati. In passato, le *Web Applications* venivano comunemente sviluppate secondo un paradigma monolitico, nel quale l'intera applicazione veniva progettata come un blocco unico ed indivisibile, all'interno del quale tutti i componenti, dalla logica di business al livello di presentazione, fino alla persistenza dei dati, risultavano fortemente correlati.

Tale approccio, se da un lato semplifica la fase iniziale di progettazione e rilascio, dall'altro comporta numerose limitazioni che emergono con particolare evidenza all'aumentare della complessità e delle dimensioni dell'applicativo. In particolare:

- qualsiasi modifica, anche marginale, richiede una nuova compilazione e il rilascio dell'intera applicazione, riducendo la frequenza e l'agilità dei rilasci;
- la manutenibilità decresce in maniera significativa con la crescita del progetto, a causa della stretta interdipendenza tra i moduli software.

Nel contesto attuale, caratterizzato da un'elevata dinamicità dei mercati, rapidi cambiamenti tecnologici e mutevoli esigenze da parte degli utenti finali, tali limitazioni si rivelano critiche. La capacità di adattarsi rapidamente al cambiamento rappresenta oggi una delle principali leve competitive per le imprese tecnologiche. Negli ambienti produttivi moderni, l'adattabilità non è più un valore aggiunto, bensì un requisito imprescindibile per mantenere la rilevanza e la sostenibilità dell'offerta digitale nel tempo. [5]

A fronte di questa necessità, l'industria ha progressivamente abbandonato il paradigma monolitico in favore di approcci più modulari e flessibili. Tra questi, l'architettura a microservizi si è affermata come una delle soluzioni più efficaci

per affrontare la complessità delle moderne Applicazioni Web. Essa prevede la suddivisione dell'applicazione in una serie di servizi indipendenti, ciascuno dei quali incapsula una specifica funzionalità di business e comunica con gli altri tramite protocolli standard, tipicamente HTTP o sistemi di messaggistica asincrona.

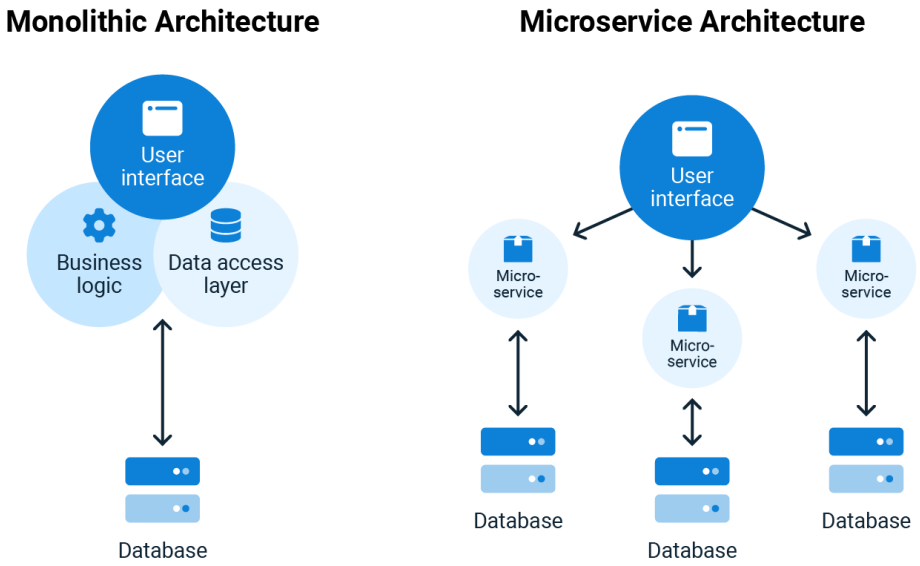

Figura 2.1: Architettura Monolitica comparata con l'architettura a Microservizi

Tale approccio comporta numerosi vantaggi. Ogni microservizio, infatti, può essere sviluppato, testato, rilasciato e scalato in maniera indipendente, riducendo il *time-to-market* e migliorando la resilienza del sistema. In caso di guasti localizzati, l'isolamento tra i servizi impedisce che un malfunzionamento in una componente comprometta l'intera applicazione. Inoltre, la suddivisione dei servizi consente ai team di sviluppo di concentrarsi sulla logica specifica del proprio dominio, senza dover necessariamente avere una visione globale dell'intero sistema, come richiesto invece nell'approccio monolitico.

2.2 Rilascio delle Web Applications: La storia

Agli albori degli anni 2000, le applicazioni web venivano distribuite e ospitate all'interno di server fisici privati, solitamente localizzati nei datacenter aziendali. Questo approccio, per quanto inizialmente adeguato, si è dimostrato rapidamente oneroso, rigido e scarsamente scalabile. Ogni macchina fisica, infatti, era dedicata all'esecuzione di una singola applicazione, con conseguente sotto-utilizzo delle risorse

Aspetto	**Architettura Monolitica**	**Architettura a Microservizi**
Architettura	Singolo Livello	Multi Livello
Dimensione	Grande, unico applicativo con tutti i componenti	Piccola, rilascio di tanti singoli microservizi
Rilascio	Rilascio dell'intero applicativo	Rilascio indipendente dei singoli microservizi
Scalabilità	Difficile da scalare orizzontalmente	Semplice da scalare orizzontalmente
Sviluppo	Semplice inizialmente	Inizialmente complesso per via dei tanti microservizi
Tecnologie	Scelte limitate	Possibilità di scegliere la tecnologia più adatta per ogni microservizio
Resistenza ai malfunzionamenti	Tutta l'applicazione può fallire se una parte fallisce	Fallimento isolato del singolo componente
Flessibilità	Scarsa, tutti i componenti sono strettamente legati	Molto flessibile, ogni componente può essere sviluppato, rilasciato e scalato indipendentemente dagli altri
Comunicazione	La comunicazione tra i componenti è semplice e veloce	La comunicazione tra i microservizi può essere più complessa e lenta per via delle chiamate di rete

Tabella 2.1: Comparativa tra l'architettura Monolitica e quella a Microservizi [6]

hardware disponibili. In un contesto storico caratterizzato dalla crescente diffusione di servizi digitali e dall'aumento della domanda di disponibilità e prestazioni, divenne evidente la necessità di adottare soluzioni più efficienti e flessibili per la gestione e il rilascio delle Web Applications.

In risposta a queste esigenze, nel decennio compreso tra il 2000 e il 2010 si è progressivamente affermato l'utilizzo delle macchine virtuali (Virtual Machines, o VM). Tali tecnologie, virtualizzando un intero sistema operativo sopra un'infrastruttura

condivisa, consentono di eseguire più ambienti isolati sulla medesima macchina fisica, garantendo separazione tra i processi e maggiore efficienza nell'utilizzo delle risorse. L'introduzione delle VM ha inoltre reso possibile simulare ambienti di produzione in fase di sviluppo o test, migliorando la portabilità e l'affidabilità del software lungo il ciclo di vita.

Tuttavia, intorno al 2010, l'adozione su larga scala delle VM ha evidenziato nuovi limiti: ciascuna macchina virtuale, infatti, replica un sistema operativo completo, determinando un notevole overhead in termini di risorse computazionali e memoria, nonché tempi di avvio piuttosto elevati. Inoltre, la gestione manuale delle configurazioni e il provisioning delle VM aumenta il rischio di errori e riduce l'agilità operativa.

È in questo contesto che, nel 2013, Docker ha rilanciato, rendendola accessibile su vasta scala, la tecnologia dei container, già esistente nei sistemi UNIX da diversi anni. Grazie alla Docker Container Platform, è stato possibile standardizzare e semplificare il processo di impacchettamento, distribuzione ed esecuzione delle applicazioni attraverso immagini containerizzate.

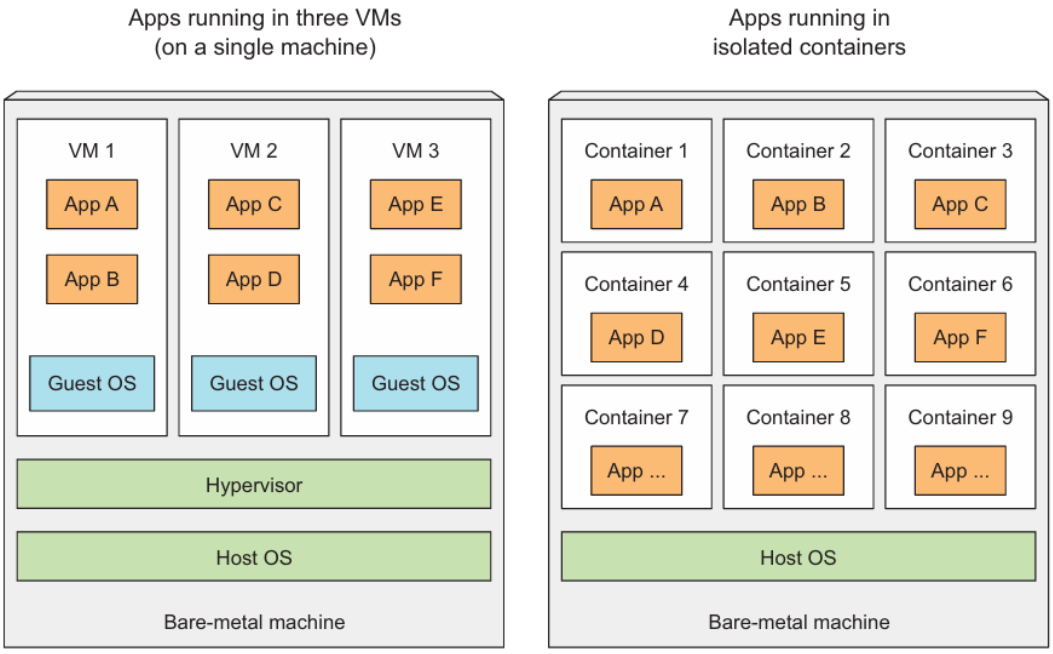

Figura 2.2: Confronto tra VMs e Containers

Un container consente di incapsulare l'applicativo e tutte le sue dipendenze in un'unità eseguibile, priva di sistema operativo completo, ma in grado di funzionare in modo coerente su qualunque host compatibile con il motore di containerizzazione. Il principale vantaggio risiede nella leggerezza e nella rapidità di avvio: mentre

una VM richiede diversi minuti per essere operativa, un container si avvia in pochi secondi, consentendo maggiore reattività ed un utilizzo ottimizzato delle risorse.

La portabilità garantita dai container, unita alla loro efficienza, ha reso questo paradigma un vero e proprio standard de facto per il rilascio delle applicazioni moderne. Parallelamente, Docker ha introdotto anche un primo orchestratore denominato Docker Compose, che consente di definire, configurare ed eseguire più container come un'unica applicazione, gestendo in maniera semplice le dipendenze e la comunicazione tra i vari servizi.

Nonostante il successo iniziale, intorno al 2015 Docker Compose ha mostrato i suoi limiti in ambienti ad alta disponibilità, specialmente nel contesto di Web Applications che richiedono servizi operativi 24 ore su 24. In particolare, Docker Compose non supporta nativamente la scalabilità orizzontale: essendo progettato per essere eseguito su un singolo nodo, l'aumento delle prestazioni richiede l'adozione di server via via più potenti, con conseguente incremento dei costi e ridotta efficienza rispetto a soluzioni basate su cluster distribuiti.

A fronte di tali criticità, Google ha reso open-source nel 2014 il progetto Kubernetes, originariamente sviluppato internamente per la gestione dei container su larga scala. Kubernetes ha rapidamente guadagnato consensi, affermandosi come nuovo standard per l'orchestrazione di container in ambienti distribuiti. Esso consente di astrarre l'infrastruttura sottostante, distribuendo dinamicamente i carichi di lavoro su più nodi e garantendo un elevato livello di affidabilità e scalabilità. Tra le funzionalità chiave introdotte da Kubernetes si annoverano:

- la sostituzione automatica dei container in caso di fallimento (*self-healing*);
- la scalabilità automatica dei servizi in base al carico (*horizontal pod autoscaling*);
- il rilascio progressivo delle applicazioni senza interruzioni di servizio (*rolling update*);
- la definizione dichiarativa dell'infrastruttura e dei flussi operativi.

2.3 Docker

2.3.1 Che cos'è Docker

Docker è una piattaforma software open source che consente la creazione, la distribuzione e l'esecuzione di applicazioni all'interno di container, ovvero unità eseguibili e portabili che racchiudono il codice dell'applicazione, le sue dipendenze, le librerie e le configurazioni necessarie per la sua corretta esecuzione in qualsiasi ambiente. Questo paradigma consente di risolvere in maniera strutturata il noto

problema denominato *"Works on my machine"*, garantendo coerenza tra ambiente di sviluppo, test e produzione.

L'adozione di Docker ha semplificato in modo significativo lo sviluppo e la gestione delle applicazioni distribuite. I container permettono di disaccoppiare il ciclo di vita delle applicazioni dall'ambiente sottostante, favorendo portabilità, scalabilità e standardizzazione. Pur essendo tecnicamente possibile creare e gestire container utilizzando le primitive native dei sistemi operativi Linux (ad esempio attraverso *cgroups* e *namespaces*), Docker ne ha abbattuto la complessità, fornendo strumenti ad alto livello per la gestione completa del ciclo di vita dei container. Questa semplicità d'uso ha contribuito in maniera determinante alla diffusione del paradigma, portando oltre 20 milioni di sviluppatori ad adottare mensilmente la piattaforma Docker. [7]

Un ulteriore vantaggio della containerizzazione risiede nell'isolamento del filesystem. Quando un'applicazione viene eseguita all'interno di un container Docker, essa vede un filesystem coerente con quello del sistema su cui è stata costruita l'immagine, indipendentemente dal sistema host su cui è in esecuzione. In tal modo, il container è isolato rispetto al sistema operativo dell'host e risulta immune da eventuali differenze di configurazione o ambienti non coerenti.

Docker adotta un modello a strati *(layered architecture)* nella costruzione delle immagini, in cui ogni layer rappresenta una modifica incrementale rispetto al precedente. I layer comuni tra più immagini possono essere condivisi, ottimizzando così l'utilizzo dello spazio su disco e velocizzando le operazioni di *build, pull e deploy.*

Quando due container condividono gli stessi layer di base, possono leggere gli stessi file senza la necessità di duplicarli. Tuttavia, per preservare l'isolamento, ogni container dispone di un layer in scrittura proprio, creato al momento dell'avvio. Nel caso in cui un processo nel container tenti di modificare un file proveniente da un layer in sola lettura, Docker applica il principio del copy-on-write: il file viene copiato nel layer in scrittura, modificato localmente e isolato dal container originario.

2.3.2 Docker: le limitazioni

Sebbene i container Docker offrano elevata portabilità e leggerezza, esistono alcune limitazioni strutturali che devono essere considerate. In primo luogo, i container non includono un kernel proprio, ma sfruttano il kernel del sistema operativo host. Questo implica che la compatibilità e il corretto funzionamento del container dipendano strettamente dalla versione e dai moduli del kernel presenti sulla macchina su cui esso viene eseguito. Pertanto, eventuali differenze a livello di kernel possono impedire l'esecuzione del container o determinare comportamenti inattesi.

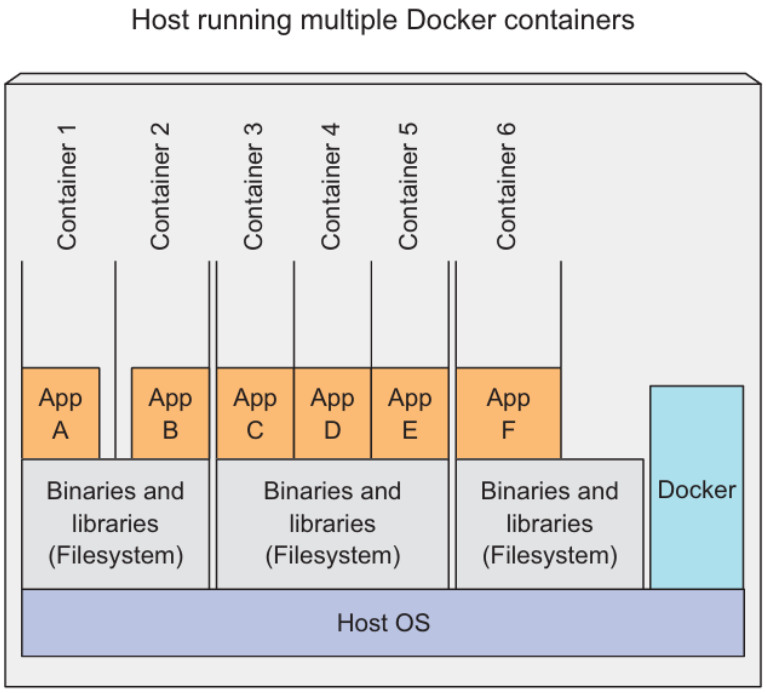

Figura 2.3: Condivisione dei layers tra container Docker

In secondo luogo, ogni container è creato per una specifica architettura hardware (ad esempio *x86*, *arm64*, ecc.) e non può essere eseguito su piattaforme diverse senza una specifica ricompilazione. Questo vincolo limita la portabilità dei container su dispositivi eterogenei, come nel caso del passaggio tra ambienti di sviluppo basati su CPU x86 e infrastrutture di produzione basate su architettura ARM (es. Raspberry Pi o Apple Silicon).

Una soluzione a questo problema è rappresentata dalla tecnica di *multi-platform build*, che consente di generare un'unica immagine Docker contenente più varianti architetturali. Questa strategia si avvale di strumenti come QEMU per l'emulazione delle architetture target e consente di realizzare immagini universali, eseguibili su sistemi differenti senza necessità di modifiche manuali.

2.3.3 Docker Compose

Docker Compose è uno strumento complementare alla piattaforma Docker, progettato per semplificare l'esecuzione di applicazioni multi-container. Tramite un file di configurazione in formato YAML, Docker Compose consente di definire in

maniera dichiarativa i servizi che compongono l'applicazione, i volumi, le reti e le relative dipendenze, rendendo possibile l'avvio dell'intera architettura con un solo comando.

Grazie a questa astrazione, Docker Compose ha trovato ampia applicazione in contesti di sviluppo e test, facilitando la configurazione di ambienti locali complessi e la replica di architetture di produzione in ambienti isolati.

Tuttavia, Docker Compose presenta limiti significativi che lo rendono inadatto in scenari di produzione su larga scala. In particolare:

- opera su una singola macchina, senza supporto nativo alla scalabilità orizzontale;
- non include meccanismi avanzati di orchestrazione, bilanciamento del carico, monitoraggio e gestione delle risorse;
- non è dotato di sistemi di auto-scaling né di funzionalità di *health-check* integrate;
- non prevede il riavvio automatico dei container in caso di fallimento;
- su sistemi Windows e macOS, l'esecuzione avviene tramite una macchina virtuale intermedia, con impatto negativo sulle prestazioni.

Tali limitazioni hanno determinato l'affermazione di sistemi di orchestrazione più sofisticati, come Kubernetes, in grado di soddisfare i requisiti stringenti di affidabilità, scalabilità e automazione richiesti dalle moderne architetture cloud-native.

2.4 Kubernetes

2.4.1 Che cos'è Kubernetes

Come discusso nei paragrafi precedenti, all'aumentare della complessità e delle dimensioni dei sistemi software, la gestione manuale e centralizzata delle applicazioni containerizzate diviene rapidamente insostenibile. Google è stata una delle prime realtà industriali ad affrontare su larga scala tale sfida, sviluppando internamente un sistema di orchestrazione denominato Borg, successivamente evoluto in Omega. Tali strumenti sono stati utilizzati per anni all'interno dell'infrastruttura di Google per il rilascio, la gestione e la scalabilità delle applicazioni.

Nel 2014, facendo tesoro di questa esperienza, Google ha reso open source il proprio sistema di orchestrazione di container con il nome di Kubernetes. Il termine deriva dal greco kubernetes, ovvero "timoniere", ed evoca la capacità del sistema di guidare ed orchestrare un insieme eterogeneo di applicazioni containerizzate.

Kubernetes è una piattaforma open source per l'orchestrazione di container che consente agli sviluppatori di distribuire, gestire e scalare applicazioni in modo efficiente, automatizzato e resiliente. Tra le principali peculiarità della piattaforma vi è la possibilità di creare cluster composti da più nodi (fisici o virtuali), sui quali è possibile distribuire e replicare i microservizi in maniera trasparente. Indipendentemente dalla dimensione o dalla complessità del cluster, il modello dichiarativo di rilascio delle applicazioni rimane coerente e uniforme, rendendo Kubernetes altamente scalabile e portabile.

2.4.2 Gli aspetti fondamentali

Un cluster Kubernetes è tipicamente costituito da un nodo di controllo (Control Plane) e da uno o più nodi di lavoro (Worker Nodes). Il Control Plane è responsabile del funzionamento complessivo del cluster, mentre i Worker Nodes sono incaricati dell'esecuzione effettiva dei container applicativi.

Attraverso file di configurazione descritti in formato YAML, lo sviluppatore specifica allo scheduler Kubernetes quali risorse distribuire e come comportarsi in caso di errore, aggiornamento o necessità di scalabilità. Il sistema si occupa poi di effettuare lo *scheduling* delle applicazioni sui nodi disponibili in modo automatico, in base alle politiche definite.

Tra le funzionalità native offerte da Kubernetes si annoverano:

- **Service Discovery e Networking**: le applicazioni possono individuarsi e comunicare attraverso nomi simbolici e un sistema DNS interno, indipendentemente dal nodo su cui risiedono.
- **Horizontal Pod Autoscaling**: il sistema monitora costantemente le risorse utilizzate (es. CPU, RAM) e adatta il numero di repliche dei servizi in funzione del carico.
- **Cluster Autoscaling**: in ambienti cloud, Kubernetes può scalare l'intero cluster, aggiungendo nuovi nodi in risposta a necessità di capacità computazionale.
- **Load Balancing**: le richieste in ingresso vengono distribuite tra le repliche di un servizio in base alla disponibilità, riducendo i colli di bottiglia.
- **Self-Healing**: in caso di malfunzionamenti, Kubernetes provvede automaticamente al riavvio dei container, al rescheduling su altri nodi o alla sostituzione dei pod.
- **Zero Downtime Deployments**: aggiornamenti progressivi dei componenti applicativi, con supporto al *rollback* in caso di errori.

Tali meccanismi permettono una gestione automatizzata dell'infrastruttura, garantendo affidabilità, disponibilità e ottimizzazione nell'utilizzo delle risorse.

2.4.3 L'architettura

Un cluster Kubernetes è strutturato secondo un'architettura composta da:

- **Control Plane**, che gestisce lo stato globale del cluster e coordina tutte le operazioni di pianificazione, monitoraggio e orchestrazione;
- **Worker Nodes**, che ospitano ed eseguono le applicazioni containerizzate sotto forma di pod.

I principali componenti del Control Plane sono:

- **Kubernetes API Server**: interfaccia RESTful che espone le operazioni disponibili e gestisce tutte le comunicazioni interne ed esterne al cluster;
- **Scheduler**: si occupa di decidere su quale nodo eseguire ogni nuovo Pod, sulla base delle risorse disponibili e dei vincoli definiti;
- **Controller Manager**: gestisce i controller responsabili della replica, della disponibilità e della riconciliazione dello stato desiderato del cluster;
- **etcd**: un database chiave-valore distribuito e ad alta disponibilità utilizzato per memorizzare in maniera consistente la configurazione e lo stato del cluster.

I nodi di lavoro, o Worker Nodes, comprendono:

- **Container Runtime**: componente responsabile dell'esecuzione dei container. Kubernetes supporta diversi runtime, tra cui `containerd`, che ha sostituito ufficialmente `Docker` a partire dal 2022;
- **Kubelet**: demone in esecuzione su ogni nodo, comunica con l'API Server per ricevere istruzioni e assicura che i container siano eseguiti correttamente;
- **Kube-proxy**: gestisce il traffico di rete all'interno del cluster, implementando meccanismi di bilanciamento del carico (load balancing) e inoltro delle richieste.

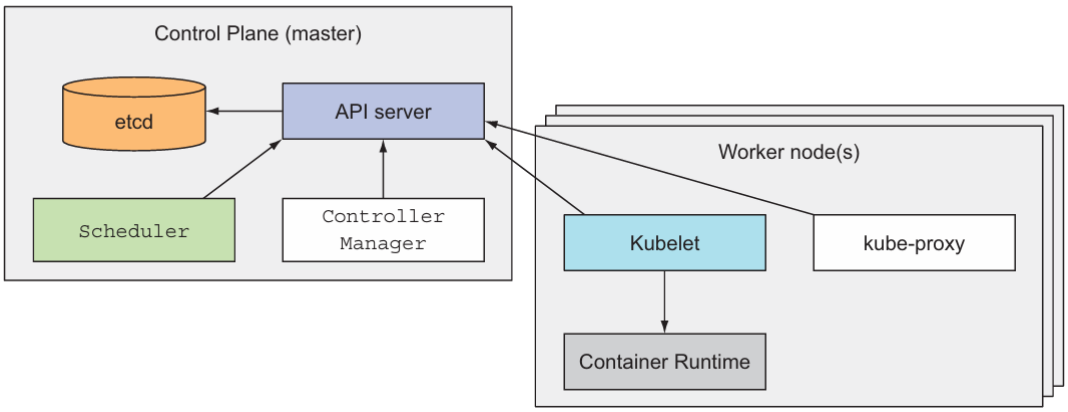

Figura 2.4: I componenti di Kubernetes

2.4.4 I principali componenti

Pods

Il Pod rappresenta l'unità minima schedulabile in Kubernetes. Ogni pod incapsula uno o più container che condividono lo stesso spazio dei nomi di rete, indirizzo IP, e possono accedere a volumi condivisi. Sebbene tecnicamente sia possibile eseguire più container all'interno di un singolo pod, la prassi prevede di avere container singoli per favorire isolamento, scalabilità e manutenzione.

La scelta di raggruppare più container in un pod è giustificata in casi specifici, ad esempio quando i container devono condividere risorse locali come file temporanei o devono cooperare strettamente (es. sidecar pattern). In tal caso, Kubernetes garantisce il riavvio coordinato dei container tramite il controllo del pod nel suo insieme.

I pod sono pensati per essere temporanei ed effimeri. Per garantire resilienza e disponibilità, è consigliabile evitare il loro utilizzo isolato in produzione, ed affidarli a controller come ReplicaSet o Deployment.

ReplicaSets e Deployments

Un ReplicaSet è un controller che garantisce che un numero specifico di repliche identiche di un pod sia sempre in esecuzione nel cluster. In caso di fallimento di un pod (o del nodo che lo ospita), il ReplicaSet provvede automaticamente alla sua ricreazione su un nodo disponibile.

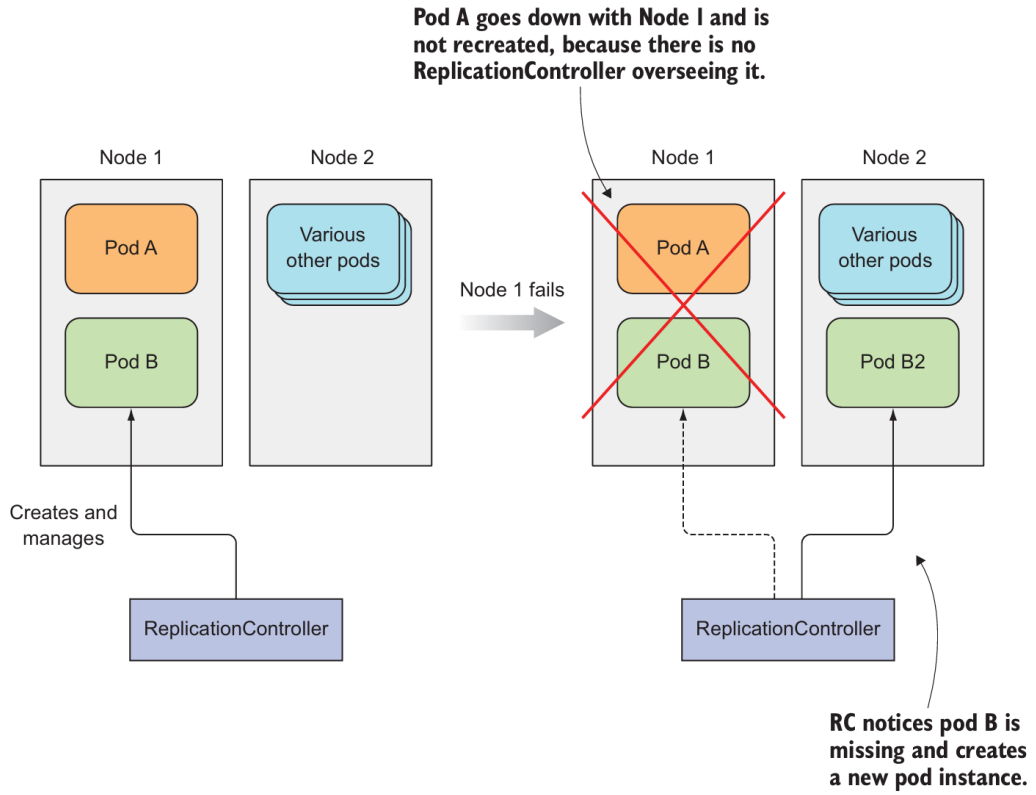

Figura 2.5: ReplicaSets

Sebbene i ReplicaSet costituiscano una soluzione efficace per il mantenimento dell'alta disponibilità, la gestione degli aggiornamenti (rollout) rimane complessa. A tale scopo, Kubernetes introduce il concetto di Deployment.

Un Deployment è un controller di più alto livello che automatizza:

- la creazione e la gestione di ReplicaSet;
- il rollout progressivo di nuove versioni dei pod;
- il rollback in caso di fallimenti;
- la scalabilità orizzontale.

Durante un aggiornamento, il Deployment crea un nuovo ReplicaSet con la nuova configurazione e progressivamente sostituisce i pod del vecchio ReplicaSet, garantendo continuità del servizio (strategia rolling update). In caso di problemi, Kubernetes può eseguire automaticamente un rollback alla versione precedente.

Figura 2.6: ReplicaSet controllato dal Deployment

L'utilizzo dei Deployment costituisce quindi la pratica consigliata per il rilascio e l'evoluzione di componenti applicativi in ambienti di produzione.

DaemonSets

Nel paradigma di orchestrazione offerto da Kubernetes, le risorse di tipo *ReplicaSet* sono pensate per garantire un numero specifico di repliche di un'applicazione su un insieme generico di nodi. Tuttavia, vi sono scenari in cui si desidera che esista una ed una sola istanza di un determinato Pod su ogni nodo del cluster. Un esempio classico è rappresentato da servizi di tipo gateway o da agenti di monitoraggio locale, i quali devono essere disponibili in modo uniforme su tutti i nodi per garantire funzionalità come bilanciamento del traffico o raccolta di metriche.

Per supportare tale esigenza, Kubernetes introduce la risorsa di tipo DaemonSet. Un DaemonSet assicura che una copia del Pod specificato venga eseguita su ogni nodo del cluster, e si occupa automaticamente di avviare una nuova istanza qualora venga aggiunto un nuovo nodo. Inoltre, l'ambito di applicazione del DaemonSet può essere ristretto ad un sottoinsieme specifico dei nodi, mediante l'uso di etichette e *nodeSelectors*.

Questa risorsa risulta quindi fondamentale per garantire uniformità di comportamento in applicazioni che richiedono la presenza costante di un determinato servizio su ogni nodo.

Jobs e CronJobs

Le risorse Kubernetes discusse fino a questo punto , come Deployment, ReplicaSet e DaemonSet, sono state progettate per garantire l'esecuzione continua dei container. Tali risorse si aspettano che l'applicazione sia persistente, ovvero attiva e in ascolto (come accade tipicamente per server HTTP o gateway di rete). Qualora il container termini, il sistema provvede a riavviarlo automaticamente.

Tuttavia, vi sono casi d'uso in cui si desidera eseguire un'attività computazionale di durata limitata, come un processo batch o una migrazione di dati, che deve completarsi una sola volta. Per questi scenari, Kubernetes fornisce la risorsa Job.

Un Job è un oggetto che garantisce l'esecuzione di uno o più Pod fino al completamento con successo del task assegnato. È possibile configurare la politica di riavvio per gestire errori transitori e definire il numero massimo di tentativi. Al completamento del task, il Pod non viene più riavviato.

Per attività pianificate o ricorrenti nel tempo, Kubernetes mette a disposizione la risorsa CronJob, che estende le funzionalità del Job mediante un sistema di scheduling simile alla sintassi UNIX del comando cron. Un CronJob consente l'esecuzione periodica di Job in momenti temporali ben definiti o ad intervalli regolari, supportando così scenari di automazione come backup, aggiornamenti, o generazione di report.

Services

Nel contesto di un'architettura a microservizi, uno degli aspetti più critici è rappresentato dalla comunicazione tra i diversi componenti applicativi. In Kubernetes, i Pod, pur rappresentando unità autonome, sono soggetti ad una dinamica di ciclo di vita altamente variabile. Essi possono essere ricreati frequentemente, cambiando di conseguenza l'indirizzo IP assegnato. Questa natura effimera rende inadeguato un modello di comunicazione basato su indirizzamenti statici.

Per risolvere questa criticità, Kubernetes introduce la risorsa denominata Service. Un Service agisce come astrazione stabile per un gruppo di Pod che erogano la stessa funzionalità, fornendo un punto di accesso unico e costante, rappresentato da un nome simbolico e da un indirizzo IP virtuale statico. Il Service si occupa automaticamente di aggiornare la lista dei Pod attivi tramite meccanismi di label selector, inoltrando il traffico in ingresso verso uno dei Pod disponibili.

La risoluzione dei nomi dei Service in IP è garantita dal componente interno kube-dns (o CoreDNS), il quale funge da server DNS per l'intero cluster. Grazie a ciò, i microservizi possono comunicare tra loro utilizzando semplici nomi logici, indipendentemente dall'indirizzo IP effettivo assegnato ai Pod sottostanti.

In aggiunta, Kubernetes consente di configurare il comportamento del bilanciamento del carico attraverso parametri come `sessionAffinity`, che consente di mantenere la persistenza della sessione verso lo stesso Pod per le richieste provenienti dallo stesso client.

Kubernetes supporta diversi tipi di Service, ognuno progettato per differenti scenari d'uso:

- **ClusterIP** (default): espone il servizio solo internamente al cluster. È la scelta ideale per la comunicazione tra microservizi.
- **NodePort**: rende il servizio accessibile dall'esterno del cluster esponendo una porta specifica su ciascun nodo. L'accesso avviene tramite *<IP-pubblico-nodo>:<porta>*.

- **LoadBalancer**: crea un bilanciatore di carico esterno sfruttando le API del cloud provider sottostante, esponendo il servizio a Internet in modo trasparente.

Nei contesti produttivi, il tipo **LoadBalancer** rappresenta la soluzione più robusta, in quanto garantisce la distribuzione omogenea del carico. Tuttavia, in ambienti on-premise o privi del supporto nativo a LoadBalancer, è possibile simulare tale comportamento utilizzando un Service di tipo **ClusterIP**, in combinazione con il campo `externalIPs` per definire indirizzi IP statici dei nodi. In tal caso, per assicurare la corretta gestione del traffico esterno, è necessario configurare il servizio come ClusterIP e specificare nei campi di configurazione del Service "spec.externalIPs" gli indirizzi IP dei nodi del cluster. Questo permette di istruire le IP tables nell'istradare i pacchetti diretti agli IP specificati in externalIPs verso il Service. Fatto ciò, è necessario configurare la externalTrafficPolicy come "Local" per evitare un redirect della connessione da un pod ad un'altro, cambiando quindi l'ip sorgente. L'internalClusterPolicy, invece, può essere configurata come "Cluster" per permettere alle connessioni interne di distribuire il carico sui vari pods.

Volumes

Come già accennato in precedenza, i container in Kubernetes sono entità effimere, soggette a riavvio o terminazione in modo arbitrario a causa di errori o logiche di aggiornamento. L'introduzione del concetto di Pod ha parzialmente risolto tale criticità, garantendo il riavvio automatico dei container da parte di Kubernetes. Tuttavia, ciò non risolve il problema della persistenza dello stato. Infatti, quando un container viene ricreato, i dati scritti nella memoria volatile andranno irrimediabilmente persi, compromettendo l'integrità dell'applicazione se essa dipende da uno stato persistente.

Per rispondere a questa esigenza, Kubernetes introduce le risorse di tipo Volume, che rappresentano unità di archiviazione accessibili dai container presenti all'interno di uno stesso Pod. Il volume è creato contestualmente al ciclo di vita del Pod che lo contiene, e rimane disponibile per tutti i container del Pod fino alla terminazione dello stesso. In questo modo, anche se un container viene riavviato, può continuare ad accedere ai dati precedentemente scritti nel volume, garantendo continuità operativa.

Un volume può essere montato in qualsiasi punto del filesystem del container ed è accessibile con le normali modalità di lettura e scrittura del file system. Kubernetes supporta numerosi tipi di volume, ciascuno pensato per scenari specifici:

- **emptyDir**: directory temporanea creata all'avvio del Pod, utile per dati transitori.
- **nfs**: volume condiviso attraverso il protocollo NFS, utile per l'accesso simultaneo da più Pod.

- **gcePersistentDisk**, **awsElasticBlockStore**, **azureDisk**: volumi integrati con i principali cloud provider, per supportare storage persistente.

- **configMap**, **secret**: volumi speciali per l'esposizione di dati di configurazione o credenziali.

Uno degli obiettivi principali nella gestione dello storage in Kubernetes è la separazione tra il ciclo di vita del Pod e quello del volume. A tal fine, vengono introdotte le risorse PersistentVolume (PV) e PersistentVolumeClaim (PVC) per garantire che il Volume non venga cancellato in caso di riavvio di un Pod. Il PV rappresenta una risorsa di archiviazione persistente nel cluster, mentre il PVC costituisce la richiesta da parte di un Pod per ottenere uno storage con determinate caratteristiche (dimensione, modalità di accesso, performance).

Kubernetes supporta anche il cosiddetto provisioning dinamico, abilitato mediante le **StorageClass**. Quando il PVC viene creato, Kubernetes può automaticamente generare un volume fisico che soddisfi i requisiti richiesti, eliminando la necessità di creare manualmente i PersistentVolume. In assenza di StorageClass, l'amministratore del cluster dovrà provvedere a definire esplicitamente i PV prima che possano essere utilizzati dai Pod attraverso i rispettivi PVC.

ConfigMaps e Secrets

Nel contesto dei microservizi, è comune che i container necessitino di parametri esterni per poter funzionare correttamente, come ad esempio le credenziali per l'accesso ad un database, la configurazione della porta d'ascolto, endpoint remoti o altre variabili di ambiente. Codificare tali informazioni all'interno dell'immagine del container risulta una pratica non ottimale, poiché impone la ricostruzione e la ridistribuzione dell'immagine ad ogni variazione dei parametri. Inoltre, l'inclusione di dati sensibili direttamente nel container può costituire un grave rischio per la sicurezza.

Per ovviare a tali problematiche, Kubernetes mette a disposizione due risorse specializzate: ConfigMap e Secret.

La risorsa **ConfigMap** consente di esternalizzare i parametri di configurazione da un'applicazione, rappresentandoli come coppie chiave-valore. Essa può essere montata come volume all'interno del container oppure iniettata direttamente come variabili d'ambiente. In questo modo, è possibile aggiornare la configurazione del servizio senza la necessità di rigenerare l'immagine Docker, incrementando notevolmente la manutenibilità dell'ambiente.

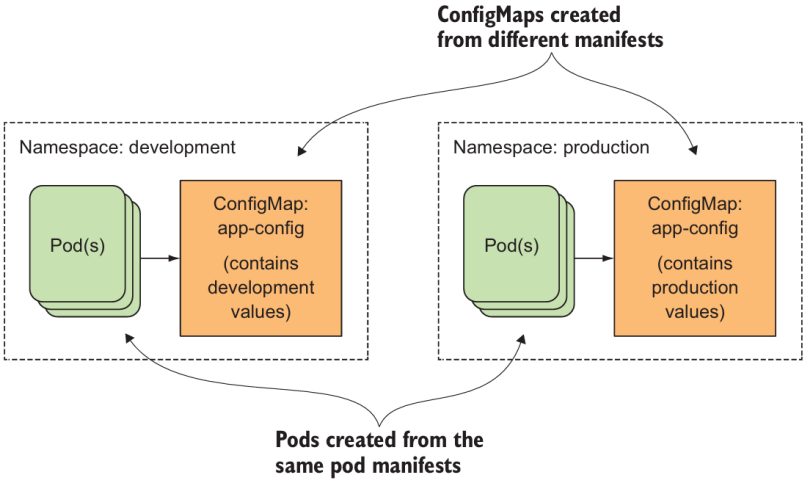

Figura 2.7: Uso di due diverse configMap per la configurazione di due ambienti diversi

Per la gestione di informazioni riservate, Kubernetes fornisce le risorse di tipo **Secret**. I Secret condividono la stessa struttura logica delle ConfigMap, ma i loro valori vengono codificati in Base64 e possono opzionalmente essere cifrati per garantire un ulteriore livello di protezione. I Secret vengono montati nei Pod che ne fanno esplicita richiesta, e sono accessibili solo a questi, grazie all'integrazione con il sistema di controllo degli accessi (RBAC). Inoltre, i dati non vengono scritti su disco ma mantenuti in memoria volatile, rendendo il sistema più sicuro contro attacchi diretti al filesystem.

La gestione centralizzata e sicura dei Secret è affidata al componente **kube-apiserver**, che li archivia all'interno del database distribuito *etcd*. È buona prassi abilitare la cifratura dei dati a riposo in etcd per rafforzare ulteriormente la sicurezza.

Infine, si osservi che alla creazione di un Pod, Kubernetes genera automaticamente alcuni Secret che ne permettono l'autenticazione verso l'API Server del cluster. Tali meccanismi risultano essenziali per l'implementazione sicura di interazioni interne all'ambiente di esecuzione Kubernetes.

Capitolo 3

Ottimizzazioni

3.1 Ottimizzazioni nel contesto delle Applicazioni Web in Cloud

Nel contesto dello sviluppo software moderno, l'ottimizzazione rappresenta un tema centrale, ampiamente trattato sia in ambito accademico che industriale. Corsi universitari, letteratura specializzata e pratiche ingegneristiche pongono grande enfasi sulla scrittura di codice efficiente, basato sulla scelta consapevole di strutture dati e algoritmi adeguati al problema da risolvere. Tuttavia, quando si entra nell'ambito delle Applicazioni Web, la questione dell'efficienza assume una rilevanza ancora maggiore per due motivi principali.

Il primo riguarda l'esperienza utente. In un'applicazione web, la reattività e la velocità di risposta del sistema rappresentano fattori critici per la soddisfazione dell'utente finale. Una piattaforma lenta o poco reattiva può portare l'utente ad abbandonare il servizio, indipendentemente dalla qualità intrinseca dell'applicazione o dal valore del contenuto offerto. In tale contesto, l'ottimizzazione delle performance non è più soltanto una questione tecnica, ma assume una valenza strategica e competitiva.

Il secondo aspetto è di tipo economico e riguarda i modelli di costo del cloud computing. L'esecuzione di microservizi in ambienti cloud comporta spesso un modello di pagamento basato sul consumo effettivo delle risorse. Ciò significa che l'efficienza computazionale ha un impatto diretto sui costi operativi dell'infrastruttura: microservizi più rapidi ed efficienti determinano una riduzione del tempo di utilizzo delle risorse (CPU, memoria, rete), con conseguente abbattimento delle spese correlate.

Alla luce di queste considerazioni, l'obiettivo primario nello sviluppo di Web Application moderne risiede nella costruzione di microservizi ottimizzati, capaci di garantire avvii rapidi, esecuzione efficiente e interazione fluida con l'utente

finale, riducendo al contempo i costi di esercizio. Tali ottimizzazioni devono essere considerate fin dalle fasi iniziali del processo di progettazione e devono guidare le scelte architetturali, tecnologiche e infrastrutturali.

3.2 Ottimizzazione dei Microservizi

Come discusso nel Capitolo 2, le moderne Web Application adottano architetture a microservizi, in cui ogni componente svolge un ruolo autonomo ma interdipendente. Una delle sfide principali di questo paradigma risiede nella propagazione dei ritardi: un singolo microservizio inefficiente può degradare le performance dell'intero sistema, introducendo latenze lungo tutta la catena di chiamate.

Pertanto, risulta cruciale progettare microservizi che non solo siano correttamente isolati e scalabili, ma che siano anche altamente performanti. In tale ambito, la scelta del framework e della piattaforma tecnologica incide in modo significativo sull'efficienza complessiva del sistema. Framework pensati per l'ambiente cloud-native, con tempi di avvio contenuti, basso consumo di risorse e ottimizzazioni specifiche per la containerizzazione, rappresentano oggi un fattore chiave per ottenere *Web Applications* realmente reattive, affidabili ed economiche.

Un framework è una piattaforma software che fornisce un insieme strutturato di componenti, librerie, convenzioni e strumenti pensati per facilitare, standardizzare e velocizzare la creazione di applicazioni lato server. Formalmente, può essere definito come un'infrastruttura software riusabile che implementa pattern architetturali comuni e fornisce un ciclo di vita applicativo gestito, basato sul principio di inversione del controllo, per lo sviluppo di applicazioni web scalabili e manutenibili.

In questo ambito, i framework come Spring Boot e Quarkus sono i più diffusi per la piattaforma Java, poichè pensati per la realizzazione di servizi web RESTful, microservizi ed applicazioni cloud-native. Entrambi adottano il paradigma della dependency injection, la gestione automatica del ciclo di vita dei componenti , la configurazione tramite convenzione e il supporto nativo a specifiche Java come JAX-RS, CDI, e JPA.

Questi framework non solo forniscono gli strumenti necessari per l'esposizione di API, l'accesso ai dati e la sicurezza, ma integrano anche meccanismi per il packaging, il deployment e l'observability, risultando essenziali per lo sviluppo moderno in ambienti DevOps e cloud-native.

3.2.1 SpringBoot

Negli ultimi anni, Spring Boot si è affermato come uno dei framework più diffusi e consolidati per lo sviluppo di applicazioni web e microservizi basati sulla piattaforma Java. Introdotto dal progetto Spring nel 2014, Spring Boot nasce con l'obiettivo di semplificare drasticamente il processo di configurazione e *bootstrap* delle applicazioni

Spring tradizionali, riducendo la complessità intrinseca e la verbosità del codice *boilerplate*. [8]

Spring Boot si basa su una serie di principi chiave che ne hanno decretato il successo:

- ***Convention over configuration***: l'applicazione adotta comportamenti predefiniti sensibili al contesto, minimizzando la necessità di configurazioni esplicite.

- **Autoconfigurazione**: grazie al meccanismo di autoconfigurazione, il framework è in grado di attivare dinamicamente *bean* e componenti in base alle dipendenze presenti.

- **Supporto completo allo Spring Ecosystem**: Spring Boot integra nativamente moduli chiave come Spring MVC, Spring Data, Spring Security e Spring Cloud, rendendolo una piattaforma solida e completa per architetture monolitiche, a microservizi e cloud-native.

Grazie a queste caratteristiche, Spring Boot è diventato lo standard de facto per lo sviluppo Java nel contesto enterprise, adottato da una vasta comunità di sviluppatori, aziende e progetti open source.

Tuttavia, con la crescente adozione di architetture basate su container, funzioni serverless e piattaforme cloud-native come Kubernetes, è emersa l'esigenza di framework più leggeri, reattivi e ottimizzati per tempi di avvio ridotti e footprint minimo.

In questo contesto si inserisce Quarkus, che nasce per rispondere a queste nuove sfide, offrendo un'alternativa moderna e performante a Spring Boot.

3.2.2 Quarkus

Quarkus è un moderno framework Java open source, sviluppato da Red Hat e introdotto nel 2019, progettato specificamente per soddisfare i requisiti delle architetture cloud-native, serverless e container-based, con un focus particolare sulla bassa latenza, rapido tempo di avvio e ridotto consumo di memoria.

Il framework si presenta come una "Supersonic Subatomic Java Platform", un'espressione coniata dal team di Quarkus per enfatizzare le sue prestazioni eccezionali in ambienti dinamici e altamente scalabili, come Kubernetes e OpenShift.

Quarkus sposta quanto più possibile l'elaborazione nella fase di pacchettizzazione per risparmiare risorse durante l'esecuzione. Così facendo l'applicazione risulta più veloce ed al contempo occupa meno spazio. [9]

Le principali caratteristiche distintive di Quarkus includono:

- **Pronto per il cloud e Kubernetes**: essendo un framework pensato e sviluppato per il cloud, Quarkus include strumenti specificamente pensati per essere eseguiti su kubernetes, pertanto tutto è ottimizzato per la piattaforma in questione. Ad esempio, include nativamente strumenti per la generazione di manifest Kubernetes, health checks e metriche, rendendolo ideale per le moderne applicazioni distribuite.

- **Pensato per i containers**: l'industria del software negli ultimi anni ha mostrato scetticismo nell'uso di Java all'interno dei containers. Infatti i container, in ambiente cloud, necessitano continuamente di riavviarsi ed hanno la necessità di avere dimensioni ridotte poichè queste metriche vengono tradotte direttamente in costi del servizio cloud. Java non è pensato per avere tempi di avvio minimi e soprattutto l'utilizzo della Java Virtual Machine, l'ambiente di esecuizione in cui far eseguire il bytecode generato, aumenta, e di non poco, le dimensioni dell'applicazione. Quarkus, mediante la Compilazione *Ahead-of-Time* (AOT) e il supporto nativo a GraalVM, consente la generazione di eseguibili nativi riducendo drasticamente i tempi di avvio, fino a pochi millisecondi, e il consumo di memoria, rispetto all'approccio Just-In-Time (JIT) tipico della JVM.

- **OpenJDK HotSpot e GraalVM**: si riferiscono rispettivamente alla possibilità di usare la Java Virtual Machine o la possibilità di creare degli eseguibili nativi in grado di essere avviati direttamente, rimuovendo l' *overhead* portato dalla JVM. Solitamente la creazione di applicazioni native è un processo tedioso che spesso induce ad errori. Con il supporto diretto a GraalVM, in Quarkus si riesce a creare un'applicazione nativa inserendo un semplice *flag*.

- **Dev Mode avanzata** : grazie al riavvio continuo ed automatico, la Dev Mode di Quarkus permette di migliorare l'esperienza dello sviluppatore applicando dinamicamente le modifiche apportate al codice o alle configurazioni, permettendo di osservare *live* i risultati dei cambiamenti senza la necessità di riavviare l'applicazione.

- **Estensioni modulari e configurazione semplificata**: il sistema di estensioni, che ne conta ben più di 700, consente una gestione automatica delle dipendenze e l'integrazione di tecnologie di terze parti (Hibernate ORM, RESTEasy, Vert.x, Kafka, OpenAPI).

Oltretutto, la compatibilità con standard Java EE / Jakarta EE garantisce una curva di apprendimento ridotta per gli sviluppatori provenienti da ambienti Spring o Java tradizionali

3.2.3 Comparativa SpringBoot - Quarkus

A differenza di Spring Boot, che nasce in un'epoca precedente alla diffusione massiva del paradigma container-based, Quarkus adotta una filosofia "container-first" e "native-first", ponendosi come un'alternativa ottimizzata al classico stack enterprise.

Tuttavia l'ecosistema Spring, essendo in uso da circa 20 anni, porta con sè una grande maturità e, soprattutto, una numerosa ed attiva *community*.

Anche SpringBoot, dalla versione 3, supporta lo sviluppo di applicazioni native mediante GraalVM nonostante, a volte, possano sorgere incompatibilità. Quarkus è stato creato per supportare nativamente GraalVM, pertanto alcune problematiche che l'ecosistema Spring riscontra tutt'oggi, come ad esempio la *reflection*, sono spesso risolte in Quarkus.

Il grafico 3.1 disponibile sul sito ufficiale di Quarkus, mostra come il framework in questione riesca a portare miglioramenti notevoli in termini di tempi di risposta e spazio occupato dall'applicazione rispetto agli standard dell'approccio tradizionale. Tuttavia sul sito non viene specificato cosa si intende per "approccio tradizionale". Nasce quindi l'esigenza di ottenere dei dati reali sulle differenze tra SpringBoot e Quarkus.

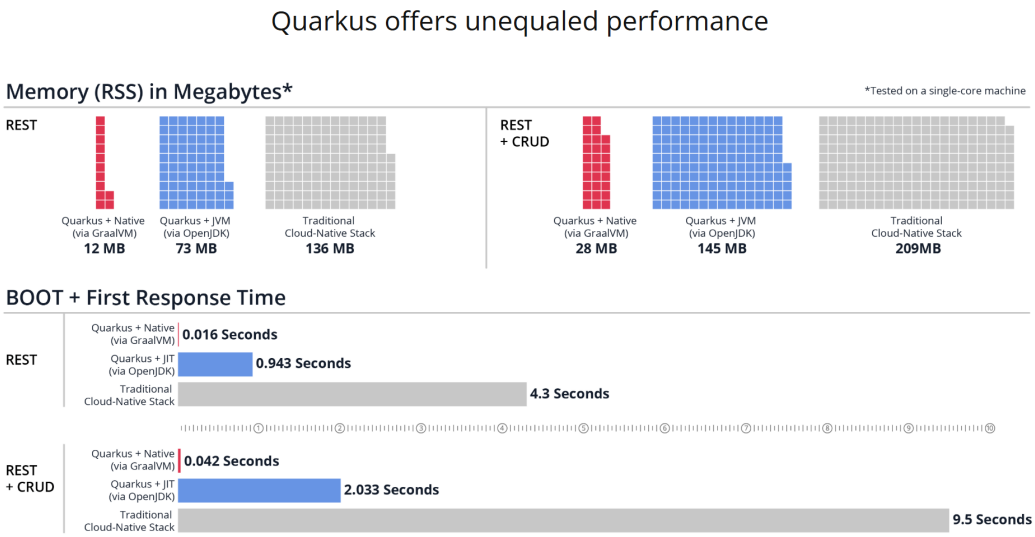

Figura 3.1: Quarkus: le performance riportate sul sito ufficiale

A tale scopo, sono state sviluppate due applicazioni completamente identiche dal punto di vista logico ma la prima è stata creata con SpringBoot mentre la seconda utilizza Quarkus. Entrambe adoperano lo stesso storage, Postgres, e sono state testate inoltrando le stesse richieste all'applicazione. I dati sono stati generati

analizzando i tempi di esecuzione sia per query semplici che per quelle più complesse con join tra varie tabelle, sfruttando i meccanismi offerti da entrambi i framework per la connessione alle fonti di dato. La comparativa è stata portata avanti tra le versioni native delle applicazioni. Per avere una soglia con la quale comparare i risultati, l'applicazione SpringBoot è stata testata anche nella versione JVM, ovvero lo standard delle applicazioni odierne. Così facendo si riesce ad avere un riscontro sulle migliorie apportate dalla compilazione native ed allo stesso tempo è possibile comparare l'efficienza di Quarkus rispetto a SpringBoot. Nella comparativa, viene mostrata anche la stessa Applicazione Web ma nella versione *reactive* per stabilire come si comportano i due framework se utilizzati nella modalità reattiva.

Metrica	**JVM SpringBoot**	**Nativa SpringBoot**	**JVM Quarkus**	**Nativa Quarkus**
Tempo di build	∼20s	2m 38s	∼20s	1m 46s
Tempo di avvio	3.527s	0.266s	2.221s	0.076s
Dimensione immagine	307 MB	248 MB	278 MB	216.4 MB
Interrogazione semplice	149.01 ms	2.40 ms	138.02 ms	2.06 ms
Interrogazione complessa	76.28 ms	16.93 ms	77.12 ms	6.68 ms

Tabella 3.1: Comparativa Performance: SpringBoot - Quarkus

Dalla tabella 3.2 si può osservare il potenziamento portato dalla compilazione nativa dell'applicazione. I tempi di avvio, computazione e la dimensione dell'immagine sono fortemente ottimizzati rispetto allo standard JVM SpringBoot. Ciò risulta possibile per via dello sforzo fatto durante la fase di compilazione che pertanto risulta estremamente lenta. Tuttavia ciò non dovrebbe essere considerato un difetto degno di nota poichè la compilazione è un processo che ha luogo solo *una tantum*, per creare l'immagine da utilizzare nei container. Gli altri fattori invece sono fondamentali per migliorare le performance delle applicazioni moderne. Come si può vedere nella Tabella 3.1, Quarkus risulta essere più efficiente di SpringBoot anche nelle versioni con Java Virtual Machine.

Per un'ulteriore conferma, si è ritenuto opportuno valutare le prestazioni delle applicazioni se sottoposte ad un carico di mille richieste contemporanee. Così facendo si è potuto osservare il comportamento delle applicazioni in termini di tempo di risposta e memoria RAM utilizzata. Nelle tabelle 3.4 e 3.5 si possono osservare

Fattore di Miglioramento	JVM SpringBoot	JVM Quarkus	SpringBoot Nativa	Quarkus Nativa
Tempo di build	1	1	7.9x	5.3x
Tempo di avvio	1	1,6x	13x	46.5x
Dimensione immagine	1	-10%	-19.2%	-29.5%
Interrogazione semplice	1	1,1x	62x	72.5x
Interrogazione complessa	1	1	4.5x	11.4x

Tabella 3.2: Fattore di miglioramento delle versioni non *reactive* rispetto allo standard SpringBoot JVM

Metrica	JVM SpringBoot Reactive	Native SpringBoot Reactive	Native Quarkus Reactive
Tempo di build	27s	2m 20s	1m 47s
Tempo di avvio	2.554s	0.112s	0.097s
Dimensione immagine	253.2 MB	137.7 MB	223.6 MB
Interrogazione semplice	24.61 ms	4.90 ms	17.16 ms
Interrogazione complessa	58.04 ms	10.42 ms	6.30 ms

Tabella 3.3: Comparativa Performance Reactive: SpringBoot - Quarkus

i comportamenti degli applicativi rispettivamente nell'approccio tradizionale e reactive.

Nei grafici 3.2, 3.4, 3.3 e 3.5 si può osservare l'andamento del tempo di esecuzione con l'evolversi del numero di richieste e nei la loro distribuzione.

Metrica	**JVM SpringBoot**	**Native SpringBoot**	**Native Quarkus**
Min Tempo di Risposta	12.5848 ms	9.2132 ms	2.6838 ms
Max Tempo di Risposta	1024.5768 ms	959.7945 ms	1385.3752 ms
Avg Tempo di Risposta	376.02 ms	292.02 ms	246.98 ms
Min Uso della memoria (a riposo)	379.2 MiB	73.91 MiB	12.82 MiB
Min Uso della memoria (con carico di 1000 richieste)	645 MiB	174.6 MiB	168 MiB

Tabella 3.4: Comparativa Performance con un carico di 1000 richieste: SpringBoot - Quarkus

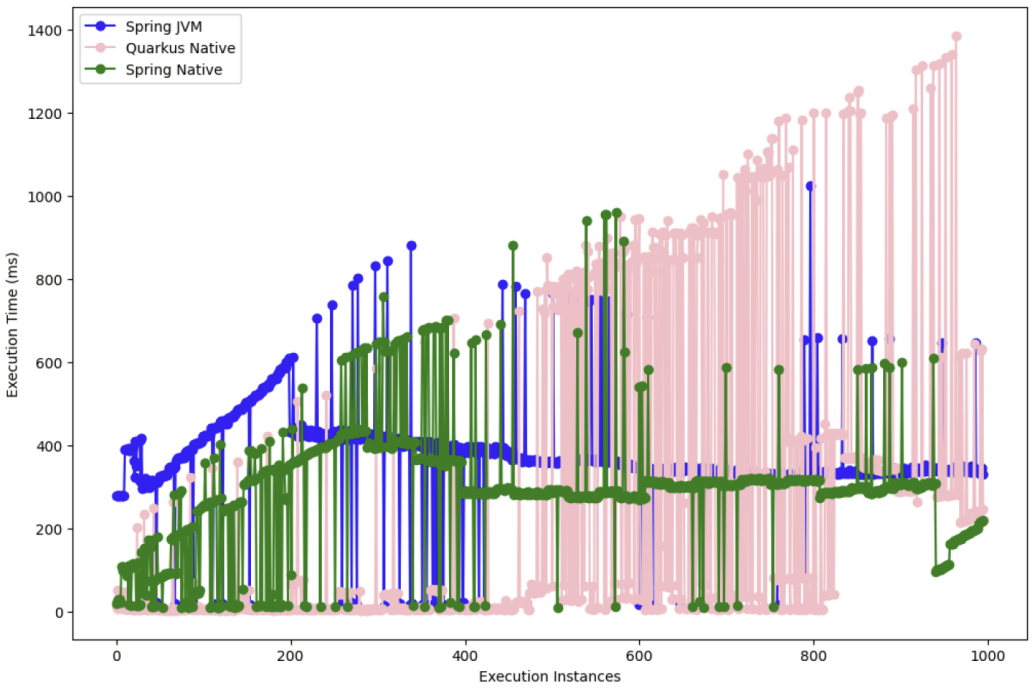

Figura 3.2: Evoluzione dei tempi di esecuzione con un carico di 1000 richieste

Metrica	**JVM SpringBoot Reactive**	**Native SpringBoot Reactive**	**Native Quarkus Reactive**
Min Tempo di Risposta	1332.4881 ms	389.0493 ms	138.7672 ms
Max Tempo di Risposta	4912.676 ms	7664.2005 ms	1614.0334 ms
Avg Tempo di Risposta	3508.02 ms	4814.71 ms	888.30 ms
Min Uso della memoria (a riposo)	325.6 MiB	160.8 MiB	16.72 MiB
Min Uso della memoria (con carico di 1000 richieste)	1.413 GiB	749.2 Mib	110.06 MiB

Tabella 3.5: Comparativa Performance Reactive con un carico di 1000 richieste: SpringBoot - Quarkus

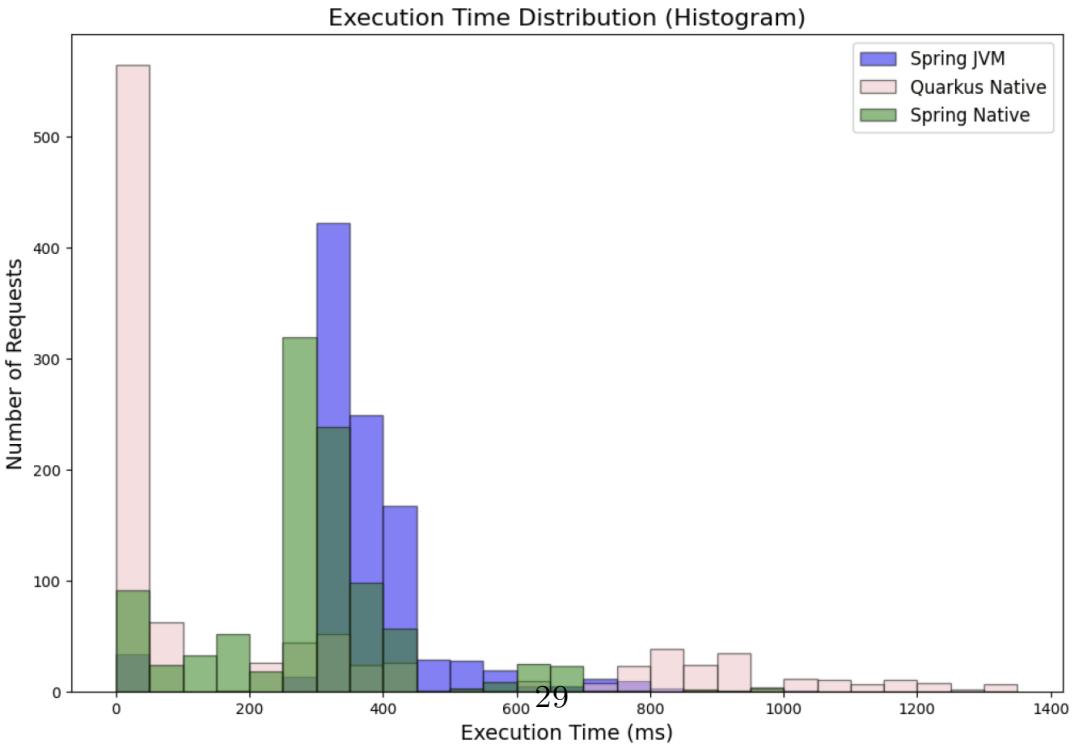

Figura 3.3: Distribuzione dei tempi di esecuzione con un carico di 1000 richieste

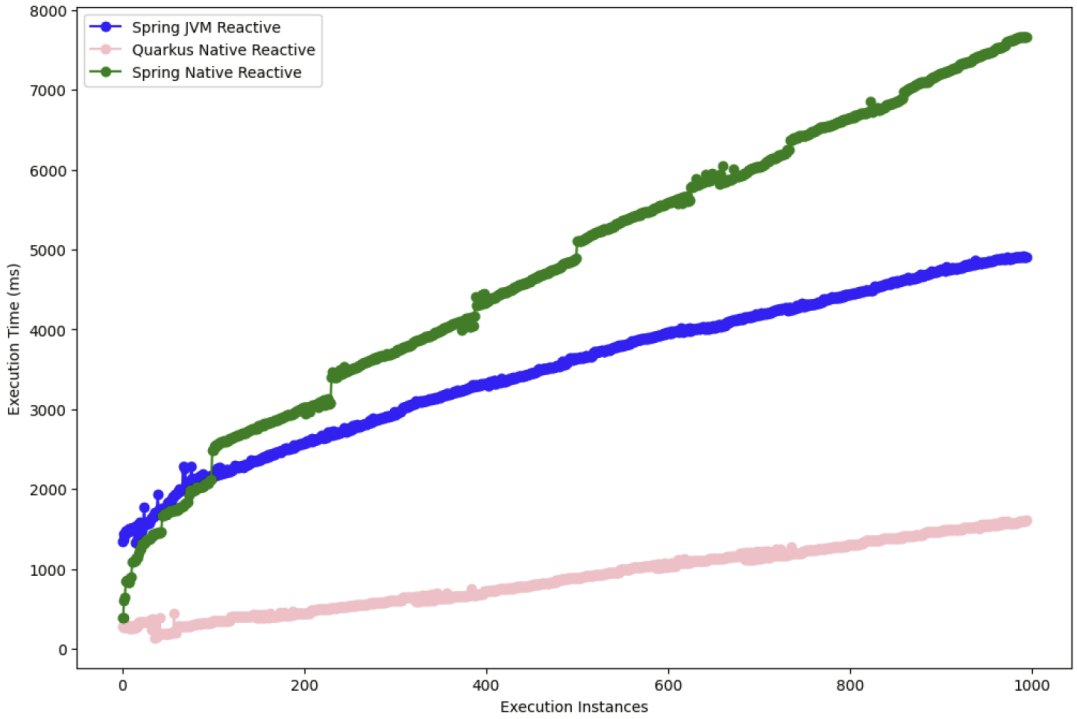

Figura 3.4: Evoluzione dei tempi di esecuzione con un carico di 1000 richieste (Reactive)

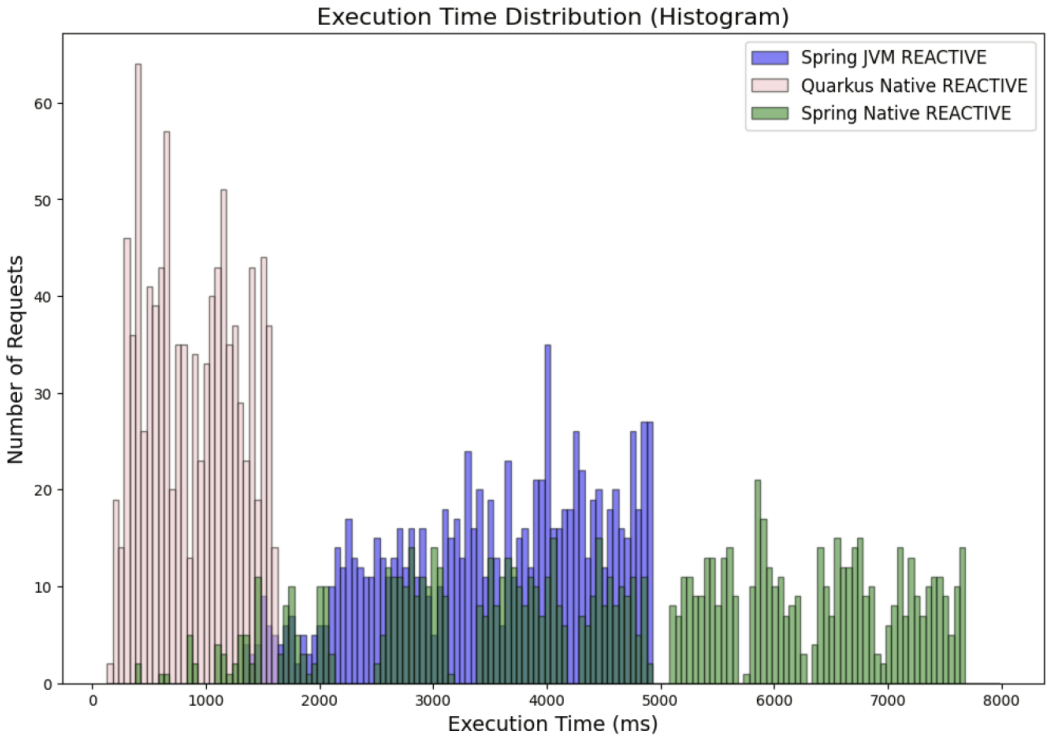

Figura 3.5: Distribuzione dei tempi di esecuzione con un carico di 1000 richieste (Reactive)

I risultati ottenuti dimostrano l'efficacia delle applicazioni native nel contesto cloud, portando le prestazioni ad un altro livello. La compilazione native fanno registrare entrambe performance eccellenti se comparate con lo standard JVM. Qualora si decidesse di optare per la soluzione con Java Virtual Machine, Quarkus risulta leggermente più efficace di SpringBoot su tempo di avviamento, consumo di memoria e tempi di risposta per un numero di richieste ragionevole. Quando si aumenta il numero di richieste per unità di tempo, SpringBoot riesce ad avere la meglio su Quarkus. Se invece volessimo comparare le versioni *reactive*, Quarkus si piazza al primo posto con un distacco netto su SpringBoot.

Nella tabella 3.2 si può osservare come un framework appositamente pensato per applicazioni in cloud possa apportare notevoli miglioramenti in termini di tempi di risposta, consumo di risorse, dimensione dell'immagine e tempi di avvio. Tutto ciò permette di minimizzare i costi verso il cloud provider e fornire un'esperienza migliore all'utente.

3.3 Ottimizzazione del Sistema Operativo

Quando si configura un server per l'esecuzione di applicazioni web, una delle prime decisioni da prendere riguarda il sistema operativo da utilizzare. Una scelta comune, soprattutto tra utenti e sviluppatori meno esperti, ricade su distribuzioni general-purpose come Ubuntu Desktop, per via della sua popolarità, semplicità d'uso e della vasta documentazione disponibile.

Tuttavia, nei contesti produttivi e professionali, è pratica diffusa optare per versioni più essenziali del sistema operativo, come Ubuntu Server. Queste varianti minimali sono prive di interfaccia grafica e di componenti non strettamente necessari, riducendo così il consumo di risorse e aumentando il livello di sicurezza del sistema. Questa tendenza riflette l'evoluzione dei requisiti dei moderni ambienti cloud-native, in cui efficienza, scalabilità e sicurezza assumono un ruolo centrale nella progettazione dell'infrastruttura.

3.3.1 Talos Linux

In questo scenario si inseriscono anche soluzioni più specializzate e moderne come Talos Linux, un sistema operativo progettato appositamente per l'esecuzione di cluster Kubernetes, che si distingue per un approccio radicalmente minimalista, sicuro e immutabile. [10]

Le sue principali caratteristiche includono:

- **Minimalismo**: Talos esclude strumenti non essenziali, riducendo la superficie di attacco e migliorando la sicurezza, lo spazio occupato e il consumo di risorse.
- **Sicurezza**: Il sistema è progettato per essere sicuro per impostazione predefinita, con funzionalità come filesystem in sola lettura e autenticazione mTLS per l'accesso API.
- **Immutabilità**: Le configurazioni del sistema sono gestite tramite file YAML, e gli aggiornamenti vengono applicati in modo atomico, garantendo coerenza e riducendo i rischi di errori manuali. Inoltre il filesystem in sola lettura, impedisce modifiche non autorizzate e previene l'istallazione di software malevoli da parte degli attaccanti.

Nelle Tabelle 3.6 e 3.7 è possibile osservare l'efficienza complessiva di Talos Linux rispetto a una distribuzione più tradizionale come Ubuntu Server. In particolare, Talos si distingue per la sua estrema leggerezza in termini di spazio occupato su disco, come evidenziato dalla significativa differenza nella dimensione dell'immagine ISO.

Oltretutto, Talos Linux, essendo pensato per il cloud e Kubernetes, semplifica notevolmente l'installazione di un cluster Kubernetes. Infatti, in Ubuntu, per

Sistema Operativo	**Dimensione ISO**
Talos Linux	∼90 MB
Ubuntu Server	∼1.5 GB
Ubuntu Desktop	∼4.5 GB

Tabella 3.6: Confronto ISO: Talos Linux, Ubuntu Server ed Ubuntu Desktop

configurare un cluster da zero, sono necessari circa trentasei comandi. Per ottenere lo stesso risultato, in Talos, ne bastano solamente tredici.

Inoltre, i dati riportati nella Tabella 3.7 mostrano come l'utilizzo delle risorse (RAM, CPU e spazio su disco) da parte di un cluster Kubernetes eseguito su nodi Talos sia sensibilmente inferiore rispetto ai medesimi nodi configurati con Ubuntu Server. Questo risultato evidenzia l'efficacia dell'approccio minimalista e ottimizzato di Talos Linux, rendendolo particolarmente adatto a scenari cloud-native che richiedono efficienza, sicurezza e ridotta impronta del sistema operativo.

Sistema Operativo	**Utilizzo RAM**	**Utilizzo Utilizzo CPU**	**Spazio su disco**
Talos Linux (Control Node)	∼2300 MB	∼ 5%	∼ 5.4GB
Talos Linux (Worker Node)	∼1600 MB	∼ 1.8%	∼3.5 GB
Ubuntu Server (Control Node)	∼3100 MB	∼ 9%	∼11.5 GB
Ubuntu Server (Worker Node)	∼2100 MB	∼ 2%	∼11.5 GB

Tabella 3.7: Confronto tra Talos Linux e Ubuntu Server nell'installazione di un cluster Kubernetes

Capitolo 4

Sicurezza

Lo sviluppo di applicazioni web in ambiente cloud introduce nuove sfide e criticità legate alla sicurezza, che risultano spesso più complesse rispetto a quelle tipiche degli ambienti tradizionali on-premise. In particolare, l'utilizzo di architetture distribuite basate su microservizi, la dinamicità dei container e l'esposizione continua ad Internet dei servizi in cloud, rendono il perimetro di attacco più esteso e meno definito.

Nel contesto di un'infrastruttura moderna, come quella orchestrata da Kubernetes, le superfici di attacco si moltiplicano: ogni nodo, pod, volume montato, configurazione errata o permesso eccessivo può rappresentare un potenziale punto di compromissione. A ciò si aggiunge la necessità di garantire la sicurezza sia contro minacce esterne, come attacchi DDoS, exploit di vulnerabilità nei container o accessi non autorizzati al cluster, sia contro minacce interne, che possono includere errori umani, configurazioni troppo permissive o comportamenti malevoli da parte di utenti autenticati.

Per affrontare tali problematiche, è fondamentale adottare un approccio multilivello alla sicurezza, che preveda strumenti e politiche specifiche per il monitoraggio, la prevenzione e la mitigazione delle minacce. In questa direzione, verranno analizzate due soluzioni complementari:

- **Falco**: un motore di rilevamento delle intrusioni runtime che consente il monitoraggio continuo dei comportamenti anomali all'interno del cluster Kubernetes. Grazie all'utilizzo di regole personalizzabili, Falco è in grado di identificare eventi sospetti come l'apertura di shell in container, l'accesso non autorizzato a file sensibili o l'esecuzione di comandi anomali, generando alert in tempo reale. Ciò consente di migliorare significativamente la visibilità sulle attività all'interno del cluster e di rispondere prontamente a comportamenti potenzialmente dannosi. [11]

- **Accessi controllati mediante kubeconfig e RBAC (Role-Based Access Control)**: mentre Falco si concentra sul monitoraggio dei comportamenti a runtime, la sicurezza deve essere garantita anche a livello di autorizzazioni. In questo contesto, l'impiego di kubeconfig con privilegi limitati e il ricorso a RBAC rappresentano pratiche fondamentali per ridurre il rischio di azioni malevole o involontarie da parte di utenti interni. Mediante RBAC è possibile definire con precisione i ruoli, le politiche e i permessi associati a ciascun utente o servizio, delimitandone l'ambito operativo, ad esempio limitando l'accesso ai soli namespace di competenza o impedendo operazioni critiche sui componenti del cluster.

L'approccio congiunto di monitoraggio attivo e controllo degli accessi consente di rafforzare la sicurezza dell'intero ambiente cloud-native. In questo capitolo verranno dunque approfondite le vulnerabilità più comuni, presentati casi d'uso pratici e illustrate le configurazioni necessarie per integrare Falco e RBAC all'interno di un cluster Kubernetes, con l'obiettivo di costruire un'infrastruttura resiliente, osservabile e sicura.

4.1 Minacce esterne e monitoraggio del cluster con Falco

Nel contesto di un'infrastruttura cloud-native, le minacce provenienti dall'esterno rappresentano una delle principali sfide da affrontare in termini di sicurezza. L'esposizione dei servizi su Internet, la comunicazione tra microservizi, l'impiego di container dinamici e spesso effimeri, nonché l'elevata complessità architetturale, amplificano il rischio di attacchi diretti ai nodi del cluster Kubernetes. Tra le minacce più comuni si annoverano: accessi non autorizzati via SSH, scansioni di rete, accessi a file sensibili, *injection* di moduli del kernel, ed esecuzione di processi non standard all'interno dei container.

Per rispondere efficacemente a queste criticità, è fondamentale dotarsi di strumenti capaci di rilevare comportamenti anomali in tempo reale, intervenendo tempestivamente prima che tali eventi possano compromettere l'integrità del sistema. In questa direzione si colloca Falco, un motore di sicurezza sviluppato dalla Linux Foundation, progettato specificamente per ambienti cloud e containerizzati.

Falco monitora gli eventi del kernel Linux e li correla a regole personalizzabili per generare alert in caso di comportamenti sospetti. La sua capacità di rilevare eventi a basso livello, come chiamate di sistema (*syscalls*), accessi a file protetti o tentativi di esecuzione non previsti, lo rende particolarmente adatto al monitoraggio proattivo di cluster Kubernetes in produzione. Alcune delle regole di sicurezza predefinite includono il rilevamento di:

- Connessioni SSH in ingresso verso i nodi del cluster;
- Accessi a file critici del sistema come `/etc/shadow` o `/etc/passwd`;
- Tentativi di ricerca di chiavi private o password;
- Operazioni di pulizia dei log, indicative di un tentativo di occultamento di attività malevole;
- Apertura di shell all'interno di un container in esecuzione;
- Utilizzo di socket per il *privilege escalation*.

È possibile, inoltre, definire delle regole custom per il monitoraggio e la generazione di alert per eventi specifici.

Un ulteriore punto di forza di Falco è la sua leggerezza: in condizioni di inattività consuma in media circa ~0.05 CPU e 100 MB di RAM, rendendolo adatto anche a cluster di dimensioni contenute.

4.1.1 Monitoraggio avanzato degli accessi SSH

Una delle limitazioni intrinseche di Falco è legata alla natura degli eventi monitorati: in quanto sistema basato sull'osservazione delle chiamate di sistema, Falco è in grado di rilevare una connessione sulla porta 22 (SSH), ma non può distinguere tra tentativi di accesso riusciti e falliti, né interpretare il contenuto dei file di log come `/var/log/auth.log`.

Per colmare questa lacuna, è possibile integrare Falco con un meccanismo di log monitoring sviluppato ad hoc. In particolare, è stato realizzato uno script Python che opera come *Daemonset* nel cluster, in grado di effettuare il *watching* continuo del file `auth.log` su ciascun nodo, identificando pattern specifici come `"Failed password"` o `"Accepted password"`. Al rilevamento di tali pattern, lo script invia un alert tramite webhook, ad esempio verso un canale Discord configurato per ricevere notifiche di sicurezza.

Questa architettura ibrida permette di unire i vantaggi dell'analisi basata su syscall (Falco) con la granularità offerta dal monitoraggio semantico dei log di sistema, ottenendo così una copertura di sicurezza più completa e informata.

Integrazione e deployment: Falco può essere installato in un cluster Kubernetes utilizzando Helm, tramite il seguente comando:

```
1 helm install --namespace falco --create-namespace
2 falco falcosecurity/falco
3 --set tty=true -f values.yaml -f falco-custom-rules.yaml
```

Il file `falco-custom-rules.yaml` permette di definire regole personalizzate persistenti, come ad esempio:

```
rule: SSH Access Detected
desc: Detect SSH connections to the server and log the source IP
condition: (evt.type = accept or evt.type=connect) and fd.sport=22
output: "SSH try to access detected from %fd.cip to port %fd.sport
    user=%user.name pid=%proc.pid"
priority: WARNING
tags: [ssh, security]
```

In combinazione con lo script di log monitoring per SSH, eseguito su ciascun nodo attraverso un `DaemonSet`, è possibile ottenere notifiche dettagliate relative a tutti gli eventi di accesso SSH, comprensive dell'esito (successo o fallimento), migliorando significativamente la capacità di rilevamento e reazione rispetto ad accessi non autorizzati.

4.2 Minacce interne e controllo degli accessi con RBAC e Keycloak

Nel contesto della sicurezza in ambienti cloud-native, non si possono trascurare le minacce interne, ovvero tutti quei rischi che provengono da soggetti o processi già legittimamente presenti nel sistema. Tali minacce includono errori umani, configurazioni errate, utilizzo improprio di privilegi elevati, ma anche comportamenti malevoli da parte di utenti interni.

Kubernetes, in quanto orchestratore di container distribuiti, offre un meccanismo integrato di controllo degli accessi denominato **Role-Based Access Control (RBAC)**, il quale consente di definire con precisione le operazioni che un determinato utente o gruppo può eseguire su specifiche risorse del cluster. La corretta configurazione di RBAC rappresenta una delle difese più efficaci contro l'escalation accidentale o malevola dei privilegi.

4.2.1 Gestione classica di RBAC

In una configurazione base, RBAC può essere implementato direttamente definendo ruoli e vincoli all'interno del cluster. Tuttavia, tale approccio presenta dei limiti quando si ha la necessità di scalare il numero di utenti, centralizzare l'autenticazione o integrare il cluster con un sistema di Identity Provider (IdP) esterno. In particolare, la gestione manuale di kubeconfig, certificati e ruoli associati a ciascun utente diventa rapidamente complessa e soggetta a errori.

4.2.2 Soluzione avanzata: RBAC integrato con Keycloak

Per ovviare a queste limitazioni, è stato implementato, nell'ambito di questo lavoro, un sistema completo per la gestione degli accessi RBAC tramite **Keycloak**, una piattaforma open-source di gestione delle identità ed autenticazione conforme agli standard OAuth 2.0 ed OpenID Connect (OIDC). L'integrazione consente di delegare completamente la gestione dell'identità a Keycloak, mantenendo Kubernetes focalizzato sulla gestione delle autorizzazioni.

L'architettura realizzata prevede due componenti principali:

- **Server-side: configurazione del cluster e mapping dei ruoli Keycloak con RBAC Kubernetes**. A tale scopo sono stati sviluppati due script:
  - *cluster-init.sh*: configura il kube-apiserver per accettare token OIDC da Keycloak [1] ed associa il ruolo cluster-admin a tutti gli utenti che appartengono al gruppo Keycloak admin, per fronire l'accesso completo agli amministratori del sistema. Inoltre, aggiunge i parametri –oidc-issuer-url, –oidc-client-id, –oidc-username-claim e –oidc-groups-claim al server Kubernetes per abilitare l'autenticazione federata.
  - *cluster-create-role-for-namespace.sh*: consente di creare dinamicamente ruoli limitati per specifici namespace e associare l'accesso a gruppi Keycloak dedicati. In questo modo è possibile implementare una segmentazione dei permessi, garantendo a ogni utente solo l'accesso al proprio spazio di lavoro.
- **Client-side: generazione automatica dei kubeconfig e gestione dell'autenticazione OIDC**. È stato implementato uno script client, *client-fetch-config.sh*, che genera automaticamente un kubeconfig contenente i parametri di autenticazione per Keycloak, utilizzando il tool *kubelogin*. L'autenticazione avviene in modalità interattiva al primo utilizzo del comando `kubectl`, e il token OIDC ottenuto include le informazioni sul ruolo dell'utente, che verrà poi interpretato da Kubernetes per l'autorizzazione.

È stato inoltre previsto uno script di logout, `client-logout.sh`, che consente all'utente di revocare il token corrente e accedere con un'identità differente. Questo facilita la gestione di ruoli multipli e migliora la sicurezza nel caso in cui sia necessario invalidare rapidamente una sessione.

4.2.3 Differenze tra approccio diretto e federato

Il lavoro sviluppato confronta due modelli di gestione:

1. **RBAC locale**: ruoli e autorizzazioni sono definiti interamente dentro Kubernetes. Ogni utente dispone di un kubeconfig statico, spesso generato manualmente. Questo approccio è semplice ma poco scalabile e difficile da integrare con policy aziendali preesistenti.
2. **RBAC federato con Keycloak**: l'autenticazione è centralizzata, le identità sono gestite da Keycloak, e Kubernetes interpreta le informazioni OIDC contenute nei token per applicare le regole RBAC. Questo approccio è più complesso da configurare inizialmente, ma altamente scalabile, sicuro e conforme agli standard moderni di enterprise IAM.

Oltretutto, l'approccio realizzato presenta diversi vantaggi:

- Maggiore sicurezza: le credenziali degli utenti non sono mai memorizzate localmente e l'accesso è revocabile centralmente.
- Scalabilità: è possibile gestire decine o centinaia di utenti tramite i gruppi e i ruoli di Keycloak.
- Tracciabilità: i token OIDC permettono di identificare ogni utente in maniera univoca.
- Flessibilità: l'accesso può essere limitato a uno o più namespace, in base al gruppo Keycloak dell'utente.

Durante la fase di sviluppo si è affrontata anche una problematica relativa alla corrispondenza tra i ruoli definiti in Keycloak e i gruppi attesi da Kubernetes nel token OIDC. Per garantire la compatibilità, è stato configurato un *Client Scope* personalizzato all'interno di Keycloak, che mappa i ruoli dell'utente al campo `groups` nel token. Questo mapping è stato abilitato nel client kubernetes del realm dedicato, e permette a Kubernetes di interpretare correttamente le autorizzazioni senza necessità di ulteriori adattamenti.

Capitolo 5

Personalizzazione dell'ambiente Cloud

5.1 La personalizzazione nel contesto Cloud

Nel contesto dell'ingegneria delle applicazioni cloud-native, la personalizzazione dell'ambiente di esecuzione riveste un ruolo sempre più centrale. Sebbene le piattaforme cloud e gli orchestratori come Kubernetes offrano funzionalità avanzate e componenti predefiniti per la gestione delle applicazioni, in molti casi è necessario estendere o adattare tali strumenti per soddisfare requisiti specifici di dominio, governance, scalabilità o integrazione.

Questa esigenza si manifesta con particolare forza nel caso delle architetture a microservizi, in cui la complessità degli ecosistemi applicativi richiede spesso soluzioni personalizzate per il bilanciamento del carico, l'autenticazione centralizzata, la gestione dei permessi o la configurazione dinamica delle rotte di accesso.

5.2 Ingress Controller ed API Gateway

Uno dei componenti fondamentali in un cluster Kubernetes è l'**Ingress Controller**, il cui scopo è quello di gestire l'esposizione delle applicazioni interne verso l'esterno del cluster. L'Ingress è una risorsa Kubernetes di alto livello che permette di definire, tramite un oggetto dichiarativo, le regole di routing HTTP(S) per uno o più servizi. Tuttavia, l'Ingress di per sé è solo una specifica, e richiede un controller sottostante che ne interpreti le direttive e configuri opportunamente un reverse proxy (come NGINX, Traefik, HAProxy o Envoy).

Parallelamente, in un contesto applicativo più avanzato, è frequente l'adozione di un **API Gateway**, il cui scopo è quello di fornire un punto di ingresso unico

per un insieme di microservizi. Tra le sue funzionalità vi sono: il routing dinamico, la gestione centralizzata dell'autenticazione, la trasformazione delle richieste, il rate-limiting e l'osservabilità.

Uno dei gateway più utilizzati in ambito Spring è **Spring Cloud Gateway (SCG)**, una soluzione reattiva, flessibile e cloud-native per la gestione del traffico HTTP. SCG è altamente configurabile tramite file YAML e può essere facilmente integrato con meccanismi di autenticazione come OAuth2.

5.3 Ingress Controller Personalizzato per Spring Cloud Gateway

Nell'ambito di questa tesi, è stato progettato e sviluppato un **Ingress Controller personalizzato** con l'obiettivo di creare un'integrazione automatica tra le risorse *Ingress* di Kubernetes e la configurazione delle *routes* dello Spring Cloud Gateway. Questo progetto risponde all'esigenza di centralizzare e automatizzare la gestione delle rotte HTTP in ambienti multi-servizio, riducendo la necessità di interventi manuali e garantendo coerenza tra le risorse dichiarative Kubernetes e la configurazione dell'API Gateway.

5.3.1 Architettura e funzionamento

Il sistema è composto da:

- Un' **applicazione Spring Cloud Gateway** containerizzata, configurata tramite un file `application.yaml` montato in una *ConfigMap*;
- Alcuni **Microservizi** sui quali reindirizzare il traffico
- Un **controller Kubernetes scritto in Go** (basato su controller-runtime e client-go), incaricato di osservare le risorse Ingress e tradurle dinamicamente in rotte per lo Spring Cloud Gateway;
- Un meccanismo di **rolling update** del deployment SCG per applicare le modifiche al ConfigMap.

5.3.2 Tecnologie e librerie utilizzate

Il controller è stato sviluppato utilizzando:

- **client-go**: per interazioni dirette e tipizzate con API Kubernetes (es. ConfigMap, Deployment);

- **controller-runtime**: per la logica di riconciliamento e gestione delle risorse osservate;
- **yaml.v3**: per il parsing e la serializzazione del file `application.yaml`;
- **Docker**: per la containerizzazione del controller in un'immagine minimale e sicura.

5.3.3 Client-go, Controller-runtime e meccanismi interni

Client-go è la libreria ufficiale, low-level, per l'interazione con l'API Server di Kubernetes in linguaggio Go. Essa espone interfacce dirette per effettuare operazioni CRUD (Create, Read, Update, Delete) su qualsiasi risorsa Kubernetes, ma non fornisce astrazioni per la scrittura di controller. Pertanto, lo sviluppatore è tenuto a gestire manualmente:

- il ciclo di vita del controller,
- la registrazione di watchers (List-Watch),
- la sincronizzazione della cache,
- il ciclo di riconciliazione e la gestione delle code di lavoro.

Controller-runtime è una libreria di più alto livello costruita sopra `client-go`, nata come parte integrante del progetto Kubebuilder. [12] Fornisce un insieme di astrazioni che semplificano drasticamente la realizzazione di controller personalizzati. Tra i componenti chiave vi sono:

- **Manager**: inizializza cache, client, informer e metriche;
- **Reconciler**: implementa la logica applicativa che risponde ai cambiamenti;
- **Controller**: osserva risorse e genera eventi di riconciliazione;
- **SetupWithManager**: collega le risorse da osservare al reconciler.

Grazie a queste astrazioni, lo sviluppatore è esentato dalla gestione manuale del ciclo di polling e delle code di lavoro, potendo concentrarsi sulla sola logica applicativa da eseguire alla comparsa di eventi (create, update, delete) sulle risorse osservate.

Nello specifico, la libreria controller-runtime è stata utilizzata per la gestione della logica di *watch* e riconciliamento sulle risorse di tipo `Ingress`, mentre client-go è stata impiegata per operazioni avanzate su oggetti come `ConfigMap` e `Deployment`, per le quali è stato necessario avere un controllo più esplicito e tipizzato.

Un ulteriore aspetto, riguarda la portabilità dell'applicazione: il codice del controller è progettato per essere eseguibile sia in ambiente **locale** che all'interno del **cluster Kubernetes**. Questo è possibile grazie al seguente frammento di codice:

```
home, _ := os.UserHomeDir()
        defaultKubeconfig := home + "/.kube/config"
        kubeconfig := os.Getenv("KUBECONFIG")
        if kubeconfig == "" {
                kubeconfig = defaultKubeconfig
        }
if _, err := os.Stat(kubeconfig); err == nil {
fmt.Println("Using kubeconfig from:", kubeconfig)
config, err = clientcmd.BuildConfigFromFlags("", kubeconfig)
} else {
fmt.Println("Using in-cluster configuration...")
config, err = rest.InClusterConfig()
}
```

Grazie a questa logica condizionale, se il file `kubeconfig` è presente nel path locale (es. `~/.kube/config`), il controller verrà eseguito con configurazione esterna usando il kubeconfig presente sul dispositivo. Se il file non è disponibile, viene automaticamente utilizzata la configurazione in-cluster, utile per ambienti di produzione o deployment su Kubernetes. Questa strategia aumenta la flessibilità e facilita lo sviluppo, il testing e la distribuzione del controller.

5.3.4 Flusso interno di funzionamento: interazione tra client-go, Informer e Controller

Per comprendere nel dettaglio il funzionamento di un controller Kubernetes personalizzato, è utile analizzare il flusso di eventi che parte dall'API Server di Kubernetes fino ad arrivare al metodo `Reconcile()` del nostro controller. La Figura 5.1 illustra in maniera schematica tale processo. [13]

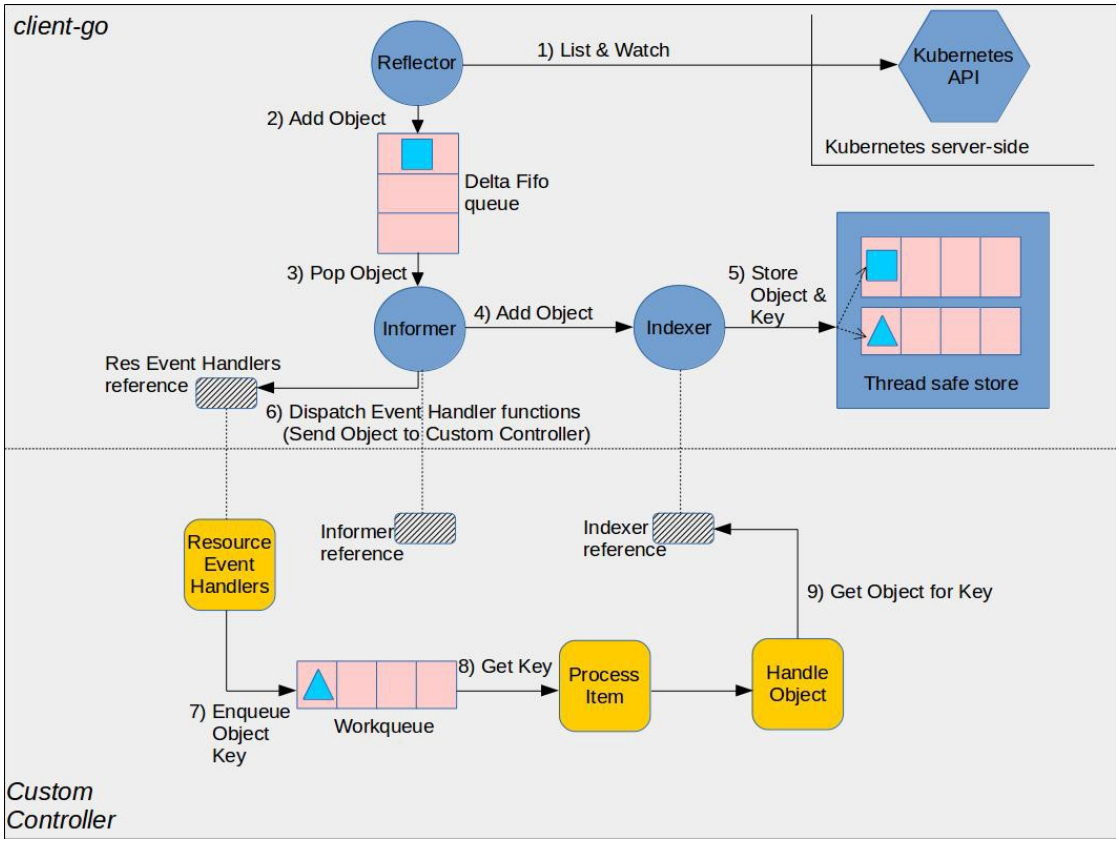

Figura 5.1: Architettura e funzionamento di un controller

Il flusso è così strutturato:

1. Il componente **Reflector** effettua un'operazione di `List` iniziale seguita da un `Watch` continuo sull'API server di Kubernetes per una determinata risorsa.

2. Ogni evento (creazione, modifica o eliminazione) viene inserito in una **Delta FIFO Queue**.

3. L'**Informer** preleva questi oggetti dalla coda e li processa.

4. Gli oggetti vengono aggiunti all'**Indexer**, che è responsabile della memorizzazione locale e sicura (cache thread-safe).

5. L'oggetto e la sua chiave vengono memorizzati nella cache condivisa, accessibile in lettura tramite Lister.

6. L'**Event Handler** associato all'informer inoltra l'evento al controller personalizzato.

7. Il controller accoda la chiave dell'oggetto (namespace/name) nella propria **Workqueue**.
8. Il ciclo di lavoro del controller preleva la chiave dalla coda.
9. Infine, la chiave viene utilizzata per recuperare l'oggetto dalla cache locale tramite l'indice, e viene chiamata la logica di riconciliazione (es. `Reconcile()`).

Questo approccio offre importanti vantaggi:

- **Efficienza**: l'uso della cache evita accessi ripetuti all'API Server;
- **Reattività**: gli eventi vengono gestiti non appena disponibili, riducendo la latenza;
- **Resilienza**: la workqueue consente un'elaborazione controllata, con eventuale *requeue* in caso di errore;
- **Modularità**: la separazione tra osservazione (Informer) e azione (Controller) consente una progettazione pulita e testabile.

In sintesi, il modello Informer-Controller in Kubernetes rappresenta un'architettura scalabile, performante ed altamente affidabile per implementare logiche reattive su risorse distribuite all'interno del cluster.

5.3.5 Reconcile e Manager

Il metodo `Reconcile(ctx, req)` implementa l'interfaccia `Reconciler`, prevista da controller-runtime:

```
1 type Reconciler interface {
2 Reconcile(context.Context, Request) (Result, error)
3 }
```

Il reconciler viene automaticamente invocato dal manager ogni volta che il sistema riceve un evento su una risorsa osservata. Il controller viene inizializzato con:

```
1 c.Watch(source.Kind(mgr.GetCache(), &networkingv1.Ingress{}), &
    handler.EnqueueRequestForObject{})
```

Questo comporta:

- La creazione di un informer per il tipo `Ingress`;

- La sincronizzazione della cache;
- La generazione di eventi di riconciliazione per ogni modifica.

```
mgr, err := manager.New(config, manager.Options{})
mgr.Start(context.Background());
```

Il **Manager** è l'orchestratore centrale di controller-runtime. Esso:

- Inizializza cache, informer e client;
- Registra i controller e i reconciler;
- Gestisce la concorrenza dei loop di riconciliazione;
- Offre meccanismi nativi per *leader election*, health check e metriche.

5.3.6 Client, Informer e Cache

Un **Informer** è un componente, creato e gestito automaticamenta dal Manager, che osserva risorse Kubernetes tramite la combinazione 'List and Watch', costruendo e mantenendo una cache locale coerente. Questo meccanismo è fondamentale per:

- Minimizzare il carico sull'API server;
- Fornire accessi in lettura più rapidi;
- Innescare notifiche efficienti agli handler come `Reconcile()`.

Ogni informatore utilizza un meccanismo di *long-polling* per ricevere eventi di tipo `ADDED`, `MODIFIED`, `DELETED`, che vengono poi processati dal reconciler.

Nel codice, entrambi i client (client-go e controller-runtime) sono costruiti a partire dalla medesima configurazione caricata da kubeconfig:

```
if _, err := os.Stat(kubeconfig); err == nil {
                fmt.Println("Using kubeconfig from:", kubeconfig)
                config, err = clientcmd.BuildConfigFromFlags("",
    kubeconfig)
        } else {
                fmt.Println("Using in-cluster configuration...")
                config, err = rest.InClusterConfig()
        }
k8sClient, _ := kubernetes.NewForConfig(config)
mgr, _ := manager.New(config, manager.Options{})
watcher := &IngressWatcher{K8sClient: k8sClient, Client: mgr.
    GetClient()}
```

5.3.7 Meccanismo di riconciliazione

Il cuore del controller è il metodo `Reconcile()`, attivato automaticamente da controller-runtime ogni volta che una risorsa Ingress viene creata, aggiornata o eliminata. Questo metodo esegue le seguenti operazioni:

1. In caso di **creazione o aggiornamento**:
   - Ottiene il contenuto YAML attuale dal ConfigMap;
   - Rimuove eventuali rotte preesistenti associate all'Ingress applicato (identificate da un prefisso coerente);
   - Costruisce le nuove rotte Spring Cloud Gateway in base alle regole specificate nell'Ingress applicato;
   - Aggiunge filtri (es. TokenRelay, RewritePath) ove richiesto tramite annotazioni (es. `stripPrefix`);
   - Serializza il nuovo YAML e aggiorna la ConfigMap;
   - Applica un rolling restart al deployment SCG aggiornando l'annotazione `restartedAt` per applicare la nuova configurazione.
2. In caso di **eliminazione**:
   - Rimuove le rotte corrispondenti all'Ingress dalla ConfigMap;
   - Riavvia il deployment SCG per ricaricare la nuova configurazione.

Funzionamento pratico del ciclo di riconciliazione (Reconcile): Ogni volta che viene creato, aggiornato o eliminato un oggetto `Ingress`, il framework `controller-runtime` attiva automaticamente il metodo `Reconcile()` del controller. Questo comportamento è garantito dalla registrazione del tipo `Ingress` all'interno della funzione `SetupWithManager()`.

```
1 func (r *IngressWatcher) SetupWithManager(mgr manager.Manager)
    error {
2         c, err := controller.New("ingress-watcher", mgr,
    controller.Options{Reconciler: r})
3         if err != nil {
4                 return err
5         }
6
7         return c.Watch(source.Kind(mgr.GetCache(), &networkingv1.
    Ingress{}), &handler.EnqueueRequestForObject{})
8 }
```

Il ciclo di vita di una riconciliazione è il seguente:

1. **Evento**: viene generato un evento (create, update o delete) su una risorsa `Ingress`;
2. **Reconcile Trigger**: `controller-runtime` intercetta l'evento e invoca `Reconcile()`;
3. **Pulizia ConfigMap**: se l'Ingress esiste, il controller rimuove tutte le route precedentemente associate a quell'Ingress (tramite un filtro sull'id);
4. **Generazione nuova configurazione**: il controller analizza la nuova configurazione dell'Ingress (path, backend, namespace) e costruisce nuove route compatibili con lo Spring Cloud Gateway;
5. **Scrittura**: la ConfigMap viene aggiornata con la nuova configurazione YAML serializzata;
6. **Restart Deployment**: viene attivato un rolling restart del pod dello Spring Cloud Gateway (SCG) aggiornando un'annotazione sul PodTemplate.

Questo approccio garantisce **idempotenza**: ogni riconciliazione genera una configurazione coerente e priva di duplicazioni. Anche se lo stesso evento si ripete o il controller si riavvia, l'applicazione resterà nello stato desiderato.

Inoltre, l'implementazione del controller sfrutta informers e cache condivise per minimizzare il carico sull'API Server e migliorare le prestazioni. Grazie a ciò, ogni evento può essere processato rapidamente senza necessità di interrogare ripetutamente Kubernetes.

Esempio pratico del comportamento di Reconcile:

```
routes:

-id: old-ingress-otherservice
uri: http://other-service.default.svc.cluster.local
predicates:
    - Path=/old

-id: my-ingress-hello-service
uri: http://hello-service.default.svc.cluster.local
predicates:
    - Path=/hello
```

Listing 5.1: Esempio di ConfigMap iniziale

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
name: my-ingress // id di riferimento per le rotte
```

```
namespace: default
spec:
rules:
    - host: my.example.com
http:
paths:
    - path: /hello
pathType: Prefix
backend:
service:
name: hello-service
port:
number: 80
    - path: /bye
pathType: Prefix
backend:
service:
name: bye-service
port:
number: 80
```

Listing 5.2: Ingress applicato

```
routes:

-id: old-ingress-otherservice
uri: http://other-service.default.svc.cluster.local
predicates:
    - Path=/old
```

Listing 5.3: Stato della ConfigMap dopo la rimozione delle route obsolete 'my-ingress-'

```
routes:

- id: old-ingress-otherservice
uri: http://other-service.default.svc.cluster.local
predicates:
    - Path=/old

- id: my-ingress-hello-service
uri: http://hello-service.default.svc.cluster.local
predicates:
    - Path=/hello

- id: my-ingress-bye-service
uri: http://bye-service.default.svc.cluster.local
predicates:
    - Path=/bye
```

Listing 5.4: Stato finale della ConfigMap dopo la riconciliazione

5.3.8 Strategia di aggiornamento e idempotenza

Per garantire il corretto funzionamento in scenari dinamici e concorrenti, il controller è progettato per essere completamente **idempotente**. Ogni ciclo di reconciliamento rimuove prima tutte le rotte generate da un determinato Ingress (identificate tramite `id`) e poi le rigenera sulla base della configurazione corrente. Questo approccio semplifica la gestione degli aggiornamenti e assicura che non rimangano configurazioni obsolete nel gateway.

Inoltre, la scelta di effettuare il restart del deployment SCG tramite aggiornamento dell'annotazione `kubectl.kubernetes.io/restartedAt` permette di forzare il ricaricamento della configurazione senza dover ricorrere a metodi più invasivi (es. eliminazione manuale dei pod).

5.3.9 Risultati

Questo approccio porta i seguenti vantaggi:

- Automatizzazione completa della generazione delle rotte del gateway a partire da risorse dichiarative Ingress;
- Possibilità di definire routing, filtri e riscritture tramite annotazioni o pattern standardizzati;
- Riduzione degli errori manuali nella configurazione del gateway;
- Integrazione trasparente con meccanismi di autenticazione OAuth2 e SCG TokenRelay;
- Elevata estensibilità grazie all'utilizzo di componenti modulari e API native Kubernetes.

Il controller rappresenta a tutti gli effetti un **operator**, in quanto ingloba logica di dominio (mappatura tra Ingress e Spring Gateway) e automatizza la gestione di un componente applicativo complesso. Questa distinzione tra semplice controller e operator è significativa in quanto definisce il livello di astrazione e responsabilità nell'automazione delle risorse cloud-native.

Il progetto implementato dimostra come sia possibile estendere in modo efficace e idiomatico il comportamento di Kubernetes tramite la creazione di controller personalizzati. L'utilizzo congiunto di `client-go` e `controller-runtime` consente un controllo fine sul comportamento del cluster, permettendo l'interazione sia con

risorse dichiarative sia con strutture complesse come i file di configurazione YAML. Inoltre, l'integrazione con strumenti DevOps moderni (Helm, Harbor, pipeline CI/CD) rende il sistema pienamente adattabile a contesti di produzione.

In sintesi, il custom ingress controller per Spring Cloud Gateway realizzato in questo lavoro rappresenta una soluzione concreta, scalabile e manutenibile per la gestione automatizzata delle rotte applicative in ambienti Kubernetes avanzati.

Capitolo 6

Motore SQL distibuito per le Applicazioni Web moderne

6.1 La necessità di un motore SQL

Nel panorama moderno delle applicazioni web in ambiente cloud, la necessità di elaborare grandi quantità di dati provenienti da fonti diverse è ormai un problema comune. Le organizzazioni, infatti, raccolgono volumi sempre crescenti di dati provenienti da fonti eterogenee: metriche di dispositivi, tracciamenti del comportamento degli utenti, transazioni aziendali, dati di localizzazione, log di test software e procedure di verifica dei sistemi. L'abilità di comprendere, analizzare e sfruttare tali dati è diventata un fattore critico per il successo di qualunque iniziativa, fino a rappresentare un elemento strategico per la competitività delle imprese.

Parallelamente, la varietà dei meccanismi di storage disponibili si è notevolmente ampliata: database relazionali, sistemi NoSQL, database a documenti, key-value store, archivi a oggetti e altri modelli emergenti sono oggi tutti ampiamente adottati. La complessità di tale ecosistema rende impraticabile l'adozione di un'unica tecnologia per la gestione dei dati. Affrontare questa eterogeneità può risultare estremamente complesso e disorientante. Le soluzioni tradizionali, basate sulla creazione e manutenzione di data warehouse centralizzati e dedicati, si sono rivelate nel tempo molto costose per le organizzazioni di tutto il mondo. Oltre all'impatto economico, tali architetture si sono dimostrate lente e poco agili rispetto alle necessità operative di molti utenti e pattern applicativi. La tanto auspicata alternativa dei data lake, pensata per offrire maggiore flessibilità, si è spesso trasformata in un semplice deposito disorganizzato di dati, difficili da analizzare e raramente utilizzati in modo efficace.

In questo scenario, Trino si propone come una soluzione innovativa in grado di superare i limiti delle architetture precedenti. Grazie al supporto per interrogazioni

federate su sistemi eterogenei, all'esecuzione parallela delle query e alla scalabilità orizzontale del cluster, Trino consente di sbloccare nuove opportunità nell'ambito dell'analisi distribuita dei dati.

6.2 Cos'è Trino

Trino è un motore di query SQL distribuito, open source, progettato per eseguire interrogazioni su sorgenti dati eterogenee, da poche centinaia di megabyte fino a diversi petabyte. La sua architettura consente di scalare orizzontalmente il carico computazionale attraverso l'aggiunta di nodi worker, superando così i limiti delle soluzioni verticali tradizionali.

Trino si pone come alternativa concreta tra le soluzioni di analisi ad alte prestazioni e ad alto costo presenti sul mercato e le soluzioni "gratuite" che richiedono investimenti significativi in risorse hardware per ottenere risultati simili.

Trino non è un database in senso stretto: non possiede uno storage interno e non memorizza dati. Il suo ruolo è quello di fornire un layer di accesso SQL standardizzato a sistemi già esistenti.

6.3 SQL-on-Anything

Una delle caratteristiche più distintive di Trino è il suo approccio *SQL-on-Anything*, ovvero la capacità di eseguire interrogazioni SQL direttamente sui dati, indipendentemente da dove essi risiedano, senza necessità di effettuare una migrazione preventiva verso un sistema centralizzato.

Nell'ecosistema attuale, i dati sono archiviati in una molteplicità di sistemi ottimizzati per casi d'uso specifici: archiviazione a oggetti (object storage), key-value store, database a documenti, database a grafo, sistemi di event streaming e altri sistemi classificati genericamente come NoSQL. Ciascuno di questi ambienti fornisce funzionalità uniche e vantaggi peculiari, ma richiede anche l'utilizzo di strumenti e tecnologie dedicate per la loro consultazione.

Trino supera questa frammentazione fornendo un layer unificato di accesso ai dati mediante SQL standard, secondo la specifica ANSI (American National Standards Institute). In questo modo, l'utente può collegarsi a tutte queste sorgenti come se fossero tabelle in un unico database, ed eseguire interrogazioni utilizzando strumenti già familiari e consolidati. Questo approccio rende Trino un sistema veramente *SQL-on-Anything*, in grado di interfacciarsi con qualsiasi tecnologia supportata da un connettore.

6.4 Query Federate

Come accennato, una delle funzionalità fondamentali di Trino è la capacità di eseguire interrogazioni su sorgenti dati eterogenee, anche nel caso in cui queste non utilizzino nativamente il linguaggio SQL. Tuttavia, molte delle domande che emergono durante l'analisi dei dati non possono essere soddisfatte interrogando una singola fonte. Al contrario, spesso è necessario estrarre informazioni da sistemi diversi, combinarle e analizzarle congiuntamente.

Trino consente questo tipo di operazioni tramite le **query federate**. Una query federata è un'interrogazione SQL che fa riferimento, all'interno della stessa istruzione, a schemi e tabelle appartenenti a sistemi completamente differenti. Tutte le sorgenti configurate in Trino sono accessibili contemporaneamente e possono essere utilizzate nella stessa query, con la stessa sintassi SQL.

Questo meccanismo permette, ad esempio, di unire dati utente da un database PostgreSQL con informazioni transazionali archiviate in MongoDB, senza doverli replicare o spostare. L'interrogazione avviene in modo trasparente e coerente, migliorando l'efficienza dei flussi analitici e semplificando l'integrazione di sorgenti dati eterogenee.

6.5 Concetti fondamentali

Trino si basa su una serie di concetti chiave:

- **Connettore**: modulo che adatta Trino a una specifica sorgente dati, ogni catalogo è associato ad uno specifico connettore;
- **Catalogo**: definisce i dettagli per accedere ala sorgente di dati e contiene uno o più schemi;
- **Schema**: struttura logica che organizza le tabelle all'interno di un catalogo;
- **Tabella**: insieme di righe e colonne, accessibile tramite SQL.

6.6 L'Architettura

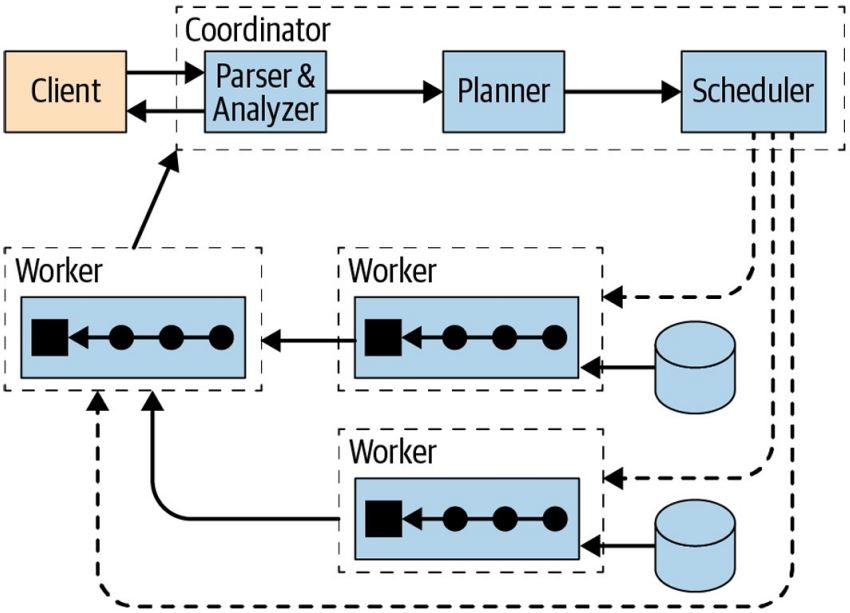

Figura 6.1: L'architettura di Trino

Trino è un motore di interrogazione SQL distribuito, ispirato al paradigma dei sistemi MPP (Massively Parallel Processing), ovvero sistemi che eseguono elaborazioni parallele su larga scala. A differenza delle soluzioni basate su scalabilità verticale — che aumentano la potenza computazionale di un singolo nodo — Trino adotta un approccio di scalabilità orizzontale, distribuendo il carico di lavoro su un cluster di server denominati *worker*.

Al centro dell'architettura di Trino si trova il **coordinatore**, un nodo speciale incaricato di ricevere le query SQL dai client, effettuare il parsing, pianificare l'esecuzione e assegnare i compiti ai nodi worker. Il coordinatore rappresenta il cervello del sistema e costituisce il punto di accesso per gli utenti, che possono interagire con esso tramite l'interfaccia a riga di comando di Trino (CLI), driver JDBC o ODBC, il client Python ufficiale oppure tramite una delle numerose librerie client disponibili per diversi linguaggi di programmazione.

Ogni installazione di Trino deve prevedere almeno un coordinatore e uno o più worker. In ambienti di sviluppo o test, è possibile configurare un'istanza singola di Trino in grado di svolgere entrambi i ruoli contemporaneamente.

Quando il coordinatore riceve una query SQL, esegue le seguenti fasi:

1. **Parsing**: analizza sintatticamente la query;

2. **Analisi semantica**: verifica la correttezza e coerenza logica;

3. **Pianificazione**: genera un piano esecutivo ottimizzato;

4. **Schedulazione**: distribuisce il piano sotto forma di task ai worker.

Il coordinatore utilizza un **servizio di discovery** per individuare i nodi presenti nel cluster. Ogni istanza di Trino, al momento dell'avvio, si registra presso tale servizio e invia periodicamente segnali di *heartbeat*, consentendo al coordinatore di mantenere una lista aggiornata dei worker disponibili. Questa lista è fondamentale per la corretta assegnazione dei compiti di esecuzione.

I **worker** di Trino sono i nodi responsabili dell'elaborazione effettiva dei dati. Ogni worker riceve i task dal coordinatore, accede ai dati attraverso i connettori configurati, li elabora, e scambia i risultati intermedi con altri worker, quando necessario. I risultati finali dell'interrogazione vengono poi restituiti al coordinatore, che si occupa dell'aggregazione e della consegna al client.

Questa architettura a componenti separati consente a Trino di eseguire query complesse in modo parallelo, garantendo elevata efficienza, scalabilità e adattabilità a diversi scenari di carico computazionale.

6.7 I connettori

Uno dei pilastri dell'architettura di Trino è la netta separazione tra computazione e storage, resa possibile dal modello a **connettori**. Ogni connettore fornisce a Trino un'interfaccia standardizzata per accedere a una specifica sorgente dati, indipendentemente dalla sua natura tecnologica.

Dal punto di vista logico, un connettore espone la sorgente dati come un insieme di tabelle, colonne e righe, aderendo al modello relazionale. Finché i dati sottostanti possono essere rappresentati attraverso strutture tabellari e tipi di dato compatibili con Trino, è possibile sviluppare un connettore ad hoc che consenta al motore di query di interagire con essi.

Trino fornisce un'interfaccia chiamata **Service Provider Interface (SPI)**, che definisce le funzionalità che ogni connettore deve implementare per abilitare specifiche operazioni. L'implementazione dello SPI consente a Trino di operare in maniera coerente e trasparente con qualsiasi sistema esterno, delegando al connettore la gestione dei dettagli specifici della sorgente dati (es. protocolli di connessione, conversione dei tipi, gestione delle partizioni, ecc.).

Questo approccio modulare favorisce l'estensibilità del sistema: nuovi connettori possono essere aggiunti senza modificare il core di Trino, abilitando l'integrazione con un numero crescente di tecnologie.

6.8 Esecuzione delle Query

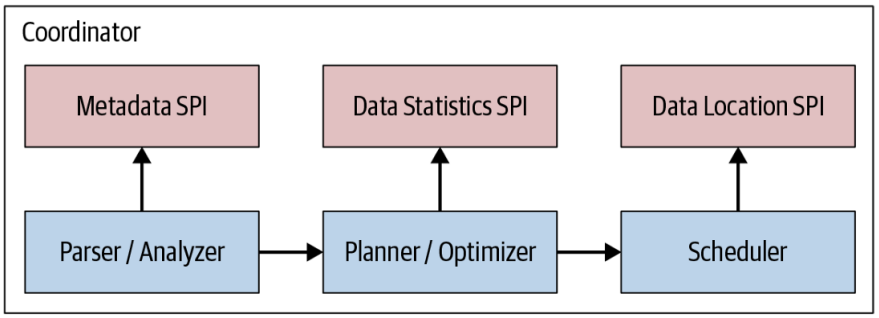

Figura 6.2: Trino: Esecuzione dell'interrogazione

Quando una query SQL viene inviata a Trino, il primo componente a riceverla è il **coordinatore**, che la riceve sotto forma di testo. Il processo di esecuzione si articola nelle seguenti fasi principali:

1. **Parsing e analisi**: la query viene analizzata sintatticamente e semanticamente. Vengono effettuati controlli sui tipi di dato, sulla validità delle espressioni, e vengono applicate le policy di sicurezza.

2. **Recupero del metadato**: Trino utilizza lo SPI per accedere ai metadati della sorgente (es. nomi delle tabelle, colonne, tipi di dato). Queste informazioni sono fondamentali per validare la query ed elaborare un piano esecutivo coerente.

3. **Ottimizzazione basata su costi (CBO)**: tramite lo **statistics SPI**, Trino recupera informazioni quantitative sulle tabelle (numero di righe, dimensione stimata) per selezionare il piano più efficiente.

4. **Creazione del piano distribuito**: attraverso lo **data location SPI**, Trino suddivide il contenuto delle tabelle in **split**, che rappresentano le unità logiche minime di lavoro parallelizzabile. Gli split vengono poi assegnati ai nodi worker per l'esecuzione distribuita della query.

L'intero piano esecutivo viene rappresentato internamente in una struttura dati denominata *query plan*, che descrive in dettaglio il flusso di operazioni da eseguire, le dipendenze tra task e l'allocazione delle risorse necessarie.

Questo modello consente a Trino di sfruttare al meglio la parallelizzazione orizzontale, migliorando le prestazioni anche su dataset di grandi dimensioni e mantenendo una forte separazione tra logica di interrogazione e specificità della sorgente dati.

6.9 Predicate Pushdown e ottimizzazione basata su costi

Uno dei meccanismi chiave di ottimizzazione adottati da Trino è il cosiddetto **predicate pushdown**, ovvero la capacità di trasferire le condizioni di filtraggio il più vicino possibile alla sorgente dati. In pratica, i predicati specificati nella clausola `WHERE` vengono inviati direttamente al sistema sottostante (se supportato dal connettore), permettendo di ridurre il volume di dati trasferiti verso il cluster Trino e di conseguenza migliorare l'efficienza computazionale dell'interrogazione. Questo approccio minimizza l'uso di CPU, memoria e banda di rete all'interno di Trino, sfruttando le capacità di elaborazione locali della sorgente dati.

Trino implementa anche una **Cost-Based Optimization** (CBO), che consente al pianificatore di scegliere il piano di esecuzione più efficiente in base a stime quantitative come numero di righe, dimensioni delle tabelle e distribuzione dei valori. Tuttavia, a differenza di un database tradizionale, Trino non gestisce direttamente i dati: pertanto, la disponibilità di statistiche è delegata al connettore specifico utilizzato per accedere alla sorgente.

Attualmente, diversi connettori supportano la raccolta di statistiche, tra cui:

- Hive, Delta Lake e Iceberg per sistemi basati su object storage;
- Alcuni database relazionali come PostgreSQL, che espongono metadati statistici a Trino.

Per acquisire tali statistiche, Trino supporta due approcci principali:

1. L'uso esplicito del comando `ANALYZE`, che esegue un'analisi diretta dei dati per raccogliere statistiche aggiornate;
2. L'abilitazione della raccolta automatica durante le operazioni di scrittura, qualora Trino venga utilizzato come writer.

Il secondo approccio appare in generale più efficiente, poiché elimina la necessità di eseguire periodicamente il comando `ANALYZE`. Tuttavia, presenta una limitazione rilevante: affinché le statistiche rimangano affidabili, è necessario che tutte le operazioni di scrittura sui dati siano effettuate esclusivamente da Trino. Se altre applicazioni o strumenti modificano direttamente la sorgente dati, le statistiche potrebbero non riflettere più lo stato reale, compromettendo la qualità dell'ottimizzazione.

Benchmark e test approfonditi hanno dimostrato che l'overhead introdotto dal processo di raccolta automatica delle statistiche durante la scrittura è trascurabile dal punto di vista delle prestazioni complessive.

6.10 Limitazioni dei connettori e uso delle viste

Sebbene Trino sia in grado di delegare parte dell'elaborazione al sistema sottostante tramite il meccanismo di **query pushdown**, ciò non è sempre possibile per tutte le tipologie di operazioni SQL. Attualmente, l'interfaccia SPI dei connettori Trino consente il pushdown esclusivamente di filtri (`WHERE`) e proiezioni di colonne (`SELECT col1, col2, ...`).

Questo limite può essere sia uno svantaggio, in termini di delega computazionale, sia un'opportunità, poiché consente a Trino di mantenere il controllo sull'elaborazione, sfruttando la propria architettura distribuita. Tuttavia, in alcuni casi può risultare utile forzare ulteriori ottimizzazioni direttamente nella sorgente dati, ad esempio in un database relazionale come PostgreSQL.

Una strategia efficace consiste nell'utilizzare **viste** (*views*) definite nel sistema sottostante. Una vista può incapsulare operazioni complesse (join, filtri, aggregazioni), ed essere esposta a Trino come una semplice tabella. In tal modo, l'elaborazione viene eseguita direttamente nella sorgente, ad esempio in PostgreSQL, e Trino si limita a interrogarla, beneficiando implicitamente di tutte le ottimizzazioni locali disponibili.

Questa tecnica rappresenta un modo pragmatico per superare i limiti strutturali dell'attuale SPI, integrando i punti di forza dell'elaborazione distribuita di Trino con le funzionalità avanzate dei motori di database tradizionali.

6.11 Implementazione in Kubernetes

Per testare l'efficacia di Trino, è stato realizzato un ambiente in Kubernetes con:

- Due sorgenti dati: PostgreSQL e MongoDB;
- Caricamento di dati sintetici tramite script Python;
- Configurazione di Trino, un coordinatore e due worker, tramite Helm chart e file di configurazione `trino-values.yaml`;
- Comparativa tra query native e query mediante Trino.
- Esecuzione ed analisi delle query federate.

Il file `trino-values.yaml` permette di configurare i connettori delle sorgenti di dati in Trino:

```
1 # trino-values.yaml
2 additionalCatalogs:
3   postgresql: |
4     connector.name=postgresql
```

```
    connection-url=jdbc:postgresql://postgres-postgresql.dbs.svc.cluster.local:5432/testdb
    connection-user=postgres
    connection-password=

  mongodb: |
    connector.name=mongodb
    mongodb.connection-url=mongodb://mongodb.dbs.svc.cluster.local:27017
```

Successivamente, attraverso uno script Python, sono stati popolati i due database, PostgreSQL e MongoDB, con i medesimi dati: 10.000 Utenti, 100 Prodotti e 100.000 Ordini.

6.11.1 Analisi sperimentale delle performance

Sono state eseguite query semplici e complesse su entrambe le sorgenti, sia in modalità nativa sia tramite Trino. La Query semplice presa in considerazione, avente cardinalità 10.000, detta anche Query "A", permette di ottenere tutti i campi di uno specifico utente filtrando per nome. La query complessa, avente cardinalità 100.000, detta invece Query "B", effettua il join tra la tabella Utenti e la tabella Ordini, mostrando tutti i campi.

```
Postgres Native
A) Basic Query: Table of 10000 rows
kubectl exec -n dbs -it pod/postgres-postgresql-0 -- psql -U postgres -d
testdb
\timing on
SELECT * FROM users WHERE name = 'Grant Pace';
Time: 3 ms
B) Query with join: Table of 100000 rows
kubectl exec -n dbs -it pod/postgres-postgresql-0 -- psql -U postgres -d
testdb
\timing on
SELECT * FROM users u JOIN orders o ON u.id = o.user_id;
Time: 2554 ms


Trino Postgres
A) Basic Query: Table of 10000 rows
kubectl exec -n trino -it $POD -- bash -c "time trino --execute \" SELECT
* FROM postgresql.public.users WHERE name='Grant Pace';\""
Time: 976ms
```

```
B) Query with join: Table of 100000 rows
kubectl exec -n trino -it $POD -- bash -c "time trino --execute \"
    SELECT
* FROM postgresql.public.users u JOIN postgresql.public.orders o
    ON
u.id = o.user_id;\""
Time: 2513ms


MongoDB Native
A) Basic Query: Table of 10000 rows
kubectl exec -n dbs -it $(kubectl get pods -n dbs -l app.
    kubernetes.io/
name=mongodb -o jsonpath="{.items[0].metadata.name}") -- mongosh
testdb
db.users.find({ name: "Grant Pace" }).explain("executionStats");
Time: 8 ms
B) Query with join: Table of 100000 rows
kubectl exec -n dbs -it $(kubectl get pods -n dbs -l app.
    kubernetes.io/
name=mongodb -o jsonpath="{.items[0].metadata.name}") -- mongosh
testdb
db.orders.aggregate([
{ $limit: 100000 },
{
$lookup: {
from: "users2",
localField: "user_id",
foreignField: "id",
as: "user_info"
}
},
{ $unwind: "$user_info" }
]).explain("executionStats");
Time: 700432 ms


Trino MongoDB

A) Basic Query: Table of 10000 rows
kubectl exec -n trino -it $POD -- bash -c "time trino --execute \"
    SELECT
* FROM mongodb.testdb.users WHERE name='Grant Pace';\""
Time: 1003ms
B) Query with join: Table of 100000 rows
kubectl exec -n trino -it $POD -- bash -c "time trino --execute \"
    SELECT
* FROM mongodb.testdb.users u JOIN mongodb.testdb.orders o ON u.id
    =
```

```
o.user_id;\""
Time: 1844ms
```

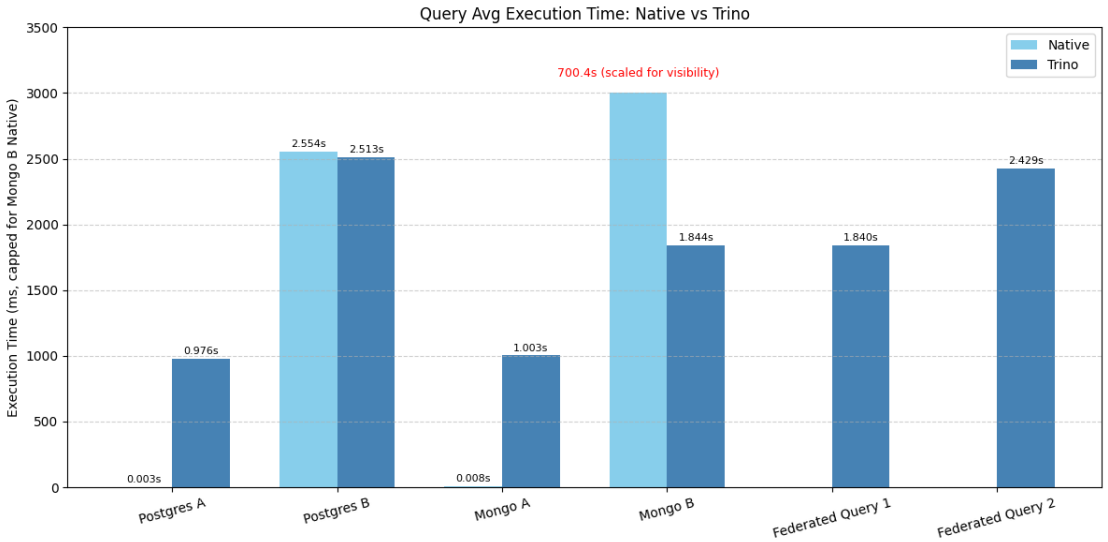

Figura 6.3: Comparativa: Trino - Nativo

Per esaminare le varie funzionalità di Trino, si è ritenuto opportuno rilevare i tempi di esecuzione delle query federate, una delle caratteristiche principali di Trino.

```
kubectl exec -n trino -it $POD -- bash -c "time trino --execute \"
SELECT *
FROM postgresql.public.users u
JOIN mongodb.testdb.orders o ON u.id = o.user_id;
\""

Time: 1840ms
```

Com'è possibile notare, l'uso di query federate non comporta un aumento significativo dei tempi di esecuzione. Essi, infatti, sono comparabili ai tempi dei join eseguiti tra tabelle della stessa fonte di dato.

Un'ulteriore analisi è stata portata avanti per valutare il comportamento del cluster di Trino sotto carico. Sono state eseguite quindi 100 query federate in sequenza per valutare sia i tempi di esecuzione che l'utilizzo delle risorse al crescere delle richieste.

```
times=()
for i in {1..100}; do
echo "Running query for product_id=$i"
duration=$(kubectl exec -n trino -i "$POD" -- bash -c "
{ time trino --execute \"
SELECT *
FROM postgresql.public.users u
JOIN mongodb.testdb.orders o ON u.id = o.user_id
WHERE o.product_id = $i;
\" > /dev/null; } 2>&1 | grep real | awk '{print \$2}'
")
times+=("$duration")
done
# Print times comma-separated
echo "${times[*]}" | tr ' ' ','
```

Nella tabella 6.4 è possibile osservare la distribuzione dei tempi di esecuzione delle 100 query federate.

I risultati hanno evidenziato che:

- Le query semplici (CRUD) sono più rapide se eseguite direttamente sulla sorgente (es. 3ms in PostgreSQL vs 976ms in Trino);
- Le query complesse o federate beneficiano significativamente di Trino (es. 700s per un join in MongoDB vs 1.8s tramite Trino);
- Il sistema gestisce bene 100 query concorrenti, mantenendo tempi medi sotto i 1.1s.

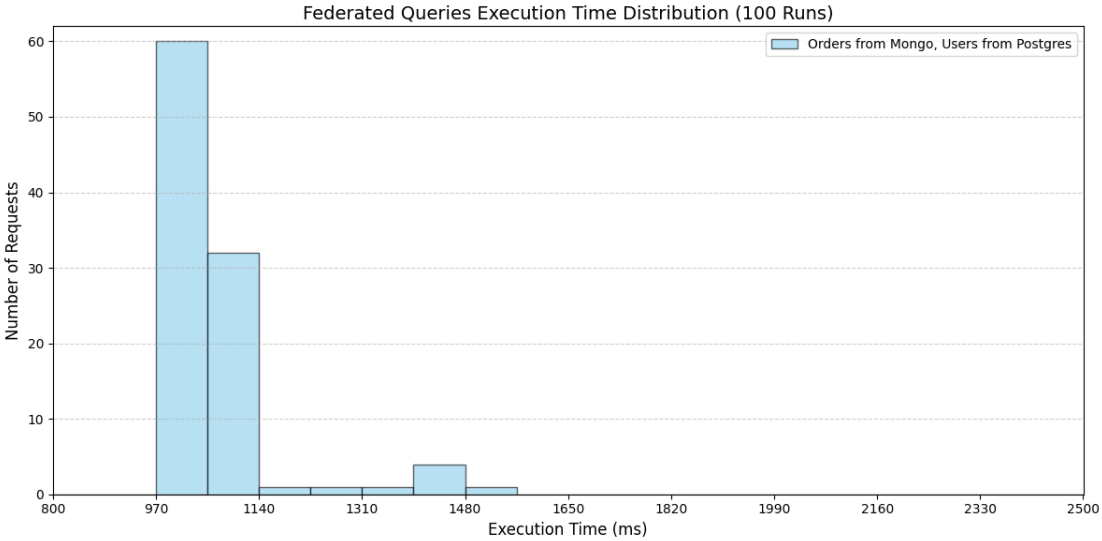

Figura 6.4: Trino: 100 richieste di query federate

6.11.2 Consumo di risorse

Durante i test sono stati rilevati i consumi del cluster Trino composto da un coordinatore e due worker:

Consumo risorse in condizioni di inattività (IDLE):

- Coordinatore: 34mCPU, 1125MiB RAM
- Worker1: 28m CPU, 1161MiB RAM
- Worker1: 73m CPU, 732MiB RAM

Consumo risorse sotto carico:

- Coordinator: 1105mCPU, 1351MiB RAM
- Worker1: 47m CPU, 2004MiB RAM
- Worker1: 146m CPU, 2025MiB RAM

Ovvero, in totale:

- In idle: ~0.135 core CPU, 2.95 GB RAM;
- Sotto carico: ~1.3 core CPU, 5.25 GB RAM.

Si osserva che il carico CPU cresce principalmente sul coordinatore (per la pianificazione delle query), mentre i worker necessitano più RAM (per l'elaborazione).

6.12 Considerazioni

Trino si è dimostrato uno strumento altamente efficiente per l'analisi distribuita su sorgenti dati eterogenee. Pur non essendo adatto a operazioni OLTP o a interrogazioni molto semplici, offre vantaggi notevoli in termini di:

- Interrogazioni federate tra sorgenti diverse;
- Uso di SQL standard;
- Scalabilità orizzontale;
- Architettura cloud-native e open source.

In contesti di data analytics avanzata e sistemi distribuiti, Trino rappresenta una soluzione moderna, efficiente e modulare per superare le limitazioni dei data warehouse tradizionali e ottenere insight unificati senza migrazione fisica dei dai.

Capitolo 7

Conclusioni

L'analisi multidimensionale condotta in questa tesi, evidenzia come lo sviluppo di Applicazioni Web moderne in ambienti cloud non possa prescindere da una profonda comprensione delle dinamiche sottostanti alle tecnologie utilizzate. È richiesto, infatti, un approccio ingegneristico avanzato, che vada ben oltre l'utilizzo convenzionale delle piattaforme cloud.

L'adozione del cloud, lungi dal rappresentare una semplice transizione infrastrutturale, richiede competenze avanzate ed una profonda comprensione delle dinamiche interne degli strumenti impiegati, al fine di saperli adattare, ottimizzare ed estendere in funzione delle specifiche esigenze progettuali.

I risultati ottenuti attraverso i casi studio sviluppati, confermano che l'adozione consapevole ed integrata di tecnologie quali Kubernetes, Quarkus, Talos Linux, Falco, Trino e componenti custom come l'ingress controller personalizzato, consente di costruire architetture robuste, efficienti, sicure e scalabili.

In particolare, il contributo originale relativo allo sviluppo di un controller Ingress-to-Gateway mostra come sia possibile trasformare logiche dichiarative in comportamento runtime dinamico, valorizzando appieno il modello operativo offerto da Kubernetes.

La tesi si propone pertanto come un riferimento operativo e metodologico per professionisti, architetti del software e ingegneri informatici impegnati nello sviluppo di sistemi cloud-native complessi, fornendo linee guida, strumenti ed approcci progettuali in grado di forgiare soluzioni resilienti, performanti e conformi ai requisiti delle Applicazioni Web moderne.

Bibliografia

[1] *Keycloak Security*. URL: https://www.keycloak.org/ (cit. alle pp. x, 38).

[2] *MongoDB*. URL: https://it.wikipedia.org/wiki/MongoDB (cit. a p. x).

[3] *Oauth2.0*. URL: https://oauth.net/2/ (cit. a p. x).

[4] *PostgreSQL*. URL: https://www.postgresql.org/about/ (cit. a p. x).

[5] *The Importance of Embracing Change in Business*. URL: https://www.forbes.com/councils/forbesbusinesscouncil/2023/04/13/the-importance-of-embracing-change-in-business/ (cit. a p. 4).

[6] *Monolith Vs Microservices*. URL: https://www.forbes.com/councils/forbesbusinesscouncil/2023/04/13/the-importance-of-embracing-change-in-business/ (cit. a p. 6).

[7] *Che cos'è Docker*. URL: https://www.ibm.com/it-it/think/topics/docker (cit. a p. 9).

[8] *SpringBoot Framework*. URL: https://spring.io/projects/spring-boot (cit. a p. 23).

[9] *Quarkus Framework*. URL: https://quarkus.io/ (cit. a p. 23).

[10] *Talos Linux*. URL: https://www.talos.dev/ (cit. a p. 32).

[11] *Falco Security*. URL: https://falco.org/docs/ (cit. a p. 34).

[12] *Controller-Runtime*. URL: https://pkg.go.dev/sigs.k8s.io/controller-runtime (cit. a p. 42).

[13] *Kubernetes Controller*. URL: https://github.com/kubernetes/sample-controller/blob/master/docs/images/client-go-controller-interaction.jpeg (cit. a p. 43).