Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Corso di Laurea in Informatica

FunLess: Analisi e Sviluppo del Deployment su Nomad e Kubernetes

Relatore:
Chiar.mo Prof.
Gianluigi Zavattaro

Presentata da:
Erik Koci

Correlatori:
Dr.
Matteo Trentin,
Giuseppe De Palma

II sessione
Anno Accademico 2022/23

Il tempo che vi divertite a sprecare
non è tempo sprecato.

Bertrand Russell

Abstract

Gli stili architetturali dei microservizi e delle piattaforme serverless rappresentano importanti evoluzioni nel campo dell'ingegneria del software, consentendo una maggiore scalabilità e distribuzione agevole delle applicazioni software. Entrambi questi approcci richiedono la decomposizione delle applicazioni monolitiche in componenti più piccole e gestibili. I microservizi dimostrano il loro massimo potenziale quando affrontano carichi di lavoro costanti e prevedibili, mentre le piattaforme serverless offrono vantaggi economici specialmente in situazioni di carico sporadico.

Queste metodologie si basano comunemente sull'uso di container, che rappresentano un ambiente isolato e portabile per eseguire applicazioni e servizi. Essi offrono una serie di vantaggi, tra cui la separazione delle risorse, la facilità di distribuzione e la consistenza tra gli ambienti di sviluppo e produzione. Tuttavia, per gestire efficacemente un numero crescente di container, è essenziale utilizzare un orchestratore, un componente fondamentale per coordinare l'avvio, la scalabilità e la distribuzione dei sistemi.

Il presente studio si propone di esaminare in modo sistematico le caratteristiche fondamentali di due grandi orchestratori, in particolare Nomad e Kubernetes, prendendo come caso di studio il deployment di FunLess, una piattaforma serverless open-source all'avanguardia. Valuteremo le loro capacità di gestire il deployment, la scalabilità e i servizi in ambienti distribuiti. Attraverso uno studio, verranno analizzati anche gli aspetti chiave di ciascuna soluzione, comprese le loro architetture, modalità di deployment, gestione delle risorse e per concludere un benchmark riguardante l'efficienza dei due orchestratori.

Indice

Listings

Elenco delle figure

1 Introduzione

Nell'ambito dell'informatica e dell'ingegneria del software, l'evoluzione delle tecnologie legate alla gestione e al deployment delle applicazioni ha assunto un ruolo di primaria importanza. In questo contesto, l'orchestrazione dei container è emersa come una pratica chiave per ottimizzare il processo di distribuzione, gestione e scalabilità delle applicazioni su infrastrutture di tipo cloud e distribuite. La crescente complessità delle applicazioni moderne ha reso essenziale l'impiego di strumenti in grado di automatizzare tali processi, consentendo agli sviluppatori di concentrarsi maggiormente sullo sviluppo delle funzionalità.

In questo ambiente nasce il DevOps, il quale rappresenta un approccio alla gestione del ciclo di vita del software migliorandone la velocità e l'efficienza nella distribuzione. Il devOps fa uso di pratiche come l'Infrastructure as Code, che consentono di definire l'intera infrastruttura di un'applicazione mediante codice. Due esempi perfetti che rappresentano questo sistema sono Nomad [15] e Kubernetes [9], essi hanno hanno come obiettivo l'automatizzare e semplificare il provisioning, la scalabilità, il bilanciamento del carico e la gestione delle applicazioni containerizzate. Questo approccio consente di trattare l'infrastruttura stessa come parte integrante del processo di sviluppo e deployment.

La presente tesi include, oltre all'analisi di queste tecnologie, un aspetto pratico e sperimentale attraverso il deployment di FunLess [14], una piattaforma FaaS (Function-as-a-Service). Verranno confrontate due tra i più rilevanti orchestratori di container attualmente disponibili: Nomad e Kubernetes, che conferiranno un valore aggiunto alla ricerca, consentendo di valutare la loro effettiva usabilità e prestazioni nel contesto di un'applicazione reale. Entrambi questi sistemi offrono soluzioni per l'automazione delle operazioni di deployment, scalabilità e gestione dei container, ma differiscono significativamente nella loro architettura, filosofia di progettazione e caratteristiche implementative.

La struttura della tesi si articolerà nel seguente modo: inizialmente, verranno

delineati i concetti fondamentali legati all'orchestrazione dei container e alle esigenze che tali tecnologie intendono affrontare. Successivamente, si procederà con un'analisi dell'architettura di Nomad, mettendo in luce le sue componenti principali e il funzionamento generale. Tale analisi sarà seguita da un'esposizione analoga di Kubernetes.

Dopo aver stabilito una base di conoscenza per entrambe le piattaforme, la tesi procederà con il deployment di FunLess esaminando nel dettaglio le configurazioni scritte, ed evidenziando le loro similitudini e differenze in termini di architettura, scalabilità, facilità d'uso e flessibilità per poi concludere con un'analisi in termini di efficienza.

2 Prerequisiti

2.1 Cloud e Serverless

2.1.1 Cloud Computing

Il concetto di cloud computing poggia sulla creazione di un ambiente virtuale in cui risorse come server, storage, reti e applicazioni possono essere allocate, gestite e fruite in modo flessibile e dinamico. Gli utenti possono accedere a queste risorse su richiesta, eliminando la necessità di investire in infrastrutture fisiche costose e complesse. Questo modello svincola gli utenti dalla responsabilità della gestione hardware e consente loro di concentrarsi esclusivamente sullo sviluppo e l'implementazione delle loro applicazioni. Questo approccio innovativo impatta in maniera significativa su diverse sfaccettature e ha dato vita a nuovi modelli di servizio, tra cui i principali:

- **Infrastructure as a Service** (IaaS), in questo modello, vengono offerti agli utenti componenti di base dell'infrastruttura, come server virtuali, storage e reti. Gli utenti hanno un alto grado di controllo su queste risorse, poiché sono responsabili della configurazione, gestione e manutenzione del sistema operativo e delle applicazioni installate su di esso.

- **Platform as a Service** (PaaS), l'attenzione si sposta dalla gestione dell'infrastruttura all'ambiente di sviluppo e alle piattaforme di runtime. Gli utenti possono sviluppare, testare e implementare applicazioni senza doversi preoccupare dei dettagli dell'infrastruttura sottostante. Questo modello accelera il processo di sviluppo e semplifica la gestione delle risorse.

- **Software as a Service** (SaaS) vengono fornite applicazioni completamente gestite agli utenti attraverso il cloud. Gli utenti possono accedere e utilizzare le applicazioni tramite un browser web, senza dover installare o gestire software localmente. Questo modello offre agilità e facilità d'uso, riducendo il carico amministrativo sugli utenti.

- **Function as a Service** (FaaS), si concentra sull'esecuzione di singole funzioni o frammenti di codice in risposta a eventi specifici, senza la necessità di gestire l'infrastruttura sottostante. FaaS offre un grado elevato di scalabilità automatica: le funzioni vengono allocate solo quando richieste e possono essere replicate su più istanze per soddisfare un carico elevato.

Un vantaggio centrale del cloud computing è la scalabilità su richiesta. Questo significa che le risorse possono essere aumentate o ridotte dinamicamente in base alle esigenze dell'applicazione. In momenti di picco di carico, le risorse possono essere aumentate per garantire prestazioni ottimali. Al contrario, in momenti di carico leggero, è possibile ridurre le risorse per risparmiare costi. Questa scalabilità elastica è fondamentale per soddisfare le esigenze mutevoli dei servizi e ridurre gli sprechi di risorse.

2.1.2 Il paradigma Serverless

Il paradigma del serverless costituisce un altro concetto molto importante. Esso rappresenta un approccio alla realizzazione di sistemi senza doversi preoccupare della gestione dell'infrastruttura sottostante, come server virtuali o container. Possiamo evidenziare due principali tipologie di servizi che rappresentano delle specifiche implementazione del paradigma serverless:

- **FaaS** (Function as a Service): che mira esclusivamente all'esecuzione della funzione di codice fornito dall'utilizzatore.
- **BaaS** (Backend as a Service): Utile per fornire un modo per collegare applicazioni a un backend cloud.

Seppur alcune caratteristiche siano analoghe a quelle delineate dal cloud, l'approccio serverless spesso introdurrà migliorie:

- **Estrema Scalabilità**: Attuando una scomposizione del servizio in unità atomiche, ovvero le funzioni o operazioni costitutive, il fornitore diventa capace di gestire la scalabilità dinamica in risposta alla domanda.
- **Semplicità d'Uso**: La responsabilità degli sviluppatori si limiterà all'implementazione della logica di business, lasciando al fornitore la cura di rendere le funzioni disponibili.

Tuttavia, il paradigma del serverless, pur contribuendo alla soluzione di molteplici problematiche, presenta alcune criticità, ad esempio:

- **Natura Effimera** dell'Esecuzione e Sistemi di Database: La breve durata dell'esecuzione delle funzioni nel contesto serverless non si adatta in modo ottimale ai sistemi di database tradizionali, i quali richiedono connessioni prolungate in grado di gestire numerose interrogazioni prima della chiusura. Benché l'accesso ai database sia possibile attraverso le funzioni serverless, la rapida apertura e chiusura delle connessioni per ogni invocazione impone uno stress significativo ai database.

- **Funzioni Stateless**: Si richiede che le funzioni siano stateless, ossia prive di stato, onde evitare la conservazione di qualsivoglia stato intrinseco da un'invocazione all'altra. Tale vincolo si rende necessario per consentire al fornitore la flessibilità di commutare tra macchine dedicate all'esecuzione delle funzioni in base alle necessità. In tal senso, ogni forma di conservazione dati tra le chiamate deve fare affidamento su servizi esterni, compromettendo l'efficacia delle comunicazioni in tempo reale a lunga durata.

Il paradigma del serverless rappresenta pertanto una prospettiva d'innovazione nel contesto dell'elaborazione applicativa, consentendo agli sviluppatori di concentrarsi esclusivamente sulla logica di business, senza dover affrontare l'ingente complessità dell'infrastruttura sottostante.

2.1.3 Docker

L'obiettivo principale di Docker è quello di creare un ambiente isolato e autonomo, noto come container, in cui le applicazioni possono essere eseguite insieme alle loro dipendenze senza influenzare l'ambiente circostante. Questa tecnologia offre un approccio più leggero e portatile rispetto alle tradizionali macchine virtuali.

Il funzionamento di Docker si basa sulla creazione e gestione di container. Un container è un'istanza di un'immagine, che rappresenta un pacchetto contenente l'applicazione e tutte le sue dipendenze, come librerie, file di configurazione e codice eseguibile. L'immagine è costruita utilizzando un file chiamato "Dockerfile", che definisce le istruzioni per creare l'ambiente dell'applicazione.

Quando un'immagine viene eseguita, Docker crea un container isolato utilizzando le risorse del sistema operativo host. Questo container condivide il kernel del sistema operativo host, ma è separato in modo netto dagli altri container e dal sistema host stesso. Ciò consente di avere un ambiente consistente tra diverse piattaforme e assicura che le applicazioni siano isolate e non interferi-

scano tra loro, riducendo così l'overhead e la complessità tipica delle soluzioni di virtualizzazione.

Docker offre un'ampia gamma di strumenti per la gestione dei container. Inoltre permette di collegare container in reti virtuali e di creare volumi per memorizzare dati in modo persistente anche dopo che il container è terminato.

Le applicazioni containerizzate con Docker possono essere facilmente distribuite su diverse infrastrutture, come server locali, cloud pubblici e ambienti di sviluppo. Questa portabilità è resa possibile dalla natura isolata dei container e dalla capacità di definire in modo dettagliato le dipendenze e l'ambiente dell'applicazione attraverso il Dockerfile.

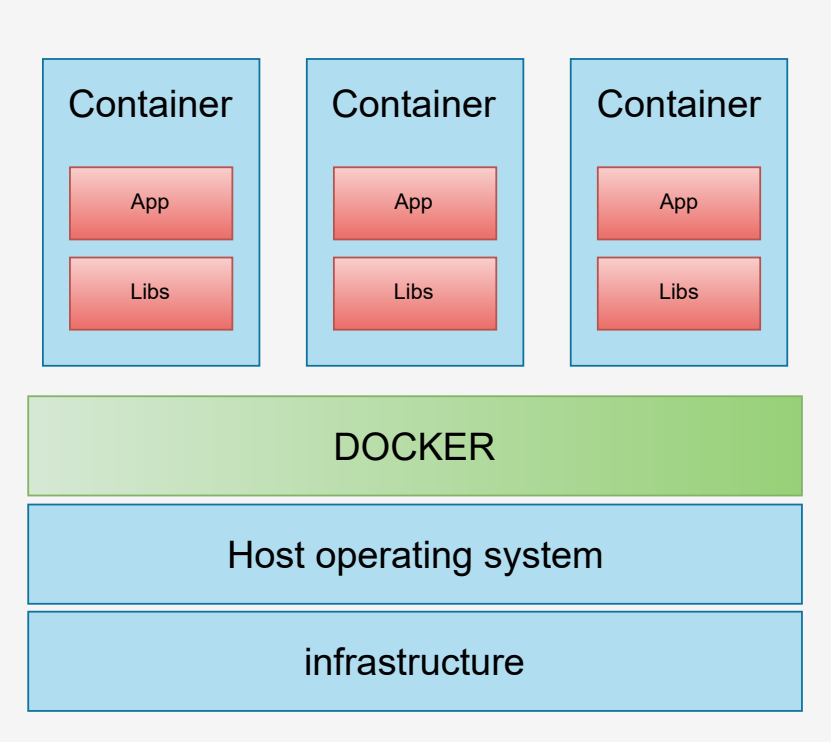

Figura 2.1: Architettura di docker

2.1.4 Sistemi di Orchestrazione

I sistemi di orchestrazione [5] rivestono un ruolo fondamentale nell'ecosistema dell'elaborazione distribuita, consentendo la gestione efficiente e la scalabilità delle applicazioni composte da componenti interconnessi, quali container o microservizi. L'implementazione di applicazioni su infrastrutture distribuite

comporta la necessità di coordinare e ottimizzare il deployment, il bilanciamento del carico, il monitoraggio e l'aggiornamento dei componenti stessi. I sistemi di orchestrazione sono stati sviluppati per soddisfare queste complesse esigenze, semplificando e automatizzando il ciclo di vita delle applicazioni. Ciascuno adotta un approccio specifico per gestire l'esecuzione su infrastrutture distribuite. Tra le principali soluzioni di orchestrazione, emergono:

- Kubernetes
- Nomad

Kubernetes, uno dei sistemi di orchestrazione maggiormente utilizzato, sorge da un progetto open-source sviluppato da Google. Si concentra sulla definizione di uno strato di astrazione e controllo sopra il cluster di host, consentendo una distribuzione trasparente delle applicazioni containerizzate. Kubernetes introduce il concetto di *pod*, una unità logica che raggruppa uno o più container. Ciò agevola la locazione dei container che condividono risorse e interconnessioni, ottimizzando le prestazioni complessive dell'applicazione. Oltre al bilanciamento del carico automatico e alla scalabilità orizzontale, Kubernetes offre strategie di deployment avanzate per il rilascio senza interruzioni, assicurando un'alta disponibilità e tempi di inattività minimi.

Nomad costituisce un'alternativa nel panorama degli orchestratori, focalizzandosi sull'automazione delle operazioni di scheduling e gestione delle risorse. La sua flessibilità gli consente di gestire carichi di lavoro containerizzati e non containerizzati su infrastrutture eterogenee. Nomad è orientato a semplificare l'esperienza di deployment, fornendo un'interfaccia pulita e un approccio più leggero rispetto a soluzioni più complesse.

Ciascuna di queste soluzioni presenta vantaggi specifici in termini di funzionalità, complessità e adattabilità, consentendo alle organizzazioni di selezionare l'orchestratore più idoneo alle proprie esigenze e ai requisiti delle applicazioni da gestire. Successivamente andremo ad analizzare nel dettaglio le differenze tra Nomad e kubernetes.

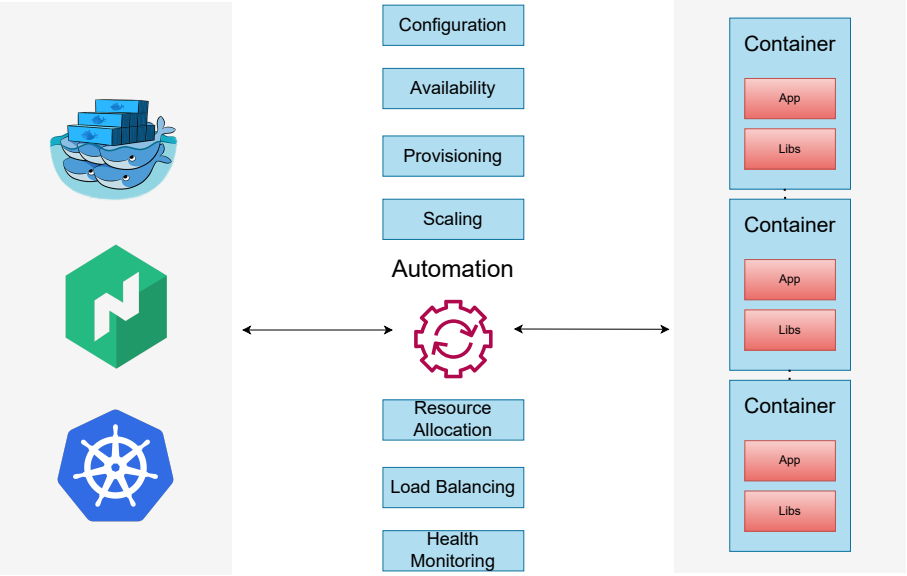

Figura 2.2: Ruolo degli orchestratori

2.2 FunLess

In questa sezione, concentriamo la nostra attenzione sull'integrazione di una piattaforma FaaS chiamata FunLess. Inizieremo procedendo a una panoramica, esaminando le sue caratteristiche fondamentali. Questo approccio sarà prezioso in seguito nell'ambito dell'analisi tra i due orchestratori, fornendo un quadro iniziale.

FunLess è una piattaforma serverless, orientata alla ricerca e sviluppata con l'obiettivo di coniugare la scalabilità tipica dell'ambiente BEAM[1] con la velocità e la sicurezza offerte da WebAssembly (Wasm). Analizzeremo gli aspetti architetturali e concettuali di FunLess, descrivendo le componenti chiave e la sua struttura operativa.

Il nucleo centrale di FunLess è rappresentato dalla componente Core che espone un'API[2], gestendo le funzioni e i moduli, comunicando con un database Postgres. Svolge inoltre il ruolo di scheduler, selezionando uno dei componenti Worker disponibili al fine di eseguire una richiesta di invocazione di funzione.

[1] Il termine BEAM si riferisce alla macchina virtuale Erlang

[2] Un'API (Application Programming Interface) è un insieme di regole e protocolli che consentono a diverse applicazioni software di comunicare tra loro e scambiare dati o funzionalità in modo strutturato.

La componente Worker rappresenta l'esecutore effettivo delle funzioni. Il Worker esegue le funzioni definite dall'utente tramite il runtime Wasmtime. Il Worker implementa un sistema di *cache* per evitare di inizializzare più volte le stesse funzioni. Quando una richiesta di invocazione arriva dal Core, il Worker verifica la presenza della funzione nella cache e, in caso contrario, richiede al Core il binario Wasm corrispondente. Il Worker esegue la funzione e restituisce il risultato al Core.

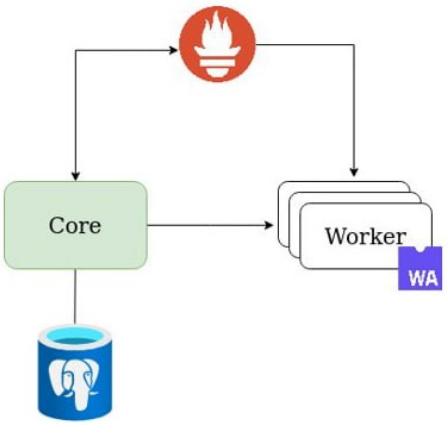

Figura 2.3: Architettura di FunLess

Inoltre, FunLess fa uso di raccolta di metriche dalla piattaforma utilizzando in particolare Prometheus e Filebeat. Sono integrati anche un insieme di microservizi che svolgono ruoli cruciali nell'ambiente complessivo, in particolare:

- **Elasticsearch**: Riveste la funzione di archivio dati, predisposta per consentire ricerche avanzate.
- **Kibana**: Agisce come interfaccia di visualizzazione e analisi dei dati immagazzinati.
- **Filebeat**: Opera come strumento chiave per la raccolta e l'inoltro dei dati di log, assicurando così una raccolta accurata delle operazioni.

Figura 2.4: EFK stack

2.2.1 Compilazione e distribuzione di funzioni in FunLess

Per iniziare ad utilizzare FunLess[13] basta scaricare l'eseguibile compresso presente all'interno della documentazione ufficiale.

```
wget https://github.com/FunLessdev/fl-cli/releases/download/v0.4.0/fl-
    v0.4.0-linux-amd64.tar.gz
```

Listing 2.1: Installazione di FunLess

Una volta terminata l'installazione è possibile effettuare il deployment di FunLess usando la CLI offerta. Essa si basa internamente su Docker Compose, pertanto è fondamentale disporre di una versione recente di Docker.

La CLI effettuerà il download e il lancio di quattro container: uno per il componente Core, uno per il Worker, uno per Prometheus e uno per Postgres.

```
fl admin deploy docker up
```

Listing 2.2: Deploy di FunLess

FunLess utilizza il runtime di WebAssembly[3] tramite Wasmtime per eseguire le funzioni. Al momento, supporta Rust e JavaScript, Per creare una nuova funzione, basta usare il comando seguente da CLI.

```
fl fn new hello_funless -l rust
```

Listing 2.3: Creare una funzione in FunLess

[3]Il runtime di WebAssembly è un ambiente di esecuzione progettato per eseguire codice compilato in formato WebAssembly (Wasm)

Questo scaricherà la cartella del template che contiene i template per Rust e JavaScript, e utilizzerà il template Rust per creare un nuovo progetto chiamato hello_funless. La cartella hello_funless conterrà un file Cargo.toml e un file lib.rs con la funzione fl_main, che può essere modificata a piacimento.

Una volta creata la funzione, essa può essere compilata e distribuita su FunLess. Il comando *create* si occuperà sia della compilazione che dell'upload:

```
fl fn create hello_funless hello_funless/ --language rust
```

Listing 2.4: Compilazione e upload della funzione

Il primo argomento del comando create è il nome della funzione all'interno di FunLess, mentre il secondo è il percorso alla directory con il codice. L'opzione *–language* specifica invece il linguaggio di programmazione da compilare.

Infine, una volta eseguita la compilazione e l'upload si può procedere all'invocazione della funzione. Poiché la funzione accetta un argomento JSON, è possibile utilizzare il flag *-j* per passare l'oggetto JSON con la chiave 'name' come input:

```
fl fn invoke hello -j '{"name": "FunLess"}'
```

Listing 2.5: Compilazione e upload della funzione

Dunque ricapitolando, la richiesta di invocazione sarà ricevuta dal componente principale, passata al componente worker e la funzione wasm verrà eseguita. Il risultato verrà restituito al componente principale e quindi alla CLI, che lo stamperà sulla console.

2.3 Nomad

Nomad è un sistema di orchestrazione e gestione delle risorse sviluppato da HashiCorp[2]. HashiCorp è un'azienda specializzata nello sviluppo di strumenti per l'automazione dell'infrastruttura, noti anche come strumenti di Infrastructure as Code (IaC).

Nomad è progettato per gestire la distribuzione e l'esecuzione di applicazioni e carichi di lavoro su un'infrastruttura distribuita. Questa piattaforma fornisce un ambiente in cui è possibile definire, pianificare e allocare risorse computazionali in modo dinamico, tenendo conto delle esigenze delle applicazioni e delle risorse disponibili nell'ambiente di calcolo.

Una caratteristica chiave è la sua capacità di gestire carichi di lavoro eterogenei. Questo significa che può coordinare l'esecuzione di applicazioni containerizzate e macchine virtuali offrendo un'ampia flessibilità agli sviluppatori e agli amministratori di sistema.

Nomad si basa su un approccio *task-based* e consolida l'idea di definire l'infrastruttura necessaria per un'applicazione in modo dichiarativo, piuttosto che configurarla manualmente. Queste specifiche includono dettagli come il numero di istanze, le risorse assegnate a ciascuna istanza, le dipendenze, i vincoli di posizionamento e altre proprietà.

2.3.1 Architettura di Nomad

Una delle caratteristiche fondamentali di Nomad è la sua architettura distribuita e resiliente. Il sistema è composto da diversi componentitra cui:

- **Agenti**: sono installati su ogni nodo dell'infrastruttura e sono responsabili dell'esecuzione dei carichi di lavoro.
- **Server**: mantengono lo stato globale del sistema e coordinano la pianificazione dei carichi di lavoro
- **Scheduler**: parte integrante dei server, è responsabile della distribuzione delle attività in base alle risorse disponibili e alle specifiche dei carichi di lavoro.

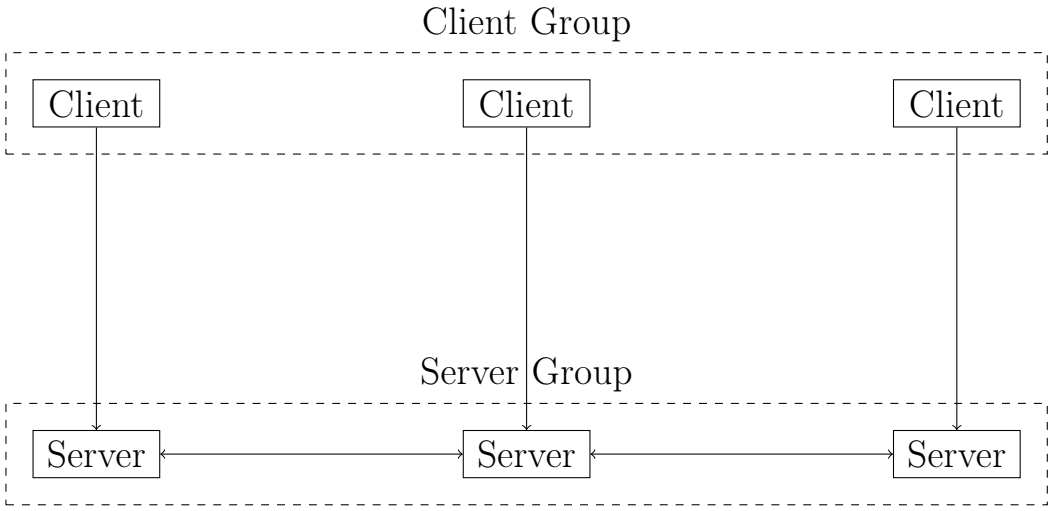

Figura 2.5: Architettura Nomad

Nomad offre anche un'interfaccia utente e una serie di API che consentono agli amministratori di configurare e gestire l'infrastruttura e agli sviluppatori di dichiarare i requisiti delle loro applicazioni.

Un aspetto rilevante da considerare è l'integrazione di Nomad all'interno della configurazione con altre tecnologie, attualmente supportate:

- Docker
- File binari
- Applicazioni Java
- Macchine Virtuali QEMU

E' possibile identificare alcune tipologie di Job Type fondamentali per definire la natura e il comportamento di un task all'interno di un cluster Nomad. Ogni Job Type riflette una modalità specifica in cui sarà eseguito e gestito, consentendo agli utenti di adattare l'orchestrazione alle esigenze specifiche del carico di lavoro. Le principali tipologie sono:

1. **Service**: è uno dei Job Types chiave in Nomad, sottolineando la natura a lungo termine delle applicazioni in esecuzione. Questa categoria è pensata per applicazioni che richiedono disponibilità continua e risorse allocate costantemente. In questo contesto, l'algoritmo di ranking utilizzato seleziona il nodo ottimale per ospitare un gruppo di task, basandosi su un algoritmo di best fit[4].

2. **Batch**: riflette il bisogno di eseguire lavori in modalità periodica o su base intermittente. Questa categoria è adatta per le elaborazioni di dati che richiedono esecuzioni pianificate, come l'elaborazione di calcolo intensivo su dataset specifici.

3. **System**: è concepito per la registrazione di lavori che devono essere eseguiti su tutti i client che soddisfano i vincoli definiti. Inoltre, questo scheduler è attivato quando i client si uniscono al cluster o passano allo stato ready. Ciò significa che tutti i lavori di sistema registrati vengono rivalutati e i relativi task vengono posizionati sui nodi appena disponibili, a condizione che i vincoli siano rispettati.

4. **Sysbatch**: è utilizzato per lavori che devono essere eseguiti su tutti i client che soddisfano i vincoli definiti. Analogamente al System, il

[4]Metodo di allocazione della memoria che propone di assegnare un processo alla partizione di memoria più piccola sufficiente per contenerlo, tra quelle disponibili.

Sysbatch pianifica i lavori, ma a differenza dei lavori di tipo batch, una volta che un task termina con successo, non viene riavviato su quel client.

2.3.2 Consul

Piattaforme esterni come Consul[4] offrono la possibilità di automatizzare la scoperta dei servizi e garantire la coerenza delle configurazioni in ambienti distribuiti. Il suo approccio basato su agenti decentralizzati rende possibile la registrazione, la scoperta e la gestione dei servizi in maniera automatica e dinamica. Questo approccio si applica a scenari di orchestrazione diversi, consentendo l'implementazione di architetture resilienti e scalabili. Le funzionalità di Consul possono essere riassunte nei seguenti punti:

- **Service Discovery**: Consul offre un meccanismo automatizzato per registrare e scoprire i servizi all'interno di un ambiente distribuito. Questo rende possibile la comunicazione tra i servizi senza l'utilizzo di indirizzi IP statici.

- **Health Checking**: La verifica periodica dello stato dei servizi da parte di Consul garantisce che solo i servizi sani siano considerati per le comunicazioni. Questo contribuisce alla resilienza dell'architettura.

- **Configurazione Centralizzata**: Fornisce uno storage chiave-valore che può essere utilizzato per la gestione centralizzata delle configurazioni. Questo separa la configurazione dall'applicazione stessa, semplificando il processo di aggiornamento e manutenzione.

- **Multi-Datacenter Support**: La possibilità di supportare architetture multi-datacenter facilita la gestione dei servizi distribuiti su aree geografiche diverse.

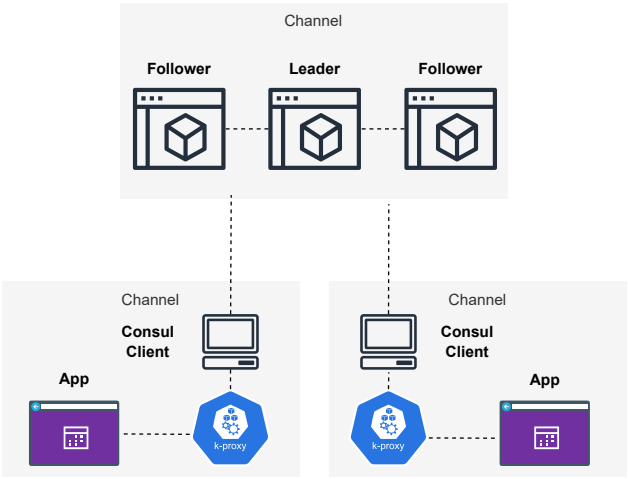

Figura 2.6: Architettura di Consul

2.3.3 Protocolli di comunicazione

Durante lo scambio di informazioni tra i client presenti all'interno del cluster Nomad viene utilizzato un protocollo proposto da Diego Ongaro e John Ousterhout, chiamato protocollo Raft [16] che mira a semplificare il processo di comprensione e implementazione rispetto ad altri algoritmi di consenso più complessi, come il Paxos[12].

L'obiettivo principale del protocollo Raft è garantire che un gruppo di nodi in un sistema distribuito possa raggiungere un accordo su uno stato consistente nonostante le possibili guasti e perdite di messaggi. Il protocollo opera attraverso la ripartizione delle responsabilità in tre ruoli chiave: leader, follower e candidato.

Il ruolo di leader è centrale nel protocollo Raft. Il leader è responsabile di inizializzare le operazioni di consenso, gestire le richieste di scrittura dai client e replicare i dati nei nodi follower. I nodi follower seguono semplicemente i comandi del leader e replicano lo stato. Nel caso in cui il leader diventi inattivo, si avvia una procedura di elezione per selezionare un nuovo leader tra i nodi candidati.

La procedura di elezione dei candidati è un aspetto cruciale del protocollo Raft. I nodi possono passare allo stato di candidato quando ritengono che il leader sia diventato inattivo. I candidati richiedono il voto degli altri nodi per diventare

leader. Per essere eletto, un candidato deve ricevere voti dalla maggioranza dei nodi del cluster. Questo meccanismo contribuisce a garantire che un singolo leader venga scelto e che non si verifichino conflitti di leadership multipla. Una volta eletto, il leader coordina le operazioni di scrittura e garantisce la coerenza dei dati replicati tra i nodi follower.

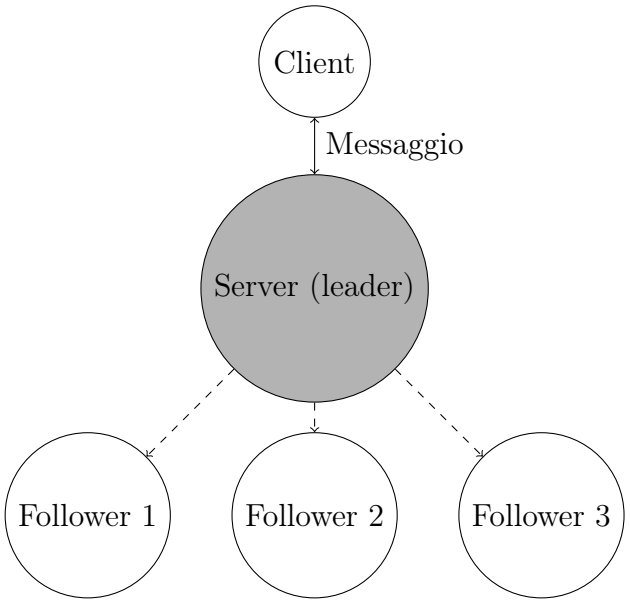

Figura 2.7: Esempio di comunicazione nel protocollo Raft.

Il protocollo Gossip [18] invece, è protocollo di comunicazione peer-to-peer utilizzato da Nomad per la diffusione delle informazioni di stato tra i nodi del cluster. Nel contesto di Nomad, il Gossip viene utilizzato per diverse finalità:

- **Scoperta del Cluster**: Quando un nuovo nodo si unisce al cluster Nomad o un nodo esistente viene rimosso, il protocollo di Gossip viene utilizzato per diffondere queste informazioni a tutti gli altri nodi nel cluster. Questo aiuta a mantenere una visione coerente dello stato del cluster tra tutti i nodi.

- **Diffusione dello Stato**: Nomad utilizza il Gossip per diffondere lo stato dei job e delle allocazioni (istanze di lavoro) all'interno del cluster. Quando uno stato cambia, come l'allocazione di una nuova risorsa o l'aggiornamento di uno stato, questa informazione viene diffusa attraverso il protocollo di Gossip. Questo permette a tutti i nodi di essere a conoscenza dei cambiamenti nello stato del sistema.

- **Rilevamento dei Nodi Disponibili**: Il protocollo di Gossip aiuta a identificare i nodi attivi e disponibili nel cluster. I nodi comunicano tra loro attraverso il Gossip per verificare quali nodi sono attivi e pronti a partecipare alle attività di gestione dei carichi di lavoro.

Poiché Nomad opera in ambienti distribuiti e dinamici, questo protocollo permette ai nodi di acquisire velocemente informazioni di stato aggiornate senza dover dipendere da un'entità centrale o da una singola fonte di verità. Questo rende il sistema più resiliente ai guasti e alle variazioni della topologia di rete.

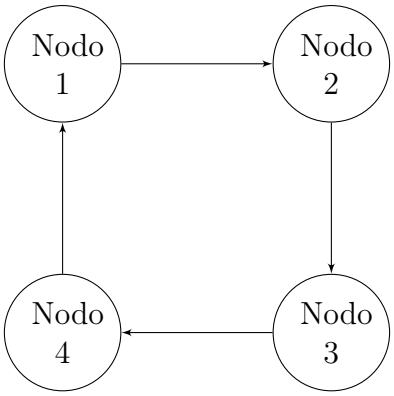

Figura 2.8: Comunicazione protocollo gossip

2.3.4 Struttura del file di configurazione

La configurazione in Nomad è espressa attraverso file di definizione dei job. Un job rappresenta un insieme di task correlati che devono essere eseguiti all'interno dell'ambiente. La progettazione accurata di questi file è essenziale per garantire il corretto posizionamento dei task, la scalabilità e la gestione delle risorse.

I file di configurazione in Nomad seguono un formato dichiarativo basato su HCL (HashiCorp Configuration Language). Ogni job è definito all'interno di un blocco di configurazione che contiene elementi chiave.

La capacità di specificare il numero di istanze di un gruppo di task attraverso il parametro count riflette l'attenzione di Nomad alla scalabilità. Questo parametro permette di adattare dinamicamente il numero di task in base alla richiesta di carico e alle risorse disponibili.

Inoltre l'approccio di Nomad all'esecuzione dei task attraverso driver[5] come ad esempio docker, dimostra un alto grado di astrazione. Questa caratteristica consente agli utenti di lavorare con diverse tecnologie di container senza dover modificare radicalmente la configurazione dei job. L'uso di variabili di enviroment e parametri specifici del driver offre ulteriori possibilità di personalizzazione.

2.4 Kubernetes

2.4.1 Architettura di kubernetes

L'architettura di Kubernetes [11] si compone di vari componenti, ognuno dei quali svolge un ruolo specifico nella gestione dei contenitori e delle applicazioni. Al centro dell'architettura si trova il *control plane*, che è costituito da diversi elementi, tra cui l'API server, lo scheduler, il controller-manager e etcd. L'API server agisce come il punto di ingresso per le richieste e le interazioni con il cluster Kubernetes, esponendo un'interfaccia RESTful per la gestione delle risorse. Lo scheduler decide su quale nodo eseguire nuovi pod basandosi su variabili come la disponibilità delle risorse e i requisiti di sistema. Il controller-manager è responsabile del mantenimento dello stato desiderato del sistema, attraverso la verifica costante delle differenze tra lo stato corrente e lo stato desiderato delle risorse.

Un elemento chiave nell'architettura di Kubernetes è anche etcd, un database distribuito che memorizza lo stato del cluster e le configurazioni delle risorse. Etcd si rileva fondamentale nella coerenza e nella persistenza delle informazioni di configurazione, essenziali per il funzionamento stabile del cluster.

Sui nodi del cluster, noti anche come worker nodes, vengono eseguiti i carichi di lavoro delle applicazioni sotto forma di pod. Ogni nodo ospita Kubelet, un agente che interagisce con il control plane per garantire che i pod vengano eseguiti in modo corretto e che le risorse siano allocate in base alle specifiche. Kube-proxy invece gestisce sul nodo le comunicazioni di rete tra i pod, garantendo il corretto instradamento dei pacchetti all'interno e all'esterno del cluster.

Un aspetto rilevante dell'architettura di Kubernetes è rappresentato dai services, che consentono la comunicazione tra i pod e la loro scoperta all'interno

[5]I driver agiscono come interfacce di collegamento tra Nomad e le diverse tecnologie di esecuzione

del cluster. I services definiscono un insieme di regole per instradare il traffico ai pod, indipendentemente dalla loro localizzazione fisica. Inoltre, Kubernetes offre anche il concetto di persistent volume (PV) per la gestione dei dati persistenti tra i cicli di vita dei pod. Questo consente alle applicazioni di mantenere i loro dati anche quando i pod vengono terminati o riavviati.

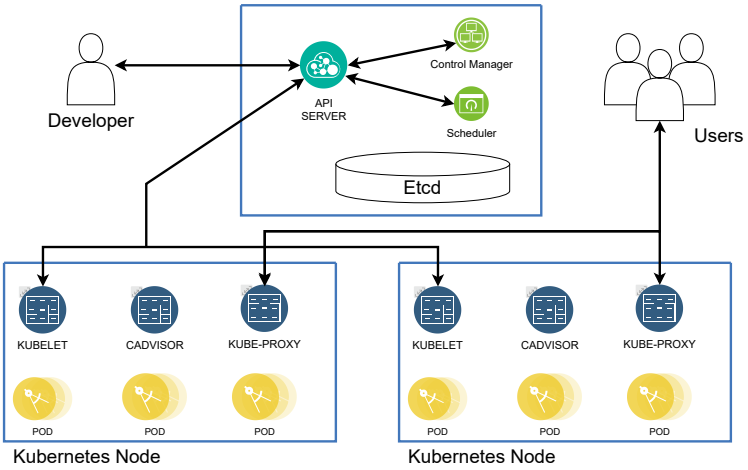

Figura 2.9: Architettura di Kubernetes

2.4.2 Kubeadm e Kubectl

All'interno del deployment di FunLess usando kubernetes come orchestratore sono stati utilizzati due tool fondamentali, Kubeadm e Kubectl.
Per inizializzare e gestire cluster Kubernetes viene usato Kubeadm[8]. L'obiettivo principale di Kubeadm è quello di ridurre la complessità e l'approccio manuale necessario per configurare un cluster Kubernetes funzionante. Kubeadm offre una serie di funzionalità chiave che semplificano la creazione e la gestione dei cluster Kubernetes:

- **Inizializzazione del Control Plane**: Automatizza la creazione e la configurazione dei componenti principali del control plane di Kubernetes, come l'API server, il controller manager e il scheduler.
- **Joining dei Nodi**: Semplifica anche il processo di aggiunta dei worker node al cluster. Fornisce un token di join che permette ai nodi di entrare nel cluster.

- **Gestione delle Versioni**: Consente agli utenti di specificare le versioni di Kubernetes da utilizzare.
- **Configurazione del Network**: Fornisce opzioni per la configurazione del network, consentendo agli utenti di selezionare la soluzione di rete più adatta alle proprie esigenze.
- **Integration con Cloud Providers**: Supporta l'integrazione con diversi fornitori cloud, facilitando l'implementazione di cluster Kubernetes su piattaforme cloud.

In Kubernetes inoltre e' possibile gestire le risorse al loro interno, in due modi principali: l'accesso diretto all'API, oppure l'utilizzo di Kubectl [6]. Esso rappresenta l'interfaccia a riga di comando principale per la gestione delle operazioni all'interno di un ambiente Kubernetes. Le principali funzionalità di Kubectl includono:

- **Creazione e Gestione di Risorse**: Consente agli utenti di creare, aggiornare e rimuovere risorse all'interno di un cluster Kubernetes.
- **Controllo dello Stato del Cluster**: Gli utenti possono utilizzare Kubectl per verificare lo stato del cluster, controllare l'utilizzo delle risorse, monitorare l'esecuzione dei pod e accedere ai log e agli eventi.
- **Scaling e Aggiornamenti**: Consente di scalare in modo dinamico le risorse e gestire gli aggiornamenti delle applicazioni.
- **Interazione con i Pod**: Gli utenti possono accedere direttamente ai terminali dei pod, consentendo la diagnosi e il debug direttamente all'interno dei container.
- **Gestione delle Configurazioni**: Supporta la gestione delle configurazioni sia attraverso i file YAML sia attraverso l'interfaccia a riga di comando.
- **Gestione delle Politiche di Accesso**: Permette di gestire i ruoli, i cluster role, le role binding e le cluster role binding, definendo chi ha accesso alle risorse e alle azioni all'interno del cluster.

2.4.3 Struttura del file di configurazione

La configurazione in Kubernetes è espressa attraverso file di definizione dei manifesti. Un manifest rappresenta l'intera specifica di un'istanza di una risorsa,

che può essere un pod, un deployment, un service e così via. La progettazione accurata di questi file è di fondamentale importanza per assicurare l'adeguata distribuzione delle risorse, la scalabilità e il corretto funzionamento delle applicazioni all'interno del cluster.

I file di configurazione in Kubernetes seguono un formato dichiarativo basato su YAML (YAML Ain't Markup Language). Ogni risorsa è definita all'interno di un blocco di configurazione che contiene diversi campi chiave. Per quanto riguarda la verbosità della configurazione l'approccio più esplicito e dettagliato, consente un maggiore controllo sulla configurazione delle risorse. Sebbene ciò possa comportare una maggiore quantità di codice per definire la stessa risorsa, offre un livello più profondo di gestione.

In Kubernetes è possibile definire diversi tipi di entità in un file manifest di configurazione. Esse rappresentano le unità fondamentali di gestione e organizzazione delle risorse all'interno di un cluster. Ogni entità rappresenta un aspetto specifico del sistema, come un'applicazione, un servizio di rete o un'istanza di un'applicazione in esecuzione. Questa divisione offre un'astrazione per la gestione delle risorse distribuite in un ambiente di orchestrazione dei container. I principali rappresentanti sono:

1. **Pod**: è l'unità più piccola di esecuzione in Kubernetes, rappresentando un gruppo di uno o più container che condividono spazio di archiviazione e networking. I container all'interno di un pod condividono lo stesso contesto e comunicano tra loro attraverso localhost.

2. **ReplicaSet**: assicura che un numero specificato di repliche dei pod sia sempre in esecuzione. Questo è utile per garantire la disponibilità delle applicazioni e gestire la scalabilità in modo automatico.

3. **Deployment**: fornisce un modo dichiarativo per gestire le applicazioni. Gestisce i ReplicaSet e offre funzionalità di aggiornamento e rollback delle versioni delle applicazioni.

4. **StatefulSet**: gestisce le applicazioni che richiedono uno stato persistente e un'identità unica. È adatto per database e applicazioni che mantengono uno stato.

5. **DaemonSet**: assicura che un'istanza di un pod venga eseguita su ogni nodo del cluster. È utilizzato per operazioni come monitoraggio e collezzione di log.

6. **Job**: è utilizzato per eseguire un processo batch, assicurando che venga eseguito con successo una volta e poi terminato.

7. **Service**: è un oggetto di rete che esegue il bilanciamento del carico tra i pod e fornisce un endpoint stabile per accedere ai pod di un'applicazione.
8. **PersistentVolume**: permette la creazione di volumi utilizzabili nel cluster.
9. **ConfigMap**: separa la configurazione dell'applicazione dal suo codice, consentendo di modificare la configurazione senza dover riavviare l'applicazione.
10. **Secret**: gestisce dati sensibili, come password e chiavi API, in modo sicuro, consentendo un accesso separato e crittografato.

2.5 Scalabilità ed ecosistema di supporto a confronto

Entrambi i sistemi di orchestrazione, Kubernetes e Nomad, supportano una grande mole di lavoro. Kubernetes è noto per supportare cluster fino a 5.000 nodi e un totale di 300.000 container, come indicato nella sua documentazione ufficiale [7]. Tuttavia, all'aumentare delle dimensioni del cluster, le interazioni complesse tra i componenti possono comportare una maggiore complessità nella gestione e nella manutenzione complessiva del sistema.

Nomad, d'altro canto, ha dimostrato una notevole capacità di scalabilità in ambienti di produzione del mondo reale. È stato testato con successo in cluster che superano i 10.000 nodi. Ciò significa che Nomad può gestire configurazioni più estese rispetto a Kubernetes. Inoltre, Nomad è progettato per gestire nativamente implementazioni multi-cluster senza dover ricorrere a cluster su cluster, semplificando notevolmente la scalabilità attraverso diverse zone di disponibilità, regioni e data center.

Un ulteriore punto a favore di Nomad è rappresentato dai test di scalabilità significativi cui è stato sottoposto. Questi includono la sfida di un milione di container nel 2016 e la sfida dei due milioni di container nel 2020. Tali test hanno dimostrato che Nomad può operare in scenari estremi senza compromettere le prestazioni.

Un altro aspetto fondamentale da assorvare è quello relativo alla documentazione e la comunità degli sviluppatori. L'ecosistema di supporto di Nomad e quello di Kubernetes hanno caratteristiche e dimensioni completamente diverse.

Nel caso di Nomad, è importante riconoscere che, sebbene stia crescendo, non raggiunge ancora la vastità e la solidità dell'ecosistema di Kubernetes. Durante il nostro percorso di apprendimento su Nomad, abbiamo incontrato alcune difficoltà nella risoluzione di problemi specifici durante la configurazione. Tuttavia, HashiCorp, l'azienda sviluppatrice di Nomad, fornisce documentazione completa e guide che sono utili per familiarizzare con la piattaforma.

Kubernetes, al contrario, beneficia di un ecosistema di supporto ampio e ben consolidato. La sua comunità di sviluppatori è vasta e attiva, con molte risorse disponibili come forum di discussione. Kubernetes offre una documentazione esaustiva e una vasta gamma di strumenti di supporto di terze parti, tra cui plugin e soluzioni per la gestione operativa.

È evidente che l'ecosistema di Nomad non è ancora paragonabile a quello di Kubernetes in termini di dimensioni e maturità. Ciò può comportare una maggiore sfida nell'ottenere risposte a domande specifiche e una minore disponibilità di guide pratiche avanzate, limitando il processo di apprendimento e la capacità di adottare funzionalità avanzate in modo agevole.

3 Analisi Pratica

Nel presente capitolo, esploreremo il processo di deployment usando FunLess come caso di studio. In particolare, ci concentreremo sulle configurazioni dei servizi presenti, mostrando le differenze effettive tra i due orchestratori precedentemente introdotti, Nomad e Kubernetes.

Durante lo svolgimento di queste analisi è stato anche testato l'utilizzo di questi orchestratori creando dei veri e propri cluster per avere maggiore potenza di calcolo.
Di seguito è presente una spiegazione riguardante le specifiche architetturali delle macchine utilizzate, l'installazione dei due orchestratori e successivamente verranno presentati i relativi file di configurazione e le integrazioni effettuate con servizi esterni. In particolare nella prima parte vedremo le configurazioni di kubernetes analizzando il deployment di FunLess, successivamente procederemo con l'analisi omologa utilizzando Nomad nella parte di integrazione dei servizi dello stack EFK formato da Elasticsearch, Filebeat e Kibana. Per concludere verrà analizzato un benchmark effettuato relativo alle prestazioni riguardante i due orchestratori.

3.1 Specifiche architetturali

Nel contesto dell'implementazione del deployment, è stata adottata un'architettura basata su un cluster di tre macchine virtuali. Questo cluster è stato creato utilizzando Proxmox[1]. Ciascuna macchina virtuale è stata configurata con sistema operativo Debian 12. Ogni nodo all'interno di questo cluster presentava le seguenti specifiche:

- RAM: Ogni macchina virtuale è stata dotata di 4 gigabyte di RAM. Questa quantità è stata dimensionata in base alle esigenze del deployment e alla sua capacità di gestire carichi di lavoro.

[1]Proxmox è una piattaforma di virtualizzazione open-source che consente la gestione centralizzata di macchine virtuali, fornendo funzionalità di virtualizzazione e clustering per server.

- CPU: Ogni macchina virtuale è stata configurata con 2 core di CPU, ognuno operante a una frequenza di 2.0 GHz.
- Storage: Ciascuna macchina virtuale è stata fornita con 10 gigabyte di spazio di archiviazione. Questo spazio è stato utilizzato per salvare all'interno dei volumi le metriche presenti all'interno dei servizi.

Queste specifiche sono state scelte in base ai requisiti del deployment e all'analisi delle prestazioni previste. L'obiettivo era fornire un ambiente che equilibrasse la capacità di elaborazione, la memoria e lo spazio di archiviazione per garantire un deployment efficiente e affidabile.

L'utilizzo di Proxmox come soluzione di virtualizzazione ha consentito la gestione agevole delle macchine virtuali nel cluster, facilitando la scalabilità e la manutenzione del sistema nel corso del deployment.

3.2 Kubernetes deployment

In questa sezione analizzeremo le configurazioni e le scelte di implementazione riguardo alla configurazione di FunLess utilizzando kubernetes come orchestatore.

3.2.1 Installazione

Kubernetes presenta un ecosistema ampio e complesso. La sua architettura basata su microservizi offre una vasta gamma di funzionalità avanzate. Kubernetes richiede un significativo impegno nello studio e nell'apprendimento delle sue componenti e concetti fondamentali, ma offre comunque un livello di personalizzazione e scalabilità una volta superata la curva di apprendimento.

Kubernetes richiede una procedura di installazione più articolata. A seconda dell'implementazione scelta, i passaggi di installazione possono variare significativamente. È necessario scaricare la sua commandline: kubectl e Kubeadm nel caso si voglia creare un semplice cluster. Durante l'implementazione del deployment di FunLess è stato utilizzato kubectl (vedi sezione 2.4.4) e un tool chiamato kind[2] rispettivamente per la gestione del cluster e la simulazio-

[2]Kind (Kubernetes in Docker) è una piattaforma open-source che consente di eseguire cluster Kubernetes all'interno di container Docker, semplificando lo sviluppo e il testing in ambienti locali.

ne di più nodi tramite container docker. Di seguito la relativa installazione omettendo alcune peculiarità di essa.

Listing 3.1: Installazione di kind e kubectl

```
curl -LO https://dl.k8s.io/release/v1.28.1/bin/linux/amd64/kubectl
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl


curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-amd64
chmod +x ./kind
sudo mv ./kind /usr/local/bin/kind
```

3.2.2 Core deployment

Il Core è un componente fondamentale nella piattaforma di FunLess, esso si occupa di selezionare uno degli worker per eseguire le funzioni invocate esponendo una API.
La gestione delle repliche riveste un ruolo centrale in questo contesto, in quanto stabilisce il numero desiderato di repliche del servizio, definendo un valore predefinito per ciascuna istanza del servizio. Successivamente, all'interno del template di creazione dei pod, sono specificate le etichette associate, tra cui il service account che si occupa dei requisiti di sicurezza e altre direttive rilevanti per la configurazione del pod.

Nella sezione dedicata ai container a riga 29, viene specificata l'immagine Docker da utilizzare, insieme alle porte di rete che devono essere esposte e rese accessibili. Inoltre, sono presenti all'interno del blocco env variabili d'ambiente utili per il funzionamento del servizio, come il token relativo al core di FunLess e altre credenziali correlate.

All'interno di questa configurazione, sono stati definiti i dettagli del modello per la creazione del pod del Worker, compresi i requisiti di sicurezza tramite la definizione del service account appropriato.

Infine, per quanto riguarda il manifesto del Service, vengono definiti i dettagli relativi alle porte da esporre per consentire l'accesso al servizio stesso.

Listing 3.2: Core deployment

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: fl-core
```

```
  namespace: fl
  labels:
    app: fl-core
spec:
  replicas: 1
  selector:
    matchLabels:
      app: fl-core
  template:
    metadata:
      labels:
        app: fl-core
    spec:
      serviceAccountName: fl-svc-account
      restartPolicy: "Always"
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: fl
                operator: In
                values:
                - "core"
      containers:
      - name: core
        image: "ghcr.io/funlessdev/core:latest"
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 4000
        - containerPort: 4369
        env:
        - name: SECRET_KEY_BASE
          valueFrom:
            secretKeyRef:
              name: fl-core-secret-key-base
              key: secret_key_base
        - name: PGUSER
          valueFrom:
            secretKeyRef:
              name: fl-core-secret-postgres-user
              key: secret_key_user
        - name: PGPASSWORD
          valueFrom:
            secretKeyRef:
              name: fl-core-secret-postgres-password
              key: secret_key_password
        - name: KIBANA_PASSWORD
          valueFrom:
```

```
54              secretKeyRef:
55                name: fl-core-secret-kibana-password
56                key: secret_key_password
57          - name: NODE_IP
58            valueFrom:
59              fieldRef:
60                fieldPath: status.podIP
61          - name: DEPLOY_ENV
62            value: "kubernetes"
63
64 ---
65
66 apiVersion: v1
67 kind: Service
68 metadata:
69   name: fl-core-service
70   namespace: fl
71   labels:
72     app: fl-core
73 spec:
74   type: NodePort
75   selector:
76     app: fl-core
77   ports:
78     - name: http
79       port: 4000
80       targetPort: 4000
81       nodePort: 30210
82       protocol: TCP
```

3.2.3 Worker deployment

Il Worker rappresenta un altro componente fondamentale nell'architettura di FunLess, poiché ha il compito di eseguire le funzioni assegnate dal core.
La configurazione specifica descritta a riga 2 utilizza un manifest di tipo DaemonSet. Questa scelta implica che una singola istanza di questo pod verrà automaticamente pianificata su ciascun nodo del cluster che rispetti i requisiti della risorsa, (nel nostro caso ogni nodo che sia stato etichettato con 'fl-worker').

Questo approccio assicura che il componente Worker sia distribuito in modo uniforme su tutti i nodi del cluster, consentendo una gestione efficiente delle funzioni e garantendo che il sistema FunLess sia altamente disponibile e scalabile. Di seguito viene trascritta la configurazione riguardante il worker di FunLess.

Listing 3.3: Worker deployment

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fl-worker
  namespace: fl
  labels:
    app: fl-worker
spec:
  selector:
    matchLabels:
      app: fl-worker
  template:
    metadata:
      labels:
        app: fl-worker
    spec:
      serviceAccountName: fl-svc-account
      restartPolicy: "Always"
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: fl
                operator: In
                values:
                - "worker"
      containers:
      - name: worker
        image: "ghcr.io/funlessdev/worker:latest"
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 4021
        - containerPort: 4369
        env:
        - name: NODE_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        - name: DEPLOY_ENV
          value: "kubernetes"
```

3.2.4 Postgres deployment

FunLess utilizza Postgres come database centralizzato per archiviare funzioni e moduli utilizzati dal core. Come per le altre configurazioni, anche in questo caso è fondamentale definire le variabili di ambiente necessarie per la creazione e la gestione del database. Inoltre, questa configurazione prevede l'utilizzo di uno storage persistente per garantire la persistenza dei dati.

Kubernetes gestisce il concetto di persistent volumes (PV) e persistent volume claims (PVC) per la gestione dei volumi di storage in modo efficiente. I PV rappresentano risorse di storage fisiche o basate su cloud, mentre i PVC sono le richieste di storage presentate dai pod. Questa separazione tra definizione dello storage e utilizzo effettivo favorisce una maggiore astrazione e semplifica gli aspetti operativi.

Kubernetes offre un supporto per diversi tipi di storage[10], tra cui i principali sono:

- **NFS**: che consente a un computer di accedere e condividere directory su una rete, come se fossero presenti nella sua cartella locale.
- **iSCSI**: il quale permette collegare interi dispositivi di storage, come dischi rigidi, a server e computer tramite una rete IP, consentendo l'accesso e l'utilizzo di questo storage come se fosse collegato localmente al sistema.

Questo consente una maggiore flessibilità nella scelta delle risorse di storage più adatte alle esigenze.

È importante notare che nella definizione del volume a riga 8 sono state specificate altre configurazioni, tra cui la modalità di accesso al volume, la capacità dello spazio di archiviazione e la politica di conservazione del volume. In questo caso, il volume non verrà automaticamente eliminato quando viene rimosso, garantendo la persistenza dei dati anche in situazioni in cui i PVC vengono modificati o eliminati.

Listing 3.4: Postgres deployment

```
1 apiVersion: v1
2 kind: PersistentVolume
3 metadata:
4   name: fl-postgres-pv
5   namespace: fl
6   labels:
7     app: fl-postgres
8 spec:
9   capacity:
```

```
10      storage: 2Gi
11    accessModes:
12      - ReadWriteOnce
13    persistentVolumeReclaimPolicy: Retain
14    storageClassName: standard
15    hostPath:
16      path: /data/postgres
17
18  ---
19
20  apiVersion: v1
21  kind: PersistentVolumeClaim
22  metadata:
23    name: fl-postgres-pvc
24    namespace: fl
25  spec:
26    accessModes:
27      - ReadWriteOnce
28    resources:
29      requests:
30        storage: 2Gi
31    storageClassName: standard
32
33  ---
34
35  apiVersion: apps/v1
36  kind: Deployment
37  metadata:
38    name: fl-postgres
39    namespace: fl
40    labels:
41      app: fl-postgres
42  spec:
43    replicas: 1
44    selector:
45      matchLabels:
46        app: fl-postgres
47    template:
48      metadata:
49        labels:
50          app: fl-postgres
51      spec:
52        serviceAccountName: fl-svc-account
53        restartPolicy: "Always"
54        affinity:
55          nodeAffinity:
56            requiredDuringSchedulingIgnoredDuringExecution:
57              nodeSelectorTerms:
58              - matchExpressions:
```

```
              - key: fl
                operator: In
                values:
                - "core"
      containers:
      - name: postgres
        image: "postgres:latest"
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 5432
        env:
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: fl-core-secret-postgres-password
              key: secret_key_password
        - name: POSTGRES_USER
          valueFrom:
            secretKeyRef:
              name: fl-core-secret-postgres-user
              key: secret_key_user
        - name: POSTGRES_HOST_AUTH_METHOD
          value: trust
        volumeMounts:
          - name: fl-postgres-data
            mountPath: /var/lib/postgresql/data
      volumes:
        - name: fl-postgres-data
          persistentVolumeClaim:
            claimName: fl-postgres-pvc

---

apiVersion: v1
kind: Service
metadata:
  name: postgres
  namespace: fl
  labels:
    app: fl-postgres
spec:
  type: ClusterIP
  selector:
    app: fl-postgres
  ports:
    - name: http
      port: 5432
      targetPort: 5432
      protocol: TCP
```

Postgres init

Prima di inizializzare il server Postgres per FunLess, è essenziale creare l'utente e il database dedicati. Questa operazione viene effettuata mediante l'utilizzo della direttiva command, che consente l'esecuzione di script personalizzati specificando gli argomenti necessari.

In particolare, lo script in questione a riga 31 ha la responsabilità di verificare lo stato di prontezza del server Postgres, garantendo che sia completamente inizializzato e pronto a rispondere alle richieste in arrivo. Questa verifica viene effettuata utilizzando il comando pg_isready, il quale stabilisce la connettività con il server Postgres specifico. Durante il processo di attesa, nel caso in cui il server Postgres non sia ancora pronto a gestire connessioni, lo script attende un secondo prima di tentare nuovamente la connessione. Una volta che il server è pronto, verrà creato il database dedicato a FunLess.

È importante notare inoltre che il tipo di manifesto utilizzato è di tipo Job, il quale è particolarmente adatto per eseguire operazioni di questo tipo con esecuzioni pianificate. Assicurando che l'operazione di creazione del database e dell'utente avvenga in modo controllato e gestito, garantendo l'integrità e la coerenza del sistema durante il processo di inizializzazione di Postgres.

Listing 3.5: init-postgres deployment

```
1  apiVersion: batch/v1
2  kind: Job
3  metadata:
4    name: fl-init-postgres
5    namespace: fl
6    labels:
7      app: fl-init-postgres
8  spec:
9    backoffLimit: 3
10   template:
11     metadata:
12       name: fl-init-postgres
13       labels:
14         app: fl-init-postgres
15     spec:
16       serviceAccountName: fl-svc-account
17       restartPolicy: "Never"
18       affinity:
19         nodeAffinity:
20           requiredDuringSchedulingIgnoredDuringExecution:
21             nodeSelectorTerms:
22             - matchExpressions:
23               - key: fl
```

```
24                operator: In
25                values:
26                - "core"
27       containers:
28       - name: init-postgres
29         image: "postgres:latest"
30         imagePullPolicy: IfNotPresent
31         command:  ["/bin/bash", "-c", "while ! pg_isready -h                postgres -U {
     POSTGRES_USER}; do sleep 1; done; psql -h postgres -U {POSTGRES_USER} -c 'CREATE DATABASE
     FunLess;'"]
32         env:
33         - name: POSTGRES_PASSWORD
34           valueFrom:
35             secretKeyRef:
36               name: fl-core-secret-postgres-password
37               key: secret_key_password
38         - name: POSTGRES_USER
39           valueFrom:
40             secretKeyRef:
41               name: fl-core-secret-postgres-user
42               key: secret_key_user
43         - name: POSTGRES_HOST_AUTH_METHOD
44           value: trust
```

3.2.5 Prometheus deployment

Prometheus svolge un altro ruolo non indifferente all'interno di questa configurazione, poiché si occupa di raccogliere metriche provenienti dalla piattaforma. Inoltre, il Core utilizza Prometheus per accedere alle informazioni di stato degli Worker, che sono fondamentali per prendere le decisioni di scheduling.

Nella presente configurazione, oltre all'impiego di storage persistenti, è possibile notare l'utilizzo del blocco initContainers a riga 62. Questo blocco consente l'esecuzione di un servizio prima dell'avvio effettivo del container principale. In particolare, in questo contesto specifico, è stato creato un initContainer per preparare la directory destinata ai dati di Prometheus prima dell'avvio del container principale.

L'utilizzo di initContainers permette di gestire le operazioni di inizializzazione in modo ordinato, garantendo che le risorse e le dipendenze necessarie siano pronte prima che il container principale venga avviato. Questa pratica applicata è utile in scenari in cui è necessario eseguire operazioni preliminari, come in questo caso la preparazione di directory o la configurazione di servizi,

prima che il servizio principale sia avviato. In questo modo, si assicura che Prometheus sia in grado di raccogliere dati.

Listing 3.6: Prometheus deployment

```
1
2  apiVersion: v1
3  kind: PersistentVolume
4  metadata:
5    name: fl-prometheus-pv
6    namespace: fl
7  spec:
8    capacity:
9      storage: 2Gi
10   accessModes:
11     - ReadWriteOnce
12   persistentVolumeReclaimPolicy: Retain
13   storageClassName: standard
14   hostPath:
15     path: /data/prometheus
16
17 ---
18
19 apiVersion: v1
20 kind: PersistentVolumeClaim
21 metadata:
22   name: fl-prometheus-pvc
23   namespace: fl
24 spec:
25   accessModes:
26     - ReadWriteOnce
27   resources:
28     requests:
29       storage: 2Gi
30   storageClassName: standard
31
32 ---
33
34 apiVersion: apps/v1
35 kind: Deployment
36 metadata:
37   name: fl-prometheus
38   namespace: fl
39   labels:
40     app: fl-prometheus
41 spec:
42   replicas: 1
43   selector:
44     matchLabels:
45       app: fl-prometheus
```

```
  template:
    metadata:
      labels:
        app: fl-prometheus
    spec:
      serviceAccountName: fl-svc-account
      restartPolicy: "Always"
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                - key: fl
                  operator: In
                  values:
                    - "core"
      initContainers:
        - name: create-data-dir
          image: busybox
          command: ["mkdir", "-p", "/data/prometheus"]
          volumeMounts:
            - name: fl-prometheus-data
              mountPath: /data/prometheus
      volumes:
        - name: fl-prometheus-data
          persistentVolumeClaim:
            claimName: fl-prometheus-pvc
      containers:
        - name: prometheus
          image: prom/prometheus:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 9090
          volumeMounts:
            - name: fl-prometheus-data
              mountPath: /prometheus/data
---
apiVersion: v1
kind: Service
metadata:
  name: prometheus
  namespace: fl
  labels:
    app: fl-prometheus
spec:
  type: ClusterIP
  selector:
    app: fl-prometheus
  ports:
```

```
95      - name: http
96        port: 9090
97        targetPort: 9090
98        protocol: TCP
```

3.2.6 Elasticsearch deployment

Listing 3.7: Elasticsearch deployment

```
1
2  apiVersion: v1
3  kind: PersistentVolume
4  metadata:
5    name: fl-elasticsearch-pv
6    namespace: fl
7  spec:
8    capacity:
9      storage: 2Gi
10   accessModes:
11     - ReadWriteOnce
12   persistentVolumeReclaimPolicy: Retain
13   storageClassName: standard
14   hostPath:
15     path: /data/elasticsearch
16
17 ---
18
19 apiVersion: v1
20 kind: PersistentVolumeClaim
21 metadata:
22   name: fl-elasticsearch-pvc
23   namespace: fl
24 spec:
25   accessModes:
26     - ReadWriteOnce
27   resources:
28     requests:
29       storage: 2Gi
30   storageClassName: standard
31
32 ---
33
34 apiVersion: apps/v1
35 kind: Deployment
36 metadata:
37   name: fl-elasticsearch
38   namespace: fl
39 spec:
```

```
  replicas: 1
  selector:
    matchLabels:
      app: fl-elasticsearch
  template:
    metadata:
      labels:
        app: fl-elasticsearch
    spec:
      serviceAccountName: fl-svc-account
      restartPolicy: "Always"
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: fl
                operator: In
                values:
                - "core"
      containers:
        - name: elasticsearch
          image: docker.elastic.co/elasticsearch/elasticsearch:8.8.0
          ports:
            - containerPort: 9200
          env:
            - name: discovery.type
              value: single-node
            - name: xpack.security.enabled
              value: "true"
        volumeMounts:
          - name: fl-elasticsearch-data
            mountPath: /var/lib/elasticsearch/data
      volumes:
        - name: fl-elasticsearch-data
          persistentVolumeClaim:
            claimName: fl-elasticsearch-pvc
```

3.2.7 Kibana deployment

Listing 3.8: Kibana deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fl-kibana
  namespace: fl
spec:
```

```
  replicas: 1
  selector:
    matchLabels:
      app: fl-kibana
  template:
    metadata:
      labels:
        app: fl-kibana
    spec:
      serviceAccountName: fl-svc-account
      restartPolicy: "Always"
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: fl
                operator: In
                values:
                - "core"
      containers:
        - name: kibana
          image: docker.elastic.co/kibana/kibana:8.8.0
          ports:
            - containerPort: 5601
          env:
            - name: ELASTICSEARCH_HOSTS
              value: "http://10-244-1-18.default.pod.cluster.local:9200" # set elastic ip
            - name: XPACK_SECURITY_ENABLED
              value: "true"
            - name: ELASTICSEARCH_USERNAME
              value: "kibana_system"
            - name: ELASTICSEARCH_PASSWORD
              value: "aRgY7Odesha8qHfvRzWB"
              # value: {KIBANA_PASSWORD}
            - name: XPACK_REPORTING_ROLES_ENABLED
              value: "false"
            - name: XPACK_FLEET_REGISTRYURL
              value: "https://epr.elastic.co"
            - name: XPACK_FLEET_AGENTS_ENABLED
              value: "true"
            - name: xpack.encryptedSavedObjects.encryptionKey
              value: "true"
---
apiVersion: v1
kind: Service
metadata:
  name: fl-kibana
  namespace: fl
```

```
  labels:
    app: fl-kibana
spec:
  ports:
    - name: http
      port: 5601
      targetPort: 5601
  selector:
    app: fl-kibana
```

3.2.8 Filebeat deployment

Listing 3.9: Filebeat deployment

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fl-filebeat
  namespace: fl
spec:
  selector:
    matchLabels:
      app: fl-filebeat
  template:
    metadata:
      labels:
        app: fl-filebeat
    spec:
      serviceAccountName: fl-svc-account
      restartPolicy: "Always"
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: fl
                operator: In
                values:
                - "core"
      containers:
      - name: filebeat
        image: docker.elastic.co/beats/filebeat:8.8.0
        args: ["--strict.perms=false"]
        securityContext:
          runAsUser: 0
        volumeMounts:
        - name: config
          mountPath: /usr/share/filebeat/filebeat.yml
```

```
35          subPath: filebeat.yml
36          readOnly: true
37        - name: varlogcontainers
38          mountPath: /var/log/containers
39          readOnly: true
40      volumes:
41      - name: config
42        configMap:
43          name: fl-filebeat-config
44      - name: varlogcontainers
45        hostPath:
46          path: /var/log/containers
```

3.2.9 Gestione dei secret

Kubernetes mette a disposizione risorse come le ConfigMap per gestire configurazioni e dati sensibili all'interno del cluster. Queste tipologie di manifesto consentono di memorizzare configurazioni e informazioni critiche in modo centralizzato, rendendole facilmente accessibili ai container all'interno dei pod. Questa caratteristica elimina la necessità di integrazioni separate, come Vault o Consul, di cui parleremo in seguito.

Durante lo sviluppo del nostro deployment, le ConfigMap si sono rivelate fondamentali. Hanno permesso di gestire i dati sensibili, come ad esempio la password per Postgres e il token del servizio core di FunLess. Grazie all'utilizzo di ConfigMap, è stato possibile separare la gestione delle configurazioni dal codice dell'applicazione, migliorando la sicurezza e la gestibilità complessiva del sistema.

Listing 3.10: Configurazione di una ConfigMap per la gestione di variabili d'ambiente

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: fl-core
5    spec:
6        containers:
7        - name: core
8          image: "ghcr.io/funLessdev/core:latest"
9          ports:
10         - containerPort: 4000
11         - containerPort: 4369
12         env:
13         - name: SECRET_KEY_BASE
14           valueFrom:
15             secretKeyRef:
```

```
16              name: fl-core-secret-key-base
17              key: secret_key_base
18
19 ---
20
21 apiVersion: v1
22 kind: Secret
23 metadata:
24   name: fl-core-secret-key-base
25 type: Opaque
26 data:
27   secret_key_base: c2VjcmV0X3NlY3JldF9rZXk=
```

3.3 Nomad deployment

In questa sezione analizzeremo invece l'installazione di nomad, le configurazioni e le scelte di implementazione riguardo l'ELK stack formato da Elasticsearch, Filebeat e Kibana utilizzando nomad come orchestatore.

3.3.1 Installazione

Nomad offre un insieme essenziale di funzionalità per il deployment e la gestione delle applicazioni containerizzate, consentendo agli utenti di aggiungere ulteriori capacità solo se necessario. Esso si distingue per la sua semplicità, offrendo una configurazione accessibile e user-friendly che semplifica notevolmente la gestione delle applicazioni e dei servizi.

La sua installazione è semplice e veloce. Essendo un prodotto single-binary, richiede meno passaggi per essere installato e può essere facilmente configurato tramite file di configurazione. Questa caratteristica lo rende particolarmente adatto per deployment rapidi e situazioni in cui la complessità deve essere minimizzata. Per iniziare a utilizzarlo è solamente necessario aggiungere la chiave GPG[3] di Hashicorp e successivamente installare nomad attraverso un package manager.

Listing 3.11: Installazione di Nomad

```
sudo apt-get install nomad
```

[3]GPG è una chiave crittografica utilizzata per crittografare e decrittografare dati, garantendo l'integrità di essi

3.3.2 Configurazione degli agenti

Nomad presenta un file di configurazione, il quale permette la gestione dell'host. Durante lo sviluppo del deployment è stato necessario adattare il file di configurazione specificando in particolare su ogni host:

- **Tipologia di agent**: il quale indica se l'host in questione deve agire da client o server.
- **Plugin**: per utilizzare plugin specifici all'interno di nomad bisogna abilitarli. In particolare nel nostro caso è stato necessario abilitare il plugin di Docker (vedi listing 3.12).
- **Volumi**: anch'essi possono essere integrati all'interno del file di configurazione, specificando il path.

Di seguito vengono riportati due esempi delle configurazione utilizzate per gli host all'interno del deployment di FunLess.

Listing 3.12: Configurazione client di Nomad per il deployment di FunLess

```
client {
  enabled = true
}

host_volume "elasticsearch-volume" {
  path      = "/usr/share/postgres/data"
  read_only = true
}

plugin "docker" {
  config {
    enabled = true
  }
}
```

Listing 3.13: configurazione server di Nomad per il deployment di FunLess

```
server {
    enabled = true
    boostrap_expect = 3 # numero di host previsti
}
```

3.3.3 Elasticsearch deployment

Elasticsearch è un sistema di analisi e archiviazione dei dati specializzato nell'indicizzazione, ricerca e analisi di grandi quantità di dati, nel nostro caso

verranno salvati tutti i relativi log di sistema e metriche.

Nella configurazione fornita, sono evidenziate alcune caratteristiche chiave. In particolare, si presta attenzione alla gestione delle variabili di ambiente di Elasticsearch a riga 15 e alla modalità di rete impostata su bridge a riga 7. In questa modalità, il container opera in una rete isolata, separata dalla rete dell'host, ma può ancora comunicare con altre risorse nella rete dei container. Questo consente il necessario scambio di dati tra Elasticsearch e gli altri servizi presenti nel sistema.

È importante notare anche la flessibilità della configurazione di Nomad presente nel blocco di resources a riga 34 il quale permette di gestire le risorse assegnate al container, in particolare specificando la quantità di CPU in megahertz e memoria in megabyte riservata all'interno del blocco container. Questo aspetto è fondamentale per garantire che Elasticsearch abbia le risorse necessarie per eseguire le operazioni di indicizzazione e ricerca in modo efficiente.

Listing 3.14: Elasticsearch deployment

```
1  job "elasticsearch" {
2    datacenters = ["dc1"]
3
4    group "efk-group" {
5      count = 1
6
7      network {
8        mode = "bridge"
9        port "elastic" {}
10     }
11
12     task "elastic" {
13       driver = "docker"
14
15       env = {
16         "ES_JAVA_OPTS"           = "-Xms512m -Xmx512m"
17         "ELASTIC_USER"           = "elastic"
18         "ELASTIC_PASSWORD"       = "elasticpassword"
19         "discovery.type"         = "single-node"
20         "bootstrap.memory_lock"  = "true"
21         "xpack.license.self_generated.type" = "basic"
22         "xpack.security.enabled" = "true"
23         "xpack.security.authc.api_key.enabled" = "true"
24       }
25
26       config {
27         image = "docker.elastic.co/elasticsearch/elasticsearch:8.8.0"
28         ports = ["elastic"]
29         volumes = [
```

```
30          "local/elasticsearch:/etc/elasticsearch/",
31        ]
32      }
33
34      resources {
35        cpu    = 2000
36        memory = 1000
37      }
38
39       service {
40         tags = ["elasticsearch"]
41         port = "elastic"
42         provider = "consul"
43       }
44
45    }
46  }
47 }
```

3.3.4 Kibana deployment

Durante il processo di deployment di Kibana, si è verificato un inconveniente relativo all'autenticazione come utente root. Di conseguenza, è stata necessaria l'impostazione della password per un utente specifico, attraverso una semplice richiesta POST all'API del servizio. Questa operazione è stata gestita all'interno dello stesso gruppo di configurazione di Kibana, in particolare alla riga 24.

È importante sottolineare la presenza del blocco lifecycle alla riga 14. Tale blocco indica che il servizio di Kibana sarà riavviato finché la richiesta di impostazione della password non verrà completata con successo. Questo approccio garantisce che l'autenticazione dell'utente avvenga correttamente prima di consentire l'accesso a Kibana.

Listing 3.15: Kibana deployment

```
1  job "kibana-service" {
2    datacenters = ["dc1"]
3    type        = "service"
4
5    group "efk-group" {
6      count = 1
7      network {
8          mode = "bridge"
9          port "kibana" {
10           static = 5601
```

```
11        }
12      }
13    task "curl" {
14      lifecycle {
15        hook = "prestart"
16        sidecar = false
17      }
18
19      template {
20        change_mode = "noop"
21        destination = "/local/request.sh"
22
23        data = <<EOH
24         curl -X POST "172.17.0.1:9200/_security/user/SET_USER?pretty" -H          'Content-
     Type: application/json' -d'
25              {
26                "password" : "SET_PASSWORD",
27                "roles" : ["superuser", "kibana_system"],
28                "full_name" : "example",
29                "email" : "example@gmail.com",
30                "metadata" : {
31                  "intelligence" : 7
32                }
33              }
34              ' -u elastic:elasticpassword
35        EOH
36      }
37
38      driver = "docker"
39
40      config {
41        image = "docker.elastic.co/kibana/kibana:8.8.0"
42        ports = ["kibana"]
43        command = "sh"
44        args = ["/local/request.sh"]
45      }
46
47        env {
48          ELASTICSEARCH_HOSTS    = "http://172.17.0.1:9200"
49          ELASTICSEARCH_USERNAME = "SET_USER"
50          ELASTICSEARCH_PASSWORD = "SET_PASSWORD"
51       }
52
53      resources {
54        cpu = 2000
55        memory = 2000
56      }
57    }
58
```

```
59    task "kibana" {
60
61      driver = "docker"
62
63      config {
64        image = "docker.elastic.co/kibana/kibana:8.8.0"
65        ports = ["kibana"]
66      }
67        env {
68          ELASTICSEARCH_HOSTS    = "http://172.17.0.1:9200"
69          ELASTICSEARCH_USERNAME = "SET_USER"
70          ELASTICSEARCH_PASSWORD = "SET_PASSWORD"
71       }
72      resources {
73        cpu = 2000
74        memory = 2000
75      }
76
77       service {
78         tags = ["kibana", "elk"]
79         port = "kibana"
80         provider = "consul"
81       }
82
83    }
84  }
85 }
```

3.3.5 Filebeat deployment

Filebeat è un altro componente chiave utilizzato per la raccolta e l'inoltro dei log. La parte più significativa della configurazione riguarda la connessione di Filebeat a Elasticsearch, il database di archiviazione e analisi dei dati di log. All'interno del file di configurazione a riga 15, è definito il comportamento di Filebeat, specificando il target di destinazione dei dati raccolti, ovvero Elasticsearch, all'indirizzo 172.17.0.1 e porta 9200. Questo significa che Filebeat invierà direttamente i log raccolti a un'istanza Elasticsearch in ascolto su quell'indirizzo e porta. Questa configurazione è essenziale per l'archiviazione dei log in Elasticsearch, facilitando le successive analisi e visualizzazioni. L'host e la porta specificati in output.elasticsearch devono essere configurati correttamente per consentire una connessione funzionante tra i due servizi.

Listing 3.16: Filebeat deployment

```
1 job "filebeat-service" {
```

```
  datacenters = ["dc1"]
  type        = "service"

  group "efk-group" {
    count = 1

    task "filebeat" {
      driver = "docker"

      config {
        image = "docker.elastic.co/beats/filebeat:8.8.0"
      }

      template {
        data = <<EOT
            filebeat.inputs:
            - type: log
              enabled: true
              paths:
                - /logs/file.log
        output.elasticsearch:
          hosts: ["172.17.0.1:9200"]
        EOT
        destination = "/local/config"
      }

      resources {
        cpu    = 500
        memory = 256
      }
    }
  }
}
```

3.3.6 Core deployment

Listing 3.17: Core deployment

```
job "core-service" {
  datacenters = ["dc1"]
  type        = "service"

  group "core" {
    count = 1

    network {
        mode = "host"
        port "funless" {
```

```
11          static = 4000
12        }
13      }
14
15    task "core" {
16      driver = "docker"
17
18      config {
19        image = "ghcr.io/funlessdev/core:latest"
20        ports = ["funless"]
21      }
22
23      env {
24        PGHOST            = "postgres-postgres-postgres"
25        PGUSER            = "postgresuser"
26        PGPASSWORD        = "postgrespassword"
27        PGDATABASE        = "funless"
28        PGPORT            = "5432"
29        SECRET_KEY_BASE   = "agoodexampleofasecretkey"
30      }
31
32      resources {
33        cpu    = 1000
34        memory = 2000
35      }
36
37       service {
38         tags = ["funless"]
39         port = "funless"
40         provider = "consul"
41       }
42
43
44    }
45  }
46 }
```

3.3.7 Worker deployment

Listing 3.18: Worker deployment

```
1 job "fl-worker" {
2   datacenters = ["dc1"]
3   type        = "system"
4
5   group "fl-worker" {
6     count = 1
7
```

```
    network {
      mode = "host"
      port "funless" {
        static = 4021
      }
      port "erlang-epmd" {
        static = 4369
      }
    }

    task "worker" {
      driver = "docker"

      config {
        image = "ghcr.io/funlessdev/worker:latest"
        ports = [
          "funless",
          "erlang-epmd"
        ]
      }

      env {
        NODE_IP    = "{attr.unique.network.ip-address}"
      }

      resources {
        cpu    = 500
        memory = 2000
      }
    }
  }
}
```

3.3.8 Postgres deployment

Listing 3.19: Postgres deployment

```
job "postgres" {
  region = "global"
  datacenters = ["dc1"]
  type = "service"
  #  name = "postgres"

  group "postgres" {
    count = 1

    network {
        mode = "host"
```

```
12        port "db" {
13          static = 5432
14        }
15      }
16    task "postgres" {
17
18      driver = "docker"
19      config {
20        image = "postgres"
21        ports = ["db"]
22      }
23
24      env {
25          POSTGRES_USER="postgresuser"
26          POSTGRES_PASSWORD="postgrespassword"
27          POSTGRES_DB="funless"
28      }
29
30      resources {
31        cpu = 200
32        memory = 200
33      }
34
35       service {
36         tags = ["postgres"]
37         port = "db"
38         provider = "consul"
39       }
40    }
41  }
42 }
```

3.3.9 Prometheus deployment

Listing 3.20: Prometheus deployment

```
1  job "prometheus" {
2    datacenters = ["dc1"]
3    type        = "service"
4
5    group "monitoring" {
6      count = 1
7
8      network {
9        port "prometheus_ui" {
10         static = 9090
11       }
12     }
```

```
13
14      task "prometheus" {
15        template {
16          change_mode = "noop"
17          destination = "local/prometheus.yml"
18
19          data = <<EOH
20  ---
21  global:
22    scrape_interval: 5s
23    evaluation_interval: 5s
24
25    external_labels:
26      monitor: "funless"
27
28  # Targets to scrape
29  scrape_configs:
30    - job_name: "prometheus"
31      static_configs:
32        - targets: ["localhost:9090"]
33    - job_name: "funless"
34      static_configs:
35        - targets: ["core:4000", "worker:4021"]
36  EOH
37        }
38
39        driver = "docker"
40
41        config {
42          image = "prom/prometheus:latest"
43
44          volumes = [
45            "local/prometheus.yml:/etc/prometheus/prometheus.yml",
46          ]
47
48          ports = ["prometheus_ui"]
49        }
50
51         service {
52           tags = ["prometheus"]
53           port = "kibana"
54           provider = "consul"
55         }
56
57      }
58    }
59  }
```

3.3.10 Integrazioni Architetturali di Nomad

Una delle considerazioni cruciali, riguarda le integrazioni architetturali che ampliano le capacità di gestione, sicurezza e rete delle piattaforme di orchestrazione. Tali integrazioni possono fornire un valore aggiunto significativo, contribuendo all'efficienza operativa, alla scalabilità e all'affidabilità delle applicazioni distribuite.

Nomad si distingue per la sua modularità e la capacità di integrarsi con tecnologie complementari. Tra queste, spiccano le integrazioni con Consul, Vault, fabio e altre soluzioni HashiCorp che arricchiscono le funzionalità di orchestrazione e sicurezza.

- **Consul**: L'integrazione di Nomad con Consul offre notevoli vantaggi. La possibilità di registrare dinamicamente i job Nomad come servizi semplifica la scoperta di essi, promuovendo una comunicazione efficiente tra le parti. Questa integrazione facilita la gestione e il monitoraggio dei servizi, contribuendo a migliorare l'affidabilità dell'architettura.

- **Vault**: L'integrazione con Vault [3] consente a Nomad di gestire in modo sicuro le informazioni sensibili, come le credenziali e le chiavi di autenticazione. Vault, attraverso la sua gestione dinamica dei segreti, aggiunge uno strato di sicurezza fondamentale alla gestione dei job Nomad, preservando l'integrità delle informazioni sensibili.

- **Fabio**: rappresenta un load balancer HTTP/HTTPS basato su Consul. Questo strumento è stato progettato per operare in modo complementare con Nomad e Consul, fornendo un meccanismo di bilanciamento del carico intelligente e resiliente. Nell'ambito dell'orchestrazione dei microservizi tramite Nomad, Fabio[1] agisce da gateway per la comunicazione tra i servizi, contribuendo all'ottimizzazione del traffico e alla scalabilità delle applicazioni.

Durante il processo di deployment di FunLess, è stata inclusa l'integrazione di Consul, il quale ha introdotto un vantaggio significativo fornendo un'ampia panoramica dello stato dei servizi attivi e semplificando notevolmente la scoperta di questi servizi all'interno dell'ambiente. In aggiunta, Consul offre un KeyValue Store che consente di memorizzare configurazioni, inclusi token segreti, in modo sicuro e accessibile.

L'installazione di Consul è un processo diretto, richiedendo solo l'installazione di un pacchetto.

```
sudo apt install consul
```

Listing 3.21: installazione di consul

Per registrare un servizio all'interno di Consul, facciamo uso del blocco service, il quale è una componente costante nelle configurazioni precedenti. Questo approccio ci consente di raggruppare i servizi utilizzando tag e semplifica l'integrazione di nuovi servizi. Inoltre, è importante notare che Consul può essere facilmente integrato anche con Kubernetes, consentendo una gestione unificata e centralizzata dei servizi, indipendentemente dall'ambiente di orchestrazione utilizzato. Di seguito un esempio di come è stata configurata l'integrazione di Consul in un job Nomad:

Listing 3.22: Integrazione di consul in nomad

```
1 service {
2     tags = ["postgres"]
3     provider = "consul"
4 }
```

3.4 Benchmark a confronto

In questa sezione, esamineremo innanzitutto il tempo necessario per mettere in esecuzione l'infrastruttura di FunLess. Successivamente, procederemo all'analisi dei risultati ottenuti dai benchmark riguardanti l'esecuzione di una funzione, utilizzando sia Kubernetes che Nomad. Si noti che utilizzando un ambiente non isolato i tempi potrebbero essere variabili a seconda delle capacità computazionali delle macchine.

Come precedentemente mostrato, l'infrastruttura presente all'interno di FunLess presenta diversi componenti, dunque per portare in esecuzione ogni componente è necessario attendere un tempo discreto. Per effettuare questo test è stato fatto uso di uno script bash, il quale utilizzando il comando time e relativi controlli per verificare che i container siano stati avviati ci restituisce il tempo totale.

Per quando riguarda il deployment di Kubernetes, esso è stato completato in due minuti e venti secondi. Questo tempo più lungo potrebbe essere attribuito a diversi fattori, tra cui la complessità del processo di provisioning delle risorse, e il coordinamento tra i diversi componenti all'interno di Kubernetes. D'altra parte, il deployment su Nomad è stato completato in un minuto e trentadue

secondi, risultando più rapido. Questo efficienza è dovuta alla sua semplicità architetturale.

```
./calculate_k.sh  140.4s    system 76%  # Kubernetes
./calculate_n.sh  92.1s     system 38%  # Nomad
```

Listing 3.23: Tempo di esecuzione del deployment

Per quanto riguarda i test relativi all'esecuzione delle funzioni utilizzando FunLess, è importante notare che i benchmark sono stati condotti utilizzando Vegeta[17]. Questo strumento è stato impiegato per effettuare le richieste e valutare le prestazioni di ciascun orchestratore.

Vegeta è un software open-source utilizzato per testare le prestazioni di servizi web. È ampiamente utilizzato nell'ambito della valutazione delle prestazioni e dei test di carico. Esso è scritto in Go e offre una serie di funzionalità avanzate per la generazione di richieste HTTP in modo concorrente e il monitoraggio delle risposte.
Di seguito sono presenti le tabelle riassuntive, dove sono stati eseguiti dei test eseguendo una funzione al secondo per cento secondi.

Tabella 3.1: Risultati di Nomad

Requests	[total, rate, throughput]	100, 1.01, 1.01
Duration	[total, attack, wait]	1m39s, 1m39s, 10.349ms
Latencies	[min, mean, 50, 95]	5.789ms, 8.792ms, 6.639ms, 15.216ms
Bytes In	[total, mean]	3400, 34.00
Bytes Out	[total, mean]	3200, 32.00
Success	[ratio]	100.00%
Status Codes	[code:count]	200:100

Tabella 3.2: Risultati di Kubernetes

Requests	[total, rate, throughput]	100, 1.01, 1.01
Duration	[total, attack, wait]	1m39s, 1m39s, 6.349ms
Latencies	[min, mean, 50, 95]	5.997ms, 7.792ms, 6.774ms, 15.066ms
Bytes In	[total, mean]	3400, 34.00
Bytes Out	[total, mean]	3200, 32.00
Success	[ratio]	100.00%
Status Codes	[code:count]	200:100

Analizzando i due orchestratori, si può notare che l'overhead introdotto tra le richieste è molto simile ed è praticamente trascurabile. Il valore più interessante riguarda la latenza media, la quale rappresenta il tempo medio che è trascorso tra l'invio di una richiesta e il ricevimento della risposta da parte dell'orchestratore. La latenza varia di un millisecondo e durante tutte e cento le esecuzioni i valori sono rimasti linearmente costanti.

Nel complesso, dopo numerose sperimentazioni relative all'utilizzo dei due orchestratori è stato possibile osservare che entrambe le piattaforme presentano notevoli funzionalità, e nonostante esse abbiano filosofie completamente differenti operano nello stesso modo.
In termini di velocità, Nomad si è dimostrato chiaramente superiore, con tempi di deployment più brevi. Tuttavia, l'esperienza complessiva di utilizzo non raggiunge il livello di Kubernetes. Nonostante l'ecosistema di Nomad si basi principalmente sulla sua facilità di utilizzo, la mancanza di documentazione ha reso il tutto molto più complicato, aumentando la difficoltà nel suo apprendimento.

Inoltre, per quando riguarda il deployment di FunLess potrebbe essere più adatta la sola integrazione di Nomad, poiché la piattaforma in questione non necessità di configurazioni particolari e non è adatta alla complessità che può offrire Kubernetes.

4 Conclusioni

In questo studio, abbiamo esplorato i concetti e le implementazioni legate all'orchestrazione dei microservizi attraverso le piattaforme Nomad e Kubernetes. Il nostro obiettivo principale era comprendere le complessità e le opportunità di utilizzo di questi ambienti, sia dal punto di vista architetturale che operativo prendendo come caso di studio il deployment di FunLess.

Dal nostro esame dei prerequisiti nell'ambito dei microservizi e del paradigma serverless, all'analisi delle caratteristiche chiave di Nomad e Kubernetes, abbiamo acquisito una visione delle soluzioni che emergono nell'implementazione dei sistemi e nel contempo, abbiamo anche esaminato gli aspetti di integrazione aggiungibili all'interno della piattaforma.

L'approfondimento delle configurazioni attraverso il deployment di FunLess sia su Nomad che su Kubernetes ci ha consentito di ottenere una visione completa di entrambi gli orchestratori. Questo processo di valutazione ci ha aiutato a osservare le configurazioni mettendole a confronto.

Infine sono state anche esaminate le prestazioni relative alle implementazioni nelle diverse configurazioni, evidenziandone una differenza di tempistiche in termini di deployment, utilizzando FunLess come caso di studio. Si faccia presente che i relativi test effettuati possono variare a seconda della potenza di calcolo delle macchine.

4.1 Sviluppi futuri

Per quanto riguarda gli sviluppi futuri, è possibile identificare alcune possibili idee per la ricerca, di seguito sono presenti alcuni possibili progetti.

4.1.1 Da docker compose a nomad

Un utile sviluppo futuro nell'ambito dell'orchestrazione dei microservizi riguarda la creazione di un software in grado di convertire configurazioni Docker

Compose in formati compatibili con Nomad. Tale iniziativa semplificherebbe la migrazione da un ambiente basato su Docker Compose a uno basato su Nomad, consentendo alle organizzazioni di beneficiare delle funzionalità offerte da Nomad senza dover riprogettare interamente le proprie configurazioni.

Per realizzare questa idea, sarebbe possibile considerare un processo che coinvolge la traduzione degli attributi, delle opzioni e delle istruzioni specifiche di Docker Compose in corrispondenti equivalenti di Nomad. Il software potrebbe esaminare le definizioni dei servizi, dei volumi, delle reti e di altri componenti all'interno dei file Docker Compose, per poi generare un file di configurazione compatibile con Nomad.

Una possibile implementazione potrebbe essere guidata da alcuni passaggi chiave:

1. Analisi delle Definizioni di Docker Compose: Il software dovrebbe analizzare i file Docker Compose e identificare le definizioni dei servizi, dei volumi e delle reti. Questo processo richiederebbe la comprensione dei campi, delle opzioni e delle interazioni specifiche di Docker Compose.

2. Mappatura delle Funzionalità: Il software dovrebbe stabilire una mappatura tra le funzionalità di Docker Compose e le equivalenti funzionalità offerte da Nomad. Ad esempio, i servizi definiti in Docker Compose dovrebbero essere tradotti in job di Nomad, mentre le reti e i volumi potrebbero essere mappati alle risorse corrispondenti di Nomad.

3. Generazione del File di Configurazione di Nomad: Utilizzando la mappatura precedentemente definita, il software dovrebbe generare un file di configurazione di Nomad che rifletta accuratamente la struttura e le caratteristiche dei servizi definiti in Docker Compose.

4. Personalizzazioni e Ottimizzazioni: In alcuni casi, potrebbe essere necessario apportare personalizzazioni o ottimizzazioni al file di configurazione generato per garantire che si adatti adeguatamente alle caratteristiche di Nomad.

Un possibile approccio grafico alla struttura di questo processo che illustra le fasi e le relazioni potrebbe apparire in questo modo:

Figura 4.1: Diagramma del Processo di Conversione

In questo modo, l'implementazione di questo software di conversione da Docker Compose a Nomad semplificherebbe questa transizione, migliorando l'efficienza e la sua adozione.

4.1.2 Docker Swarm

Un ulteriore sviluppo futuro nell'ambito dell'orchestrazione dei microservizi consiste nell'esplorazione dell'implementazione del deployment di FunLess attraverso Docker Swarm. Questo progetto si pone l'obiettivo di analizzare come una piattaforma di orchestrazione integrata come Docker Swarm, possa essere adottata e confrontata con le piattaforme Nomad e Kubernetes già esaminate, considerando le loro caratteristiche, vantaggi e limitazioni.

L'implementazione del deployment tramite Docker Swarm presenterebbe un approccio diverso rispetto a Nomad e Kubernetes. Docker Swarm offre un'esperienza di orchestrazione più integrata, utilizzando concetti familiari di Docker.

Bibliografia

1. Frank Schröder. *Fabio*. 2023. URL: https://fabiolb.net/.
2. HashiCorp team. *HashiCorp: Infrastructure enables innovation*. 2023. URL: https://www.hashicorp.com/ (visitato il 05/09/2023).
3. Hashicorp Team. *Vault by HashiCorp*. 2023. URL: https://www.vaultproject.io/.
4. Hashicorp Team. *What is Consul? - HashiCorp Developer*. 2023. URL: https://developer.hashicorp.com/consul/docs/intro.
5. R. Kamal e S. Agrawal. «A design framework of Orchestrator for computing systems», 2010. DOI: 10.1109/CISIM.2010.5643506. URL: https://ieeexplore.ieee.org/document/5643506.
6. Kubernetes Team. *Command line tool (kubectl) - Kubernetes*. 2023. URL: https://kubernetes.io/docs/reference/kubectl/.
7. Kubernetes Team. *Considerations for large clusters - Kubernetes*. 2023. URL: https://kubernetes.io/docs/setup/best-practices/cluster-large/#:~:text=More%20specifically%2C%20Kubernetes%20is%20designed,more%20than%20150%2C000%20total%20pods.
8. Kubernetes Team. *Kubeadm - Kubernetes*. 2023. URL: https://kubernetes.io/docs/reference/setup-tools/kubeadm/.
9. Kubernetes team. *Kubernetes*. 2023. URL: https://kubernetes.io/ (visitato il 06/09/2023).
10. Kubernetes team. *Persistent Volumes*. 2023. URL: https://kubernetes.io/docs/concepts/storage/volumes/.
11. M. Luksa. *Kubernetes in Action*. Manning Publications, 2018.
12. Martin Fowler. *Paxos*. 2023. URL: https://martinfowler.com/articles/patterns-of-distributed-systems/paxos.html.
13. Matteo Trentin, Giuseppe De Palma. *funLess Doc Homepage*. 2023. URL: https://github.com/funLessdev/funLess (visitato il 01/09/2023).
14. Matteo Trentin, Giuseppe De Palma. *funLess Quick Start*. 2023. URL: https://funless.dev/docs/getting-started/quick-start/ (visitato il 23/09/2023).
15. Nomad team. *Nomad by HashiCorp*. 2023. URL: https://www.nomadproject.io/ (visitato il 20/09/2023).

16. D. Ongaro e J. Ousterhout. «In Search of an Understandable Consensus Algorithm», 2014, 305–320. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.
17. Tomás Senart. *Vegeta*. 2023. URL: https://github.com/tsenart/vegeta.
18. Wikipedia contributors. *Gossip protocol — Wikipedia, The Free Encyclopedia*. 2023. URL: https://en.wikipedia.org/wiki/Gossip_protocol.

Ringraziamenti

Oggi si conclude questo primo grande traguardo che mi ha permesso di crescere tanto a livello personale. Vorrei ringraziare tutti quelli che mi hanno aiutato fino ad oggi in questo percorso. Ringrazio il Professor Gianluigi Zavattaro, e i dottorandi Matteo Trentin e Giuseppe De Palma per avermi supportato durante la stesura della tesi ed essere stati sempre disponibili in qualunque momento.

Grazie in particolare a mia mamma e mio babbo che mi hanno permesso di essere qui e Ale che mi ha sempre aiutato diventando un punto di riferimento fino ad oggi. Grazie Zia per tutti i pasti preparati e Angi e Lori per avermi aiutato nei momenti del bisogno.

Ringrazio Prini Liuk e Alfano per questi primi tre anni insieme molto belli e divertenti. Mi avete permesso di crescere e concludere questo percorso (i prossimi siete voi).

Nel corso di questi anni ogni esperienza mi è stata utile e ha contribuito a quello che sono oggi. Ringrazio tutti i miei amici del corso perché senza di loro non sarei qui, in particolare Luca che con pazienza mi ha sempre aiutato, Andreea, Fede e Gaia per la vostra vivacità e simpatia, mi avete anche aiutato tantissimo con i vostri appunti meravigliosi (soprattutto con algebra). Grazie a Mattia e Volpe, siete stati una fonte di ispirazione per l'impegno costante che mettevate nelle cose nonostante il lavoro. Grazie a Paolo per la sua disponibilità nei momenti del bisogno, ricordo ancora le disperazioni durante i progetti. Grazie a Gian che mi ha insegnato a non arrendermi mai facendomi puntare sempre in alto in qualunque cosa facessi. Grazie a Geno, Ale, Dan, Fabio, Yonas, Apo con i quali ho passato le migliori serate che non dimenticherò mai. Grazie anche ad Andrea che nonostante il poco tempo passato qui a Bologna è stato sempre gentilissimo e un vero amico.

Grazie a Samu, Mascio e Barto, siete sempre stati disponibili e aperti con me, mi avete ospitato in Zanardi come se fosse la mia seconda casa. Grazie anche agli amici di Riccione per tutti gli anni passati e che passeremo ancora insieme.