



UNIVERSITÀ DEGLI STUDI DI CATANIA
CORSO DI LAUREA MAGISTRALE IN INFORMATICA

ROSARIO CANNAVÒ

ORCHESTRAZIONE E MONITORING
DI CLUSTER DI CONTAINER

APPROFONDIMENTO SISTEMI CENTRALI

Docente: Prof. Fabrizio Messina

Anno Accademico 2022-2023

Indice

1	Preliminari e storia del deployment	2
2	Containerizzazione	3
2.1	Vantaggi della containerizzazione	5
2.2	Use case della containerizzazione	6
2.3	Ciclo di vita di un container	8
2.4	Implementazione dei container	9
2.4.1	Container on bare-metal	10
2.4.2	Container on VM	11
2.4.3	Tradeoff	11
3	Docker	12
3.1	Immagini Docker	13
3.2	Dockerfile	14
3.3	Docker Demo	15
4	Orchestrazione di container	18
5	Kubernetes	19
5.1	Features	20
5.2	Architettura	21
5.2.1	Struttura del Master Node	22
5.2.2	Struttura di un worker node	23
5.2.3	Cosa non è Kubernetes	25
5.3	Componenti principali	26
5.4	Helm	28
6	Monitoring di Kubernetes	29
6.1	Kubernetes metric server	30
6.2	Monitoring tramite Prometheus	31
6.3	Esempio file di configurazione	34
6.4	Kubernetes Demo	38
7	Conclusioni	40

Introduzione

La continua espansione dei servizi e delle relative infrastrutture odierne hanno generato diverse volte la necessità di ingegnerizzare nuovi sistemi che permettessero ai fornitori di soddisfare le richieste e allo stesso tempo trovare un modo intelligente per gestire le proprie architetture. Il seguente lavoro vuole porsi come introduzione all'attuale stato dell'arte: le architetture orientate ai microservizi, la containerizzazione e il monitoring dei cluster virtuali, quest'ultimo necessario per garantire il corretto svolgimento delle operazioni nel minimo dei costi in delle soluzioni che non possono essere supervisionate a livello utente.

1 Preliminari e storia del deployment

Prima di introdurre le tematiche trattate, si reputa necessario ripercorrere i principali step del deployment di software. Le principali metodologie di sviluppo che sono state adottate nel tempo sono le qui sotto elencate.

L'era del deployment tradizionale: All'inizio, le organizzazioni eseguivano applicazioni su server fisici. Non c'era modo di definire i limiti delle risorse per le applicazioni in un server fisico e questo ha causato non pochi problemi di allocazione delle risorse. Ad esempio, se più applicazioni vengono eseguite sullo stesso server fisico, si possono verificare casi in cui un'applicazione assorbe la maggior parte delle risorse e, di conseguenza, le altre applicazioni non hanno le prestazioni attese. Una soluzione per questo problema sarebbe quella di eseguire ogni applicazione su un server fisico diverso. Ma questa non è una soluzione ideale, dal momento che le risorse vengono sottoutilizzate, inoltre, questa pratica risulta essere costosa per le organizzazioni, le quali devono mantenere numerosi server fisici.

L'era del deployment virtualizzato: Come soluzione venne introdotta la virtualizzazione. Essa consente di eseguire più macchine virtuali (VM) su una singola CPU fisica. La virtualizzazione consente di isolare le applicazioni in più macchine virtuali e fornisce un livello di sicurezza superiore, dal momento che le informazioni di un'applicazione non sono liberamente accessibili da un'altra. La virtualizzazione consente un migliore utilizzo delle risorse riducendo i costi per l'hardware, permette una migliore scalabilità, dato che un'applicazione può essere aggiunta o aggiornata facilmente, e ha molti altri vantaggi. Ogni VM è una macchina completa che esegue tutti i componenti, compreso il proprio sistema operativo, sopra all'hardware virtualizzato.

L'era del deployment in container: I container sono simili alle macchine virtuali, ma presentano un modello di isolamento più leggero, condividendo il sistema operativo (OS) tra le applicazioni. Pertanto, i container sono considerati più leggeri. Analogamente a una macchina virtuale, un container dispone di una segregazione di filesystem, CPU, memoria, PID e altro ancora. Poiché sono disaccoppiati dall'infrastruttura sottostante, risultano portabili tra differenti cloud e diverse distribuzioni.

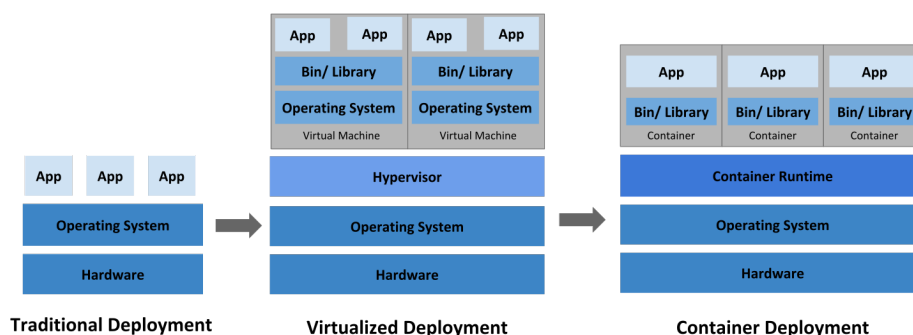


Figura 1: Fasi del deployment

2 Containerizzazione

La tecnologia di containerizzazione[1] è stata proposta per la prima volta da IBM nel 1979. I primi esperimenti risalgono al sistema operativo UNIX V7, che introduceva la chiamata di sistema chroot. Questo progresso è stato l'inizio del processo di isolamento, con l'esecuzione di gruppi isolati su un singolo host. L'isolamento fa leva su diverse tecnologie di base costruite sul kernel di Linux: namespace e cgroup. I container Linux sono insiemi di uno o più processi isolati dal resto del sistema in grado di eseguire applicazioni in modo isolato, poiché utilizzano le funzionalità di virtualizzazione a livello di sistema operativo fornite dal kernel Linux, come ad esempio i namespace,

i cgroups e i chroot. Questi meccanismi consentono ai container di avere una visione limitata e controllata delle risorse del sistema operativo, senza interferire con le altre applicazioni o con il sistema operativo host. Inoltre essi risultano essere portabili e coerenti in tutti gli ambienti, dallo sviluppo, ai test, fino alla produzione. Questo li rende molto più veloci da utilizzare, rispetto ai tradizionali flussi di sviluppo che dipendono dalla replica degli ambienti di test tradizionali.

La tecnologia dei container, intesi con accezione moderna, è stata implementata per la prima volta a partire dal 2006 da parte di Google, che aveva bisogno di un modo per gestire le proprie applicazioni su larga scala in modo più efficiente. Google ha creato un sistema chiamato "**cgroups**" (control groups), che consente di limitare e isolare le risorse del sistema, come la CPU e la memoria, per le diverse applicazioni. Questo sistema è stato poi incorporato nel kernel Linux, diventando parte integrante del sistema operativo. Il primo sistema di container Linux è stato **LXC** (Linux Containers), sviluppato nel 2008 come parte del progetto OpenVZ. LXC utilizzava la tecnologia dei cgroups per isolare le risorse del sistema e creare un ambiente di esecuzione autonomo per le applicazioni. Tuttavia, LXC non era ancora molto user-friendly e richiedeva una certa conoscenza del sistema operativo Linux per essere utilizzato. Nel 2013 è stato introdotto **Docker**, un software che ha semplificato l'utilizzo dei container Linux, offrendo un'interfaccia più user-friendly e facilitando la distribuzione di applicazioni containerizzate. Docker ha introdotto il concetto di immagini Docker, ovvero pacchetti software completi di tutto il necessario per eseguire un'applicazione, compresi i file di configurazione, le librerie e le dipendenze. Grazie alle immagini Docker, è diventato molto più facile distribuire le applicazioni containerizzate, senza la necessità di installare e configurare le dipendenze sul sistema operativo

host. Oltre a Docker, sono stati sviluppati altri software per la gestione dei container Linux, come ad esempio Kubernetes[4], che offre un'infrastruttura completa per la gestione dei container in un ambiente distribuito. Kubernetes consente di orchestrare i container, garantendo che le risorse siano allocate in modo efficiente e che le applicazioni siano distribuite in modo uniforme su diversi host. In sintesi, i container Linux sono un'evoluzione delle tradizionali macchine virtuali che offre un ambiente di esecuzione isolato e sicuro per le applicazioni, con una maggiore portabilità, sicurezza e gestione delle risorse. Grazie alla loro natura leggera ed efficiente, i container Linux sono diventati uno degli strumenti principali per la distribuzione e la gestione di applicazioni containerizzate.

2.1 Vantaggi della containerizzazione

Oltre alla modernità dell'architettura, la containerizzazione comporta altri vantaggi[2] di diversi natura, tra i più evidenti si ritrovano:

1. **Portabilità:** Gli sviluppatori di software utilizzano la containerizzazione per implementare le applicazioni in più ambienti, senza che sia necessario riscrivere il codice del programma. Creano l'applicazione una sola volta e la implementano così com'è su più sistemi operativi. Ad esempio, eseguono gli stessi container su sistemi operativi Linux e Windows. Utilizzando i container per l'implementazione, gli sviluppatori aggiornano anche il codice delle applicazioni legacy alle versioni più recenti.
2. **Scalabilità:** I container sono componenti software leggeri che offrono un'esecuzione efficiente. Ad esempio, una macchina virtuale può avviare un'applicazione containerizzata più velocemente perché non ha bisogno di avviare un sistema operativo. Pertanto, gli sviluppatori di

software possono facilmente aggiungere su una singola macchina più container per diverse applicazioni. Il cluster di container utilizza le risorse di calcolo da uno stesso sistema operativo condiviso, ma un container non interferisce con il funzionamento degli altri.

3. **Tolleranza ai guasti:** I team di sviluppo software utilizzano i container per creare applicazioni con tolleranza ai guasti. Usano più container per eseguire i microservizi su cloud. Poiché i microservizi containerizzati operano in spazi utente isolati, la presenza di un guasto su un singolo container non influisce sugli altri container. Ciò aumenta la resilienza e la disponibilità dell'applicazione.
4. **Agilità:** Le applicazioni containerizzate vengono eseguite in ambienti informatici isolati. Gli sviluppatori di software possono risolvere i problemi e modificare il codice dell'applicazione senza interferire con il sistema operativo, l'hardware o altri servizi applicativi. Con questo modello basato sui container, possono abbreviare i cicli di rilascio del software e lavorare rapidamente sugli aggiornamenti.
5. **Isolamento:** La containerizzazione consente di creare container isolati per ogni applicazione, in modo che ciascuna applicazione possa essere eseguita in un ambiente separato. Ciò previene problemi di conflitto tra le diverse applicazioni e garantisce che ogni applicazione possa funzionare correttamente senza influenzare le altre.

2.2 Use case della containerizzazione

Questa metodologia di sviluppo permette di ottenere anche altri vantaggi in scenari specifici che sarebbero molto più complessi e onerosi da realizzare in modo diverso, tra i tanti troviamo:

- **Migrazione nel cloud:** La migrazione al cloud, o approccio *lift-and-shift*, è una strategia software che prevede l'incapsulamento delle applicazioni legacy in container e la loro implementazione in un ambiente di cloud computing. Le organizzazioni possono modernizzare le proprie applicazioni senza riscrivere l'intero codice software. Con i container, la migrazione al cloud può essere rapida ed efficace. Essi permettono inoltre di modernizzare le applicazioni e di sfruttare i vantaggi del cloud senza dover ricodificare o ricostruire. Una volta nel cloud, questa tecnologia è utile anche negli scenari in cui è necessario creare o estendere rapidamente gli ambienti. In definitiva, i container sono una parte fondamentale del mix di adozione del cloud.
- **Adozione dell'architettura di microservizi:** Nelle architetture monolitiche tutti i processi sono strettamente collegati tra loro e vengono eseguiti come un singolo servizio. Ciò significa che se un processo dell'applicazione sperimenta un picco nella richiesta, è necessario ridimensionare l'intera architettura. Aggiungere o migliorare una funzionalità dell'applicazione monolitica diventa più complesso, in quanto sarà necessario aumentare la base di codice. Tale complessità limita la sperimentazione e rende più difficile implementare nuove idee. Le architetture monolitiche rappresentano un ulteriore rischio per la disponibilità dell'applicazione, poiché la presenza di numerosi processi dipendenti e strettamente collegati aumenta l'impatto di un errore in un singolo processo. Con un'architettura basata su microservizi, un'applicazione è realizzata da componenti indipendenti che eseguono ciascun processo applicativo come un servizio. Tali servizi comunicano attraverso un'interfaccia ben definita che utilizza API leggere. I servizi sono realizzati per le funzioni aziendali e ogni servizio esegue una sola funzio-

ne. Poiché eseguito in modo indipendente, ciascun servizio può essere aggiornato, distribuito e ridimensionato per rispondere alla richiesta di funzioni specifiche di un'applicazione. Le organizzazioni che cercano di creare applicazioni cloud con microservizi richiedono una tecnologia di containerizzazione, in grado di fornire uno strumento software per creare pacchetti di microservizi come programmi implementabili su piattaforme diverse.

- **Dispositivi IoT:** I dispositivi di Internet of Things (IOT) contengono risorse informatiche limitate, il che rende l'aggiornamento manuale del software un processo complesso. La containerizzazione consente agli sviluppatori di implementare e aggiornare facilmente le applicazioni su dispositivi IoT.

2.3 Ciclo di vita di un container

Il ciclo di vita di un container si riferisce all'esplorazione degli stati possibili per il container. Il gestore di container fornisce un framework che offre una serie di API. Questo consente di gestire facilmente il ciclo di vita del container, considerando la creazione, la costruzione, l'esecuzione e la manutenzione del contenitore. Per prima cosa, lo sviluppatore deve creare la sua nuova immagine come modello. In seguito, tutte le operazioni saranno eseguite con questa. L'intero ciclo di vita del container, dopo la creazione dell'immagine di base, passa allo stato di **running**. Inoltre, può attraversare diversi stati, come **paused**, **killed** e **stopped**. Il ciclo di vita delle applicazioni container inizia con degli sviluppatori che creano un'immagine basata su container e pongono tutto al suo interno, compreso il software, le librerie e tutte le dipendenze, è inoltre raccomandato l'uso di librerie diverse da quelle utilizzate dal sistema operativo ospitante per rafforzare la sicurezza del container e

risolvere i conflitti applicativi tra altri ambienti. Una volta creata l'immagine è possibile convalidarla e inserirla all'interno di un registry remoto e, infine, condividerla facilmente con le IT Operations e lanciarla nell'ambiente di produzione. I vantaggi consistono nel semplificare la creazione, il test e la distribuzione delle pipeline del flusso di lavoro dei container per l'ambiente DevOps, dove gli sviluppatori possono avvalersi di terze parti come **Jenkins** per automatizzare i rilasci delle loro applicazioni Docker.

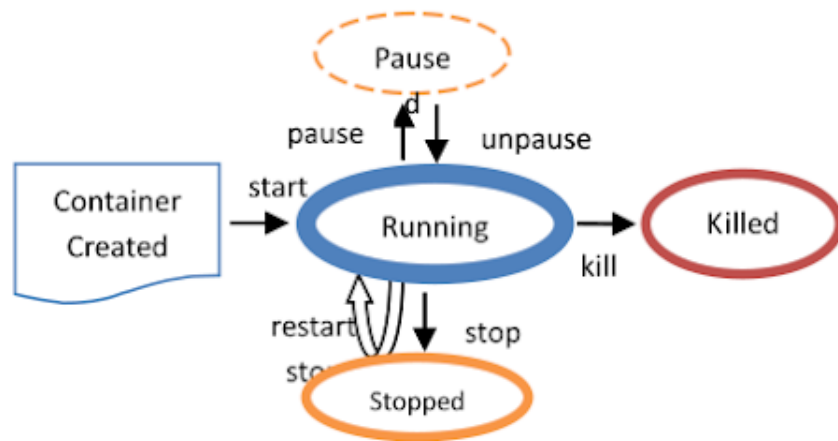


Figura 2: Ciclo di vita di un container

2.4 Implementazione dei container

Scendendo a livello implementativo, esistono due diverse filosofie per l'implementazione dei container all'interno di macchine server, entrambe con particolari lati negativi o positivi. È possibile eseguire i container su **macchine virtuali** (VM), su **bare metal** o su entrambi. Ma qual è la soluzione migliore? Il dibattito sull'uso dei container su macchine virtuali o bare metal si riduce in realtà alla velocità e all'uso efficiente delle risorse hardware rispetto all'isolamento e alla facilità di portabilità. Vediamo i vantaggi e gli svantaggi di entrambi

2.4.1 Container on bare-metal

L'esecuzione di container su server bare metal presenta numerosi vantaggi. Innanzitutto, si elimina l'overhead del sistema operativo host, o sia dell'hypervisor che del sistema operativo guest, se il container viene eseguito in una macchina virtuale. I container in esecuzione su bare metal utilizzano le risorse di sistema in modo più efficiente rispetto ai container basati su macchine virtuali. Per loro stessa natura, i container dispongono di tutte le risorse necessarie per l'esecuzione, compresi i propri filesystem e stack di rete, il che consente di eseguire molti più container sulla stessa infrastruttura rispetto alle macchine virtuali. L'uso efficiente della memoria, dell'elaborazione e delle risorse di rete ha convinto alcuni operatori del settore che i container bare metal sono la strada da percorrere. Inoltre, se l'azienda che adotta i container paga per un hypervisor premium, l'esecuzione di container su server bare metal potrebbe ridurre notevolmente i costi di implementazione. Se la preoccupazione principale è quella di sfruttare al meglio l'infrastruttura, questa è la soluzione ottima. Le principali preoccupazioni legate all'esecuzione di container su server bare metal riguardano la sicurezza, ma non è l'unico problema. Le argomentazioni sulla sicurezza si riducono alla mancanza di un vero isolamento. Poiché i container utilizzano il kernel del sistema operativo e, nella maggior parte dei casi, le stesse librerie, potrebbero potenzialmente rappresentare un'opportunità migliore per un aggressore se il container potesse essere compromesso. La mancanza di isolamento può anche consentire a diversi container di monopolizzare le risorse dell'host, interferendo con altri container in esecuzione sulla stessa piattaforma. L'ultimo problema riguarda il sacrificio della portabilità; poiché il container utilizza il kernel dell'host, potrebbero esserci problemi quando si cerca di portarlo su piattaforme diverse.

2.4.2 Container on VM

Aggiungere un livello di virtualizzazione al sistema operativo host significa aggiungere un livello di software all'ambiente, il che comporta sia vantaggi che svantaggi. In un ambiente di virtualizzazione, l'hypervisor offre molte funzionalità e consente di massimizzare l'utilizzo dell'hardware. I vantaggi principali sono che i carichi di lavoro possono essere migrati facilmente da un host all'altro, anche se non condividono lo stesso sistema operativo host sottostante. Questo è un aspetto utile soprattutto per i container, che sono desiderabili per la loro portabilità da un luogo all'altro, ma dipendono dal sistema operativo in cui sono stati creati. L'uso di un particolare panorama di virtualizzazione fornirà un ambiente software coerente in cui eseguire applicazioni containerizzate anche se il sistema operativo host è diverso.

2.4.3 Tradeoff

La distribuzione ottimale può essere una combinazione delle due tecnologie. L'implementazione di un container all'interno di una macchina virtuale soddisferà i puristi dell'isolamento, migliorando al contempo la portabilità del container, poiché le macchine virtuali possono essere spostate da una piattaforma all'altra come i container su un server bare metal non possono fare. I container bare metal non hanno la scalabilità e l'elasticità che spingono la maggior parte delle aziende verso le soluzioni cloud. L'implementazione del container all'interno di una macchina virtuale potrebbe offrire questi vantaggi, preservando al contempo i benefici in termini di prestazioni che ci si aspetta dall'utilizzo dei container.

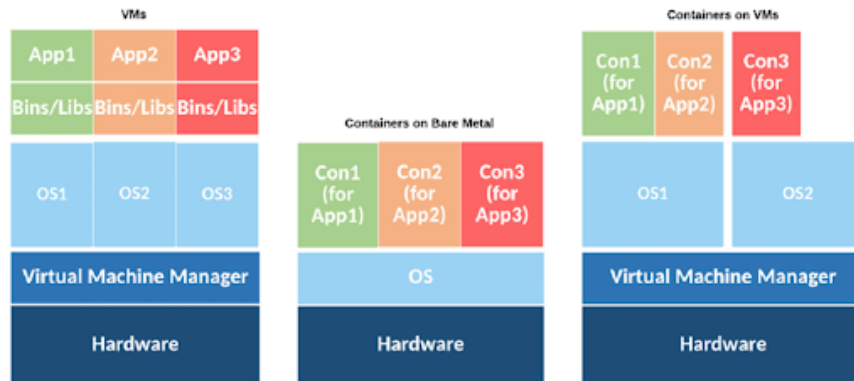


Figura 3: Bare-metal vs on-virtual machine

3 Docker

Docker[3] è una piattaforma software che permette di creare, testare e distribuire applicazioni con la massima rapidità. Essa raccoglie il software in unità standardizzate chiamate **Docker container** che ricalcano esattamente l'essenza dei container Linux. Con Docker, è possibile distribuire e ricalibrare le risorse per un'applicazione in qualsiasi ambiente, tenendo sempre sotto controllo il codice eseguito. Docker permette di distribuire il codice più rapidamente, standardizzare il funzionamento delle applicazioni, trasferire il codice in modo ottimizzato e risparmiare denaro migliorando l'utilizzo delle risorse. Con Docker è possibile ottenere un singolo oggetto che può essere eseguito in modo affidabile in qualsiasi posizione. La sua sintassi è semplice e permette di tenere le risorse sotto controllo. Il fatto che sia già diffuso significa che offre un ecosistema di strumenti e applicazioni pronte all'uso.

3.1 Immagini Docker

Un'immagine Docker è un template di sola lettura che viene fornito con le istruzioni per il deploy dei container. In Docker, tutto ruota fondamentalmente intorno alle **immagini**. Un'immagine consiste in una collezione di file (o layer) che mettono insieme tutte le necessità, come le dipendenze, il codice sorgente e le librerie, necessarie per impostare un ambiente container completamente funzionale. Le immagini possono essere memorizzate in locale, o su un registro remoto come **Docker hub**[12], in modo da essere sempre fruibili e utilizzabili per la creazione di container su qualsiasi ambiente.

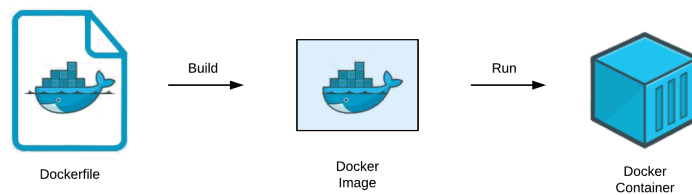


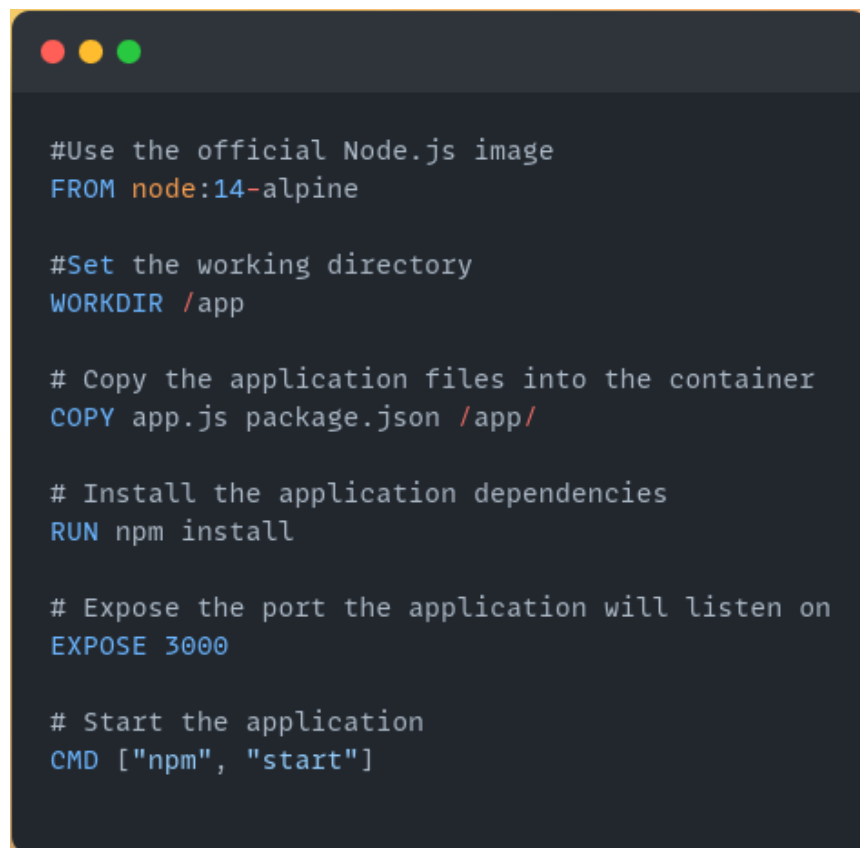
Figura 4: Pipeline di creazione di un'immagine docker

3.2 Dockerfile

I Dockerfile sono file utilizzati per creare in modo programmatico immagini Docker. Essi consentono di creare in modo rapido e riproducibile un'immagine Docker e quindi sono utili per la containerizzazione di applicativi. I Dockerfile sono formati da istruzioni per la creazione di un'immagine Docker, dove ogni istruzione è scritta su una riga e viene data nella forma:

$$< INSTRUCTION > < argument(s) >$$

Durante la creazione di un Dockerfile, il **client Docker** invierà un "build context" al **daemon Docker**, il processo che permette di eseguire i container, dove il contesto di costruzione include tutti i file e le cartelle nella stessa directory del file Docker. Grazie ad essi la containerizzazione di applicazioni diventa diretta e facile da utilizzare, rendendo Docker uno strumento ancora più performante.



```
#Use the official Node.js image
FROM node:14-alpine

#Set the working directory
WORKDIR /app

# Copy the application files into the container
COPY app.js package.json /app/

# Install the application dependencies
RUN npm install

# Expose the port the application will listen on
EXPOSE 3000

# Start the application
CMD ["npm", "start"]
```

Figura 5: Esempio di Dockerfile

3.3 Docker Demo

Il seguente esempio si concentra sull'utilizzo di Docker per la creazione di un semplice server Express[6]. L'obiettivo è mostrare la facilità con cui è possibile creare un'immagine Docker e farla funzionare, senza dover preoccuparsi di configurazioni complesse o dipendenze. Il server Express in questione è stato creato utilizzando il framework Node.js, con l'aggiunta di un endpoint API che restituisce una stringa di testo. Per creare l'immagine Docker, è stato utilizzato un file Dockerfile, che definisce le istruzioni per la creazione dell'immagine. Il Dockerfile è stato progettato in modo da essere il più semplice

possibile, con l'aggiunta di solo le istruzioni necessarie per far funzionare il server. È stato utilizzato un'immagine base di Node.js, con l'aggiunta delle dipendenze necessarie per il server Express. Una volta creato il Dockerfile,



```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
const port = 3000;

app.use(bodyParser.json());

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(port, () => {
  console.log(`Server running on port ${port}`);
});
```

Figura 6: Server Express.js

è stata eseguita la build dell'immagine del server Express che è stata facilmente distribuita e utilizzata in un ambiente di sviluppo o di produzione, senza la necessità di installare manualmente il server e le relative dipendenze. Inoltre, Docker ha permesso di creare un ambiente isolato per il server, garantendo la sicurezza e la compatibilità con altri software. Per la costruzione dell'immagine il comando da utilizzare è il seguente:

docker build -t express - api - docker

Mentre per lanciare il container è necessario eseguire il comando:

docker run -p 3000 : 3000 express - api - docker

```
# Usa l'immagine Node.js versione 14 come base
FROM node:14

# Crea una directory per l'app
WORKDIR /usr/src/app

# Copia il file package.json e package-lock.json
COPY package*.json ./

# Installa le dipendenze
RUN npm install

# Copia il codice sorgente dell'app
COPY . .

# Espone la porta 3000
EXPOSE 3000

# Avvia l'app
CMD ["npm", "start"]
```

Figura 7: Dockerfile

4 Orchestrazione di container

Un cluster di container è un gruppo di server o nodi (nodes) o host che lavorano insieme per eseguire, gestire e distribuire i container delle applicazioni. Un cluster di container consente di distribuire e gestire i container in modo efficace, migliorando la flessibilità e l'affidabilità dell'infrastruttura di gestione dei container. L'orchestrazione di container è un processo di gestione e coordinamento di un cluster di container. L'obiettivo dell'orchestrazione è di automatizzare il deployment, la scalabilità e la gestione di un'applicazione containerizzata, garantendo un'infrastruttura altamente disponibile e scalabile. Le piattaforme di orchestrazione di container più comuni sono Docker Swarm e Kubernetes. Entrambe queste piattaforme consentono di gestire e scalare facilmente un'infrastruttura di container, semplificando il deployment e la gestione di applicazioni containerizzate su larga scala. Una tipologia di orchestrazione lightweight è invece Docker Compose[5] che permette di gestire piccoli insiemi di container in modo semplice e unificato.

Docker Compose è progettato per orchestrare e gestire un insieme di container su un singolo host. Con Docker Compose è possibile definire un file di composizione che descrive l'infrastruttura dell'applicazione e specifica i servizi, le reti e i volumi necessari per far funzionare l'applicazione. Inoltre, Docker Compose consente di avviare e fermare facilmente tutti i container dell'applicazione in modo coordinato.

Docker Swarm, d'altra parte, è progettato per gestire un cluster di host Docker, in cui i container possono essere distribuiti su più host per garantire una maggiore disponibilità e scalabilità dell'applicazione. Docker Swarm[10] consente di definire servizi e stack di servizi, che possono essere distribuiti su più host, e gestisce automaticamente la ripartizione del carico, la scalabilità e la disponibilità degli stessi. Inoltre, Docker Swarm fornisce funzionalità

avanzate di sicurezza e gestione, come il controllo degli accessi e la crittografia dei dati in transito.

Kubernetes, o anche K8s, è una piattaforma di orchestrazione di container open-source sviluppata da Google. Quest'ultima verrà trattata nello specifico nel capitolo dedicato. In tutte le piattaforme presentate, il deployment di un'applicazione containerizzata avviene attraverso la definizione di un file YAML o JSON che specifica le risorse, i volumi, le variabili di ambiente e le configurazioni necessarie per il corretto funzionamento dell'applicazione. L'orchestratore di container si occupa poi di gestire la distribuzione dei container sui vari host disponibili, garantendo che l'applicazione funzioni correttamente e che la risorsa disponibile sia utilizzata in modo efficiente. In sintesi, concentrandosi sugli orchestratori di cluster, sia Docker Swarm che Kubernetes sono piattaforme di orchestrazione di container che consentono di gestire e scalare facilmente un'infrastruttura di container. Mentre Docker Swarm è una soluzione più semplice per cluster di dimensioni ridotte, Kubernetes offre funzionalità più avanzate per cluster di dimensioni maggiori e per applicazioni di produzione.

5 Kubernetes

Il nome Kubernetes deriva dal greco, significa timoniere o pilota, tale nome rappresenta esattamente la sua essenza in quanto esso è una piattaforma portatile, estensibile e open-source per la gestione di carichi di lavoro e servizi containerizzati, in grado di facilitare sia la configurazione dichiarativa che l'automazione. La piattaforma vanta un grande ecosistema in rapida crescita e servizi, supporto e strumenti sono ampiamente disponibili nel mondo Kubernetes. L'aumento delle applicazioni basate su microservizi ha portato all'aumento dell'utilizzo dei container, in quanto essi offrono il perfetto am-

biente per applicazioni piccole e indipendenti come appunto i microservizi. Questo trend ha portato ad applicazioni odierne che fanno uso di centinaia di container che in passato venivano gestiti attraverso script handmade o tools privati molto difficili da utilizzare. Da questo scenario è nata la necessità di avere un tool semplice da usare e universale per l'orchestrazione di container come Kubernetes.

5.1 Features

Kubernetes fornisce un framework per far funzionare i sistemi distribuiti in modo resiliente integrando l'uso di diverse proprietà quali:

- Scoperta dei servizi e bilanciamento del carico, Kubernetes può esporre un container usando un nome DNS o il suo indirizzo IP. Se il traffico verso un container è alto, Kubernetes è in grado di distribuire il traffico su più container in modo che il servizio rimanga stabile.
- Orchestrazione dello storage, Kubernetes permette di montare automaticamente un sistema di archiviazione a scelta, come per esempio storage locale o i dischi forniti da cloud pubblici.
- Ottimizzazione dei carichi, fornendo a Kubernetes un cluster di nodi per eseguire i container lo si può istruire su quante risorse utilizzare, esso allocherà i container sui nodi per massimizzare l'uso delle risorse a disposizione.
- Self-healing, Kubernetes riavvia i container che si bloccano, sostituisce container, termina i container che non rispondono agli health checks, e evita di far arrivare traffico ai container che non sono ancora pronti per rispondere correttamente.

- Gestione di informazioni sensibili e della configurazione, Kubernetes consente di memorizzare e gestire informazioni sensibili, come le password, i token OAuth e le chiavi SSH.
- Bilanciamento del carico, Kubernetes fornisce funzionalità avanzate di bilanciamento del carico, che consentono di distribuire il traffico in modo equilibrato tra le istanze dell'applicazione. Ciò garantisce un'esperienza utente uniforme e prevenire la congestione dell'infrastruttura.
- Gestione delle configurazioni, Kubernetes consente di gestire le configurazioni dell'applicazione in modo efficace e scalabile, fornendo un'infrastruttura flessibile e sicura per la gestione delle variabili di ambiente e delle configurazioni dell'applicazione.

5.2 Architettura

Un cluster Kubernetes è composto da almeno un **nodo master** connesso ad una serie di **worker node**, ognuno dei quali ha un processo kubelet in esecuzione su di esso. Un **kubelet** non è altro che un processo Kubernetes che consente ai nodi del cluster di comunicare tra di loro e di eseguire operazioni come fare il run di un'applicazione. Ogni worker node ha al suo interno diverse applicazioni containerizzate, quindi, i worker node sono effettivamente le macchine su cui girano le applicazioni. Il master node esegue invece diversi processi Kubernetes che sono necessari alla corretta gestione del cluster. Kubernetes è *technology agnostic* perché è stato progettato per essere un framework di gestione delle applicazioni indipendente dalla tecnologia sottostante. Ciò significa che esso non è limitato a un singolo tipo di container runtime o a una specifica piattaforma cloud. Invece, supporta una vasta gamma di tecnologie container e cloud, tra cui Docker, rkt, CRI-O

e molte altre. Inoltre, Kubernetes è altamente configurabile e personalizzabile. Gli utenti possono utilizzare strumenti e risorse personalizzati per integrare Kubernetes con altre tecnologie, come i servizi di monitoraggio, le reti software-defined, le soluzioni di sicurezza e molti altri. Ciò rende questo strumento una piattaforma estremamente flessibile e adatta a una vasta gamma di casi d'uso, dalle applicazioni Web alle applicazioni di intelligenza artificiale e di machine learning. Vediamo adesso la struttura di ogni nodo del cluster.

5.2.1 Struttura del Master Node

All'interno di un master node sono presenti diversi processi che permettono di gestire il cluster, i suoi componenti e la comunicazione con il mondo esterno.

Tra i principali componenti del master node troviamo:

- server API, è a sua volta un container e non è altro che l'entry point al cluster Kubernetes, ovvero, il processo con cui i diversi client si interfaceranno;
- controller manager, è un processo il cui compito è quello di tenere sott'occhio la situazione dei vari nodi che compongono il cluster, ad esempio esso ha il compito di capire se qualcosa deve essere riparato, di capire se un container è morto e deve essere riavviato, ecc;
- scheduler, il processo il cui compito è quello di schedulare i container sui diversi nodi del cluster in base al carico di lavoro e alle risorse disponibili su ogni nodo;
- etcd, ovvero un archivio di valori chiave che mantiene in ogni momento lo stato corrente del cluster Kubernetes, mantiene dunque tutti i dati di configurazione e tutti i dati di stato di ogni nodo e di ogni container

all'interno dello specifico nodo, contiene inoltre lo snapshot del backup e del ripristino utili in fase di disaster recovery;

- L'ultimo componente di che permette ai worker node di comunicare con il master node è la rete virtuale che abbraccia tutti i nodi che fanno parte del cluster: tramite questa rete, tutti i nodi del cluster riescono a lavorare effettivamente come un'unica macchina che ha come potenza la somma di tutte le risorse dei vari nodi.

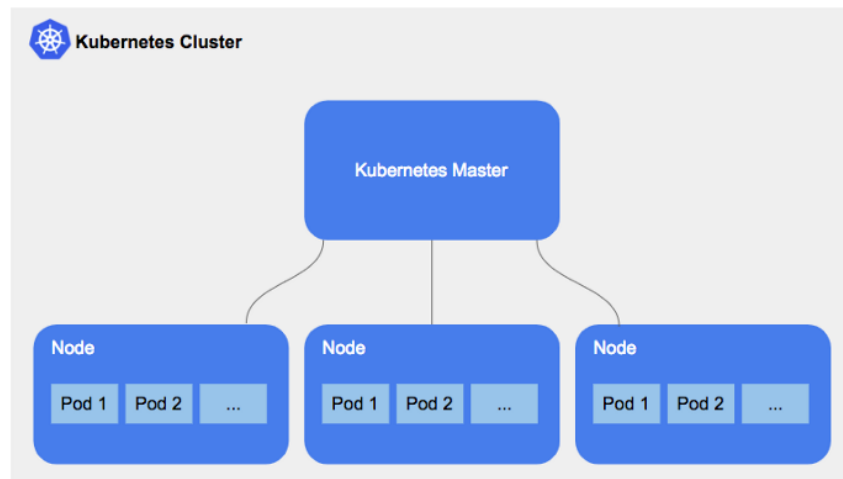


Figura 8: Architettura di un cluster Kubernetes

5.2.2 Struttura di un worker node

All'interno di ogni worker node sono invece contenuti altri processi che hanno il compito di gestire l'esecuzione dei Pod al loro interno. Tra di essi troviamo:

- Kubelet: Un agente che è eseguito su ogni nodo del cluster. Si assicura che i container siano eseguiti in un pod. La kubelet riceve un set di PodSpecs che vengono fornite attraverso vari meccanismi, e si assicura che i container descritti in questi PodSpecs funzionino correttamente e siano sani.

- Kube-proxy: è un proxy eseguito su ogni nodo del cluster, responsabile della gestione dei Kubernetes Service. I kube-proxy mantengono le regole di networking sui nodi. Queste regole permettono la comunicazione verso gli altri nodi del cluster o l'esterno.
- Container Runtime: Il container runtime è il software che è responsabile per l'esecuzione dei container. Kubernetes supporta diversi container runtimes: Docker, containerd, cri-o, rktlet e tutte le implementazioni di Kubernetes CRI (Container Runtime Interface).

E' importante notare che il più delle volte la potenza dei worker node è maggiore rispetto a quella del master node in quanto i worker node hanno l'effettivo carico di lavoro (eseguono i container contenenti le applicazioni) mentre nel master vengono eseguiti solo pochi processi necessari al coordinamento dell'interno cluster.

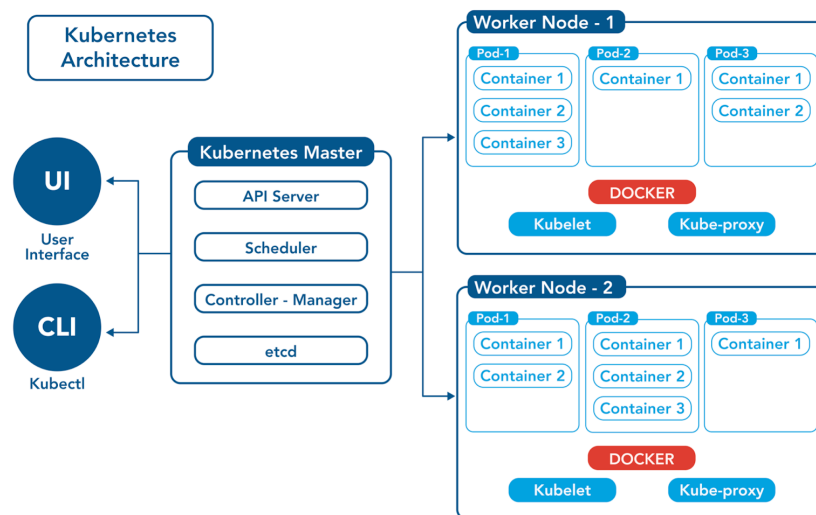


Figura 9: Dettaglio dell'architettura di un cluster kubernetes

5.2.3 Cosa non è Kubernetes

Kubernetes non è un sistema PaaS (Platform as a Service) tradizionale e completo. Dal momento che Kubernetes opera a livello di container piuttosto che a livello hardware, esso fornisce alcune caratteristiche generalmente disponibili nelle offerte PaaS, come la distribuzione, il ridimensionamento, il bilanciamento del carico, la registrazione e il monitoraggio. Tuttavia, Kubernetes non è monolitico, e queste soluzioni predefinite sono opzionali ed estensibili. Kubernetes fornisce gli elementi base per la costruzione di piattaforme di sviluppo, ma conserva le scelte dell'utente e la flessibilità dove è importante. Kubernetes:

1. Non limita i tipi di applicazioni supportate. Kubernetes mira a supportare una grande varietà di carichi di lavoro, compresi i carichi di lavoro stateless, stateful e elaborazione di dati. Se un'applicazione può essere eseguita in un container, dovrebbe funzionare bene anche su Kubernetes.
2. Non compila il codice sorgente e non crea i container. I flussi di Continuous Integration, Delivery, and Deployment (CI/CD) sono determinati dalla cultura e dalle preferenze dell'organizzazione e dai requisiti tecnici.
3. Non fornisce servizi a livello applicativo, come middleware (per esempio, bus di messaggi), framework di elaborazione dati (per esempio, Spark), database (per esempio, mysql), cache, né sistemi di storage distribuito (per esempio, Ceph) come servizi integrati. Tali componenti possono essere eseguiti su Kubernetes, e/o possono essere richiamati da applicazioni che girano su Kubernetes attraverso meccanismi come l'Open Service Broker.

4. Non impone soluzioni di logging, monitoraggio o di gestione degli alert. Fornisce alcune integrazioni come dimostrazione, e meccanismi per raccogliere ed esportare le metriche.
5. Non fornisce né rende obbligatorio un linguaggio/sistema di configurazione (per esempio, Jsonnet). Fornisce un'API dichiarativa che può essere richiamata da qualsiasi sistema.
6. Inoltre, Kubernetes non è un semplice sistema di orchestrazione. Infatti, questo sistema elimina la necessità di orchestrazione. La definizione tecnica di orchestrazione è l'esecuzione di un flusso di lavoro definito: prima si fa A, poi B, poi C. Al contrario, Kubernetes è composto da un insieme di processi di controllo indipendenti e componibili che guidano costantemente lo stato attuale verso lo stato desiderato. Non dovrebbe importare come si passa dalla A alla C. Anche il controllo centralizzato non è richiesto. Questo si traduce in un sistema più facile da usare, più potente, robusto, resiliente ed estensibile.

5.3 Componenti principali

I principali componenti di kubernetes sono elencati di seguito:

- *Pod*, sono le più piccole unità di calcolo distribuibili che è possibile creare e gestire in Kubernetes. Un Pod è un gruppo di uno o più container, con risorse di rete e di storage condivise. I contenuti di un Pod sono sempre co-localizzati e co-programmati, ed eseguiti in un contesto condiviso. Un Pod modella un "host logico" specifico dell'applicazione: contiene uno o più container di applicazioni che sono relativamente strettamente accoppiate. Un altro importante concetto di kubernetes

è che i pod sono effimeri, ovvero possono morire molto facilmente a causa di problemi di varia natura.

- *Services*, sono un modo astratto per esporre un'applicazione in esecuzione su un insieme di Pod come servizio di rete. Con Kubernetes è necessario modificare l'applicazione per utilizzare un meccanismo di service discovery, esso stesso dà ai Pod i loro indirizzi IP e un singolo nome DNS per un insieme di Pod.
- *Volumi*, i file su disco in un container sono effimeri, il che presenta alcuni problemi per applicazioni non banali quando vengono eseguite su container. Un problema è la perdita di file quando un container va in crash. Il kubelet riavvia il contenitore ma senza ripristinare. L'astrazione del volume di Kubernetes risolve questo problema.
- *Namespace*, forniscono un meccanismo per isolare gruppi di risorse all'interno di un singolo cluster.
- *Deployment*, è un oggetto che fornisce aggiornamenti dichiarativi alle applicazioni. Un deployment consente di descrivere il ciclo di vita dell'applicazione, specificando ad esempio le immagini da utilizzare, il numero di pod necessari e le modalità di aggiornamento relative, può essere visto come un "blueprint" di un pod.
- *StatefulSet*, è l'oggetto utilizzato per gestire le applicazioni stateful. Gestisce il deployment e lo scaling di un insieme di Pod, e fornisce garanzie circa l'ordine e l'unicità di questi Pod. Come un Deployment, uno StatefulSet gestisce i Pod che sono basati su specifiche univoche. A differenza di un Deployment, uno StatefulSet mantiene un'identità per ciascuno dei suoi Pod. Questi Pod sono creati dalla stessa specifica,

ma non sono intercambiabili: ognuno ha un identificatore persistente che mantiene attraverso qualsiasi riprogrammazione. A questo punto si può dedurre che per applicazioni tipicamente stateful, come i database, il deployment viene effettuato attraverso gli statefulset mentre le i contesti stateless, come le applicazioni statiche, vengono usati i deployment.

- *ReplicaSet*, lo scopo di un ReplicaSet è quello di mantenere un insieme stabile di Pod replica in esecuzione in un dato momento. Come tale, è spesso utilizzato per garantire la disponibilità di un determinato numero di Pod identici.
- *Job*, crea uno o più Pod e continua a riprovare la loro esecuzione fino a quando un numero specificato di essi si conclude con successo. Man mano che i Pod si completano, il job tiene traccia dei completamenti riusciti e una volta raggiunto un numero specificato essi, viene innescato un meccanismo che elimina tutti i Pod che sono serviti a portare a termine il compito specificato.

5.4 Helm

Helm[11] è uno strumento open-source per la gestione del deployment di applicazioni Kubernetes. Kubernetes è un sistema di orchestrazione dei container, mentre Helm offre un modo più semplice e organizzato per installare, gestire e aggiornare le applicazioni all'interno di Kubernetes. Helm si basa su una struttura a pacchetti chiamati *chart*, che contengono tutti i file necessari per installare e configurare un'applicazione in Kubernetes. Questi pacchetti sono composti da template YAML e possono includere file di configurazione, dipendenze e istruzioni di installazione personalizzate. Inoltre, Helm offre

funzionalità avanzate come il rollback delle installazioni, la gestione di release multiple, la condivisione dei pacchetti tra team e la personalizzazione dell'installazione in base alle esigenze dell'utente. Helm è molto utilizzato nella community di Kubernetes per semplificare la gestione dei deployment di applicazioni complesse e offre un'esperienza utente molto user-friendly grazie alla sua interfaccia a riga di comando (CLI) e alla vasta documentazione disponibile.

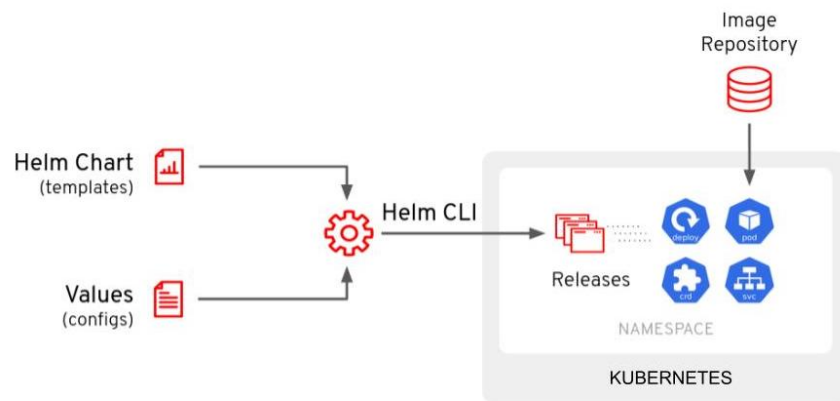


Figura 10: flowchart Helm

6 Monitoring di Kubernetes

Il monitoring di un cluster Kubernetes è essenziale per garantire l'affidabilità e la disponibilità delle applicazioni containerizzate. Il monitoraggio è una parte importante dell'infrastruttura di gestione dei container, che consente di raccogliere dati sulle prestazioni e le risorse del cluster, identificare eventuali problemi e risolverli in modo proattivo. La crescita esplosiva dei container nelle aziende di livello enterprise ha portato molti vantaggi a sviluppatori, DevSecOps e team IT di tutto il mondo. Tuttavia, la flessibilità e la scalabilità che Kubernetes offre nella distribuzione di applicazioni containerizzate

presenta anche nuove sfide. Poiché non esiste più una correlazione 1 a 1 tra un'applicazione e il server su cui viene eseguita, tenere traccia dello stato di salute delle applicazioni - astratte dai container e nuovamente astratte da Kubernetes - può essere scoraggiante senza gli strumenti adeguati. Kubernetes fornisce molte funzionalità integrate di monitoraggio, che consentono di rilevare e risolvere rapidamente eventuali problemi che possono verificarsi nel cluster, ad esempio, offre una dashboard integrata che fornisce una vista completa delle risorse del cluster, dei nodi e dei container in esecuzione. Tuttavia possono essere integrati altri strumenti di monitoraggio per migliorare ulteriormente l'efficacia del monitoraggio del cluster, la più diffusa è **Prometheus**[8].

6.1 Kubernetes metric server

Kubernetes Metric Server è un componente di Kubernetes che raccoglie le metriche delle risorse del cluster, come CPU e utilizzo della memoria, e le rende disponibili per l'analisi e la visualizzazione. È progettato per supportare le funzionalità di scaling automatico, monitoraggio e allerta all'interno di un cluster Kubernetes. Metric Server è costituito da due componenti principali: un server di raccolta dati e un database a serie temporali. Il server di raccolta dati raccoglie le metriche dalle risorse del cluster, come i pod, i nodi e i servizi, e le archivia nel database a serie temporali. Le metriche raccolte sono poi rese disponibili attraverso un endpoint API HTTP. Le metriche raccolte dal Metric Server possono essere utilizzate per monitorare le prestazioni del cluster e per automatizzare il provisioning delle risorse, ad esempio utilizzando la funzionalità di scaling automatico di Kubernetes. Inoltre, le metriche possono essere utilizzate per l'allerta in tempo reale in caso di problemi o di un'elevata domanda di risorse. Il Metric Server è un

componente fondamentale per il monitoraggio del cluster Kubernetes e può essere integrato con altri strumenti di monitoraggio, come Prometheus, per fornire una visibilità completa delle prestazioni del cluster. Tuttavia, va notato che il Metric Server non è adatto per monitorare applicazioni all'interno del cluster, per questo motivo spesso viene integrato con altre soluzioni di monitoraggio come appunto Prometheus.

6.2 Monitoring tramite Prometheus

Prometheus è un sistema di monitoraggio open-source che utilizza un'architettura decentralizzata per raccogliere, archiviare e analizzare metriche provenienti da diverse fonti. L'architettura di Prometheus è costituita da quattro componenti principali: server di raccolta dati, endpoint di esportazione, database a serie temporali e strumenti di interrogazione e visualizzazione. Il server di raccolta dati di Prometheus è il componente principale del sistema. Raccoglie le metriche da diverse fonti tramite endpoint HTTP esposti dalle applicazioni o da client specifici chiamati "exporter". Le metriche raccolte vengono quindi archiviate in un database a serie temporali. L'endpoint di esportazione è un'interfaccia HTTP utilizzata dalle applicazioni per esporre le metriche a Prometheus. Le metriche sono rappresentate come coppie chiave-valore e sono raggruppate in "famiglie di metriche" in base al loro nome e alle loro etichette. Il database a serie temporali di Prometheus è costituito da una serie di file di dati in cui le metriche raccolte vengono archiviate. Il database consente di interrogare e analizzare i dati nel tempo, consentendo di monitorare l'evoluzione delle metriche nel tempo e di individuare eventuali anomalie. Infine, gli strumenti di interrogazione e visualizzazione di Prometheus permettono di interrogare i dati del database e di visualizzarli sotto forma di grafici e tabelle. In particolare, il linguaggio

di query **PromQL** di Prometheus offre una sintassi flessibile per selezionare e aggregare le metriche archiviate. L'architettura decentralizzata di Prometheus consente di distribuire il server di raccolta dati in modo da gestire carichi di lavoro di grandi dimensioni. Inoltre, grazie alla sua flessibilità e alla sua vasta gamma di strumenti di monitoraggio e allerta, questo tool è diventato uno strumento di monitoraggio molto popolare nella comunità di sviluppatori e negli ambienti cloud-native.

Per monitorare il cluster Kubernetes tramite Prometheus, è necessario configurare i target di monitoraggio. I target di monitoraggio sono costituiti da endpoints specifici per le metriche, che possono essere esposti dai servizi o dai pod. In Kubernetes, i pod possono esporre endpoint delle metriche tramite un'interfaccia HTTP specifica che segue le specifiche del formato metriche di Prometheus. Per semplificare la configurazione dei target di monitoraggio, Kubernetes fornisce un'API di discovery dei servizi che permette a Prometheus di scoprire automaticamente i servizi che devono essere monitorati. Ciò significa che, una volta che il cluster Kubernetes è stato configurato correttamente, non è necessario configurare manualmente i target di monitoraggio per ciascun servizio. Inoltre, Prometheus fornisce un'ampia gamma di funzionalità di monitoraggio, tra cui il rilevamento delle anomalie, l'allerta in tempo reale e l'analisi delle prestazioni. È inoltre possibile utilizzare **Grafana**[9], uno strumento di visualizzazione dei dati open-source, per creare dashboard personalizzate per monitorare i dati di Prometheus in modo chiaro e intuitivo. In sintesi, Prometheus è un'opzione popolare e potente per il monitoraggio di cluster Kubernetes, in quanto offre una vasta gamma di funzionalità di monitoraggio e integrazione nativa con Kubernetes. Con Prometheus, gli utenti possono monitorare facilmente i loro cluster Kubernetes e avere una migliore comprensione delle prestazioni delle loro

applicazioni. Per monitorare un cluster Kubernetes tramite Prometheus, è necessario seguire alcuni passaggi di configurazione. In generale, i passaggi da seguire includono:

- Installare e configurare un server Prometheus nel cluster Kubernetes. Questo può essere fatto utilizzando uno strumento come Helm per semplificare il processo di installazione:
 - Creare un file di configurazione per Prometheus YAML;
 - Creare un deployment per Prometheus usando il file YAML;
 - Creare un servizio per il deployment di Prometheus;
- Configurare i target di monitoraggio per i servizi, i pod e i nodi del cluster Kubernetes. Ciò può essere fatto utilizzando il file di configurazione di Prometheus, che definisce quali endpoint di metriche devono essere monitorati:
 - Creare un file di configurazione per il scraping YAML;
 - Configurare il file YAML per indicare i pod e i container da monitorare;
 - Applicare la configurazione di scraping a Kubernetes;
- Utilizzare le query Prometheus per monitorare le metriche raccolte. Le query possono essere utilizzate per ottenere informazioni sulle prestazioni dei servizi, dei pod e dei nodi, ad esempio la quantità di utilizzo della CPU e della memoria:
 - Creare un file di configurazione per la regola di allerta YAML;
 - Configurare il file YAML per specificare le condizioni di allerta;
 - Applicare la configurazione della regola di allerta a Kubernetes;

- Utilizzare Grafana per creare dashboard personalizzate per visualizzare le metriche raccolte. Grafana fornisce una vasta gamma di opzioni di visualizzazione dei dati, come grafici a linee, grafici a barre e diagrammi a torta;
- Monitorare e testare il sistema:
 - Verificare che i dati siano correttamente raccolti da Prometheus;
 - Verificare che le regole di allerta funzionino correttamente;

6.3 Esempio file di configurazione

```

global:
  scrape_interval: 15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: kubernetes-apisservers
    kubernetes_sd_configs:
      - role: endpoints
    scheme: https
    tls_config:
      ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      insecure_skip_verify: true
    bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
    relabel_configs:
      - source_labels: [__meta_kubernetes_namespace, __meta_kubernetes_service_name,
        __meta_kubernetes_endpoint_port_name]
        action: keep
        regex: default;kubernetes;https

  - job_name: kubernetes-nodes
    kubernetes_sd_configs:
      - role: node
    relabel_configs:
      - source_labels: [__address__]
        action: replace
        target_label: __address__
        regex: (.+):10250
        replacement: ${1}:9100
      - source_labels: [__address__, __meta_kubernetes_node_label_kubernetes_io_hostname]
        action: replace
        target_label: instance
        regex: (.+):.*
        replacement: ${2}
        separator: -

  - job_name: kubernetes-pods
    kubernetes_sd_configs:
      - role: pod
    relabel_configs:
      - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
        action: keep
        regex: true
      - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scheme]
        action: replace
        target_label: __scheme__
        regex: (https?)
      - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_path]
        action: replace
        target_label: __metrics_path__
        regex: (.+)
      - source_labels: [__address__, __meta_kubernetes_pod_annotation_prometheus_io_port]
        action: replace
        target_label: __address__
        regex: ([^:]+)(?::\d+)?(\d+)?
        replacement: $1:$2
        separator: :
      - source_labels: [__meta_kubernetes_namespace]
        action: replace
        target_label: kubernetes_namespace
      - source_labels: [__meta_kubernetes_pod_name]
        action: replace
        target_label: kubernetes_pod_name

```

Figura 11: Esempio Configurazione Prometheus

- `global`: questa sezione definisce le impostazioni globali per Prometheus, come la frequenza di raccolta delle metriche (`scrape_interval`) e la frequenza di valutazione delle regole di allarme (`evaluation_interval`).
- `scrape_configs`: questa sezione definisce le configurazioni di scraping per le diverse fonti di metriche. In questo esempio, ci sono tre configurazioni di scraping, una per gli API server di Kubernetes, una per i nodi di Kubernetes e una per i pod di Kubernetes.
- `job_name`: questo parametro definisce il nome del job, ovvero il nome del target da monitorare.
- `kubernetes_sd_configs`: questa sezione definisce la configurazione di discovery di servizi di Kubernetes. In questo esempio, si utilizza il discovery degli endpoints per gli API server, dei nodi e dei pod.
- `scheme`: questo parametro definisce il protocollo di trasporto da utilizzare per raggiungere gli endpoint. In questo esempio, viene utilizzato il protocollo HTTPS.
- `tls_config`: questa sezione definisce la configurazione per il protocollo TLS, come ad esempio il file CA da utilizzare per la verifica del certificato del server.
- `bearer_token_file`: questo parametro definisce il file contenente il token di accesso necessario per autenticarsi al servizio di Kubernetes.
- `relabel_configs`: questa sezione definisce le configurazioni di rielaborazione delle etichette delle metriche. In questo esempio, si utilizzano diverse configurazioni per manipolare le etichette e renderle conformi alle convenzioni di naming delle metriche di Kubernetes.

- `source_labels`: questo parametro definisce l'etichetta di origine utilizzata per la rielaborazione. In questo esempio, vengono utilizzate diverse etichette di origine per rielaborare le etichette delle metriche.
- `action`: questo parametro definisce l'azione da eseguire sulla rielaborazione delle etichette. In questo esempio, vengono utilizzate diverse azioni come "keep", "replace" e "drop".
- `regex`: questo parametro definisce l'espressione regolare utilizzata per la rielaborazione delle etichette. In questo esempio, le espressioni regolari vengono utilizzate per estrarre parti delle etichette originali e manipolare le etichette di destinazione.
- `target_label`: questo parametro definisce il nome dell'etichetta di destinazione per la rielaborazione. In questo esempio, vengono utilizzati diversi nomi di etichetta di destinazione come "instance", "kubernetes_namespace" e "kubernetes_pod_name".
- `replacement`: questo parametro definisce la stringa di sostituzione da utilizzare nella rielaborazione delle etichette. In questo esempio, vengono utilizzate diverse stringhe di sostituzione per manipolare le etichette di destinazione.
- `separator`: questo parametro definisce il separatore da utilizzare nella rielaborazione delle etichette. In questo esempio, viene utilizzato il carattere "-" come separatore.

6.4 Kubernetes Demo

Nell'implementazione effettuata è stato utilizzato Kubernetes per realizzare il deploy una coda di messaggi basata su Beanstalkd[7], insieme ad un producer e dei consumer scritti in Python.

Beanstalkd è un sistema di messaggistica open-source sviluppato in C che permette la comunicazione asincrona tra diverse applicazioni distribuite. È basato su un'architettura client-server e funziona con un'organizzazione a code (queue-based), dove i messaggi vengono inseriti in una coda di attesa e successivamente processati dai lavoratori (workers) che accedono alla coda. Il producer implementato si avvale delle API messe a disposizione dal noto social network **Reddit**[15] e si occupa di leggere gli ultimi post pubblicati sulla piattaforma e scriverli sulla coda.

I vari consumer invece, leggono dalla coda i messaggi presenti, consumandoli e applicano ad essi l'algoritmo di sentiment analysis **VADER**[16] che restituisce in output il sentiment del post appena consumato.

Inoltre, è presente un server Prometheus per il monitoring del sistema agganciato ad un metric server di Kubernetes. La totalità dei container è hostata su un registry remoto come docker hub per quelli sviluppati appositamente e Helm per quanto riguarda le immagini dei servizi come Prometheus.

Questo semplice esempio giocattolo permette di capire quanto sia semplice grazie alla containerizzazione e in particolare a Kubernetes scalare un servizio che su grosse moli di dati sarebbe impossibile da eseguire solo tramite un singolo consumer, inoltre, rende trasparente l'importanza di realizzare interlacciamento lasco tra i vari processi di un'architettura a basata sui microservizi e il monitoring che va effettuato su di essa.

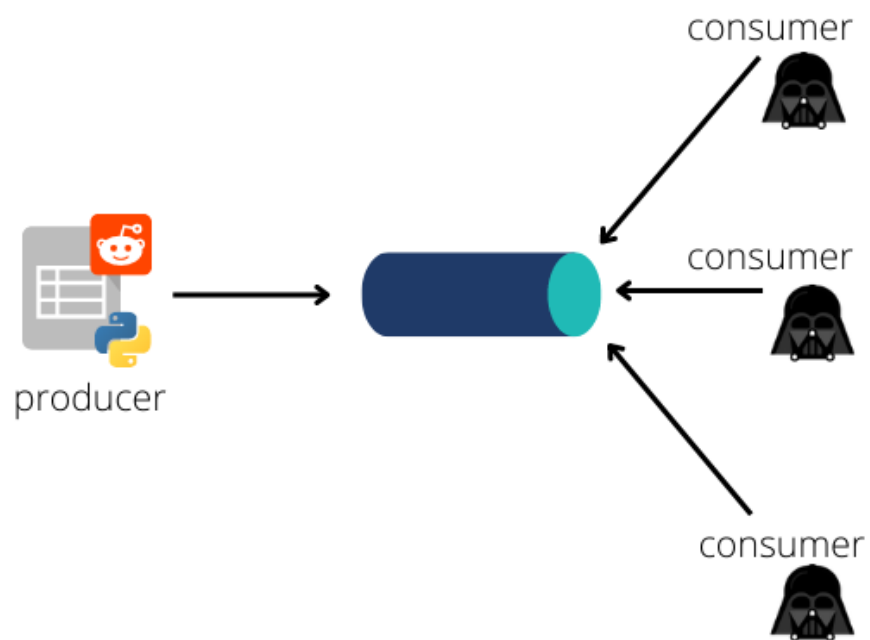


Figura 12: flowchart della demo

7 Conclusioni

La containerizzazione è un'innovazione importante che ha rivoluzionato il modo in cui le applicazioni vengono sviluppate, distribuite e gestite. Con l'aumento della complessità delle applicazioni moderne, la necessità di un'infrastruttura scalabile e affidabile è diventata sempre più importante. In questo contesto, Kubernetes è emerso come una piattaforma leader per la gestione di container su larga scala. In particolare, il monitoring di un cluster Kubernetes è essenziale per garantire che le applicazioni eseguite sui container siano disponibili e performanti. Il monitoring consente di identificare tempestivamente eventuali problemi e di agire di conseguenza per ripristinare la disponibilità delle applicazioni. Inoltre, il monitoring permette di ottimizzare l'uso delle risorse e di identificare eventuali bottleneck nella piattaforma.

Questo lavoro di approfondimento si pone come introduzione a quello che si presenta essere uno scenario molto ampio e ancora in via di sviluppo, che si rivela tuttavia di importanza direttamente proporzionale a quella della crescita degli applicativi e delle infrastrutture che li ospitano, per garantire sempre gli obiettivi principali quali il supporto continuo e l'abbattimento dei costi.

Riferimenti bibliografici

- [1] Containerization technologies: taxonomies, applications and challenges
Ouafa Bentaleb, Adam S. Z. Belloum, Abderrazak Sebaa, Aouaouche
El-Maouhab¹
- [2] Containerization for creating reusable model code Manuela Vanegas Fer-
ro, Allen Lee¹, Calvin Pritchard, C. Michael Barton, and Marco A.
Janssen; School of Complex Adaptive Systems, Arizona State Univer-
sity, USA Inuvialuit Regional Corporation, Canada; School of Human
Evolution and Social Change, Arizona State University, USA; School of
Sustainability, Arizona State University, USA
- [3] Docker, <https://www.docker.com>
- [4] Kubernetes, <http://kubernetes.io/>
- [5] Docker Compose, <https://docs.docker.com/compose/>
- [6] Express.js, <https://expressjs.com/it/>
- [7] Beanstalkd, <https://beanstalkd.github.io/>
- [8] Prometheus, <https://prometheus.io/>
- [9] Grafana, <https://grafana.com/>
- [10] Docker Swarm, <https://docs.docker.com/engine/swarm/>
- [11] Helm, <https://helm.sh/>
- [12] Docker Hub, <https://hub.docker.com/>
- [13] Aws Containerization, <https://aws.amazon.com/it/what-is/containerization/>

- [14] Red Hat Containerization <https://www.redhat.com/en/topics/cloud-native-apps/what-is-containerization>
- [15] Reddit, <https://www.reddit.com/>
- [16] Vader, <https://www.nltk.org/modules/nltk/sentiment/vader.html>