University of Modena and Reggio Emilia

Department of Engineering

---

**Bachelor's Degree Course in**

**Computer Engineering**

# Container technologies survey trough Docker

**First thesis supervisor:**

Prof. Francesco Guerra

**Candidate:**

Bilel Arfaoui

Academic Year 2021/2022

«*I found a mirror under the table.*»
Link.

**Sommario**

Negli ultimi anni, la containerizzazione ha rivoluzionato il modo in cui le applicazioni vengono sviluppate, distribuite e gestite. Questa tesi esplora il percorso evolutivo dalle macchine virtuali ai container, evidenziando le differenze architetturali, i vantaggi in termini di efficienza e portabilità, e le nuove sfide introdotte.

Il cuore dell'elaborato è dedicato a Docker, la piattaforma di containerizzazione più diffusa, di cui vengono analizzate le componenti fondamentali, il funzionamento delle immagini e dei container, e gli strumenti per la gestione e l'automazione dei flussi di lavoro.

Viene inoltre trattato il tema dell'orchestrazione dei container, confrontando Docker Swarm e Kubernetes, con particolare attenzione alle loro architetture, funzionalità e casi d'uso.

Infine, si presentano alcune estensioni e strumenti complementari all'ecosistema Docker, come Docker Compose, Docker Hub e soluzioni per la sicurezza, seguiti da una panoramica su reali casi d'utilizzo nel contesto dello sviluppo software moderno, della CI/CD e della migrazione di applicazioni legacy.

Questa tesi si propone quindi come una guida completa e aggiornata all'universo Docker, evidenziandone il ruolo chiave nei moderni ambienti DevOps e cloud-native.

# Indice

# 1. Introduzione

Il presente elaborato studia la piattaforma di virtualizzazione Docker avendo i seguenti obiettivi:

- Dare una panoramica di base sul deployment dei servizi informatici e l'evoluzione nel corso della storia.
- Fornire un'introduzione alla tecnologia di Docker, focalizzandosi sull'architettura della piattaforma, unità di elaborazione fondamentali e strumenti principali che assistono il funzionamento della piattaforma.
- Mettere luce le sfide che gli utenti di Docker devono affrontare. Si parlerà della gestione di progetti di larga scala attraverso tool d'orchestrazione (Swarm e Kubernetes) e la sicurezza nell'infrastruttura, specificando i rischi e le buone pratiche per contrastarli.
- Mostrare i casi d'uso e applicazioni di Docker nel settore, specificando i vantaggi che porta la piattaforma.

# 2. Tipologie di Deployment

Le fonti da cui il contenuto di questo capitolo è tratto, sono indicate nel capitolo 8 Bibliografia.

Nel corso degli anni, il deployment delle applicazioni è evoluto, con l'obiettivo di distribuire applicazioni in modo leggero, efficiente e con costi ridotti.

### 2.1 Deployment Tradizionale

Inizialmente le applicazioni erano eseguite in un server fisico, condividendo le risorse.

Questo tipo di approccio era fonte di problemi, perché si poteva verificare il caso in cui un'applicazione impiegasse la maggior parte delle risorse, danneggiando le prestazioni delle altre applicazioni. Una soluzione è stata quella di eseguire ogni applicazione in server fisici diversi, quindi costosa e con risorse sottoutilizzate. Questo però ha generato un grande problema per le organizzazioni, dove il costo della manutenzione di diversi server fisici tendeva ad accumularsi.

### 2.2 Deployment Virtualizzato

La virtualizzazione è una tecnologia che consente di creare uno o più ambienti di elaborazione simulati da un unico pool di risorse, comportando una efficienza maggiore.

Cioè si astraggono le risorse hardware e le si rendono disponibili al software sotto forma di risorse virtuali. La virtualizzazione permette una maggiore efficienza dell'uso delle risorse.

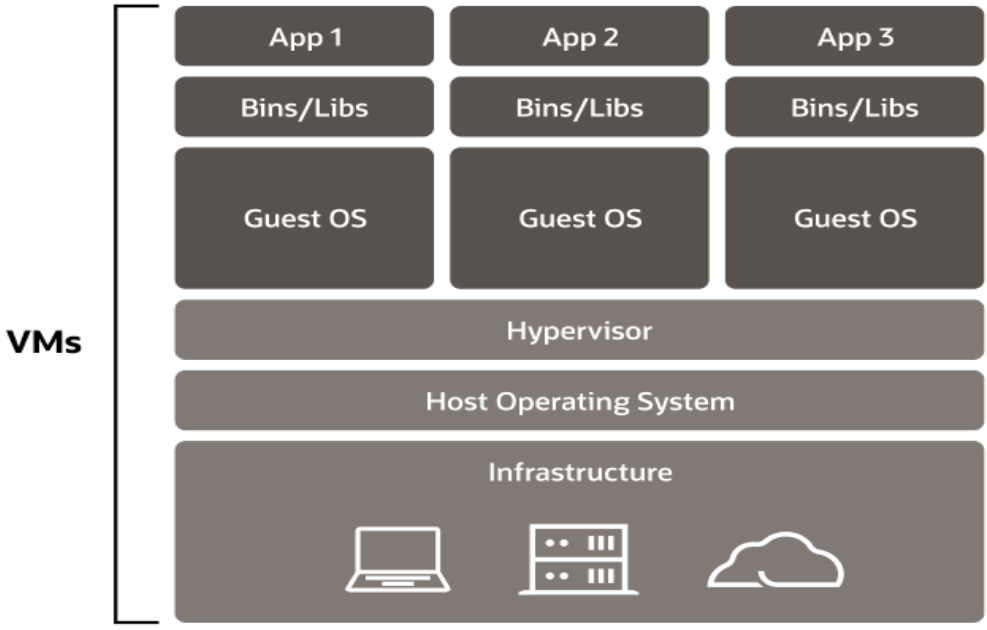


**Figura 1**: esempio grafico del deployment virtualizzato

**Fonte**: https://www.oracle.com/it/cloud/cloud-native/container-registry/what-is-docker/

Nasce negli anni '60 come soluzione al deployment tradizionale, e consiste nell'esecuzione delle applicazioni all'interno delle macchine virtuali, software che si comportano come computer fisici e offrono le stesse funzionalità, simulando tutte le componenti hardware(Disco fisso, RAM, CPU, Scheda di rete), quindi permettevano installare interi sistemi operativi e applicazioni.

Il deployment virtualizzato (come indicato nella figura 1), permette di eseguire più macchine virtuali su un'unica CPU, grazie a un processo chiamato Hypervisor che è incaricato di creare e gestire più macchine virtuali, allocando le risorse virtuali in base alle esigenze. L'Hypervisor agisce

come un layer (strato) software aggiuntivo che separa le macchine virtuali dal computer fisico.

### 2.3 Deployment containerizzato

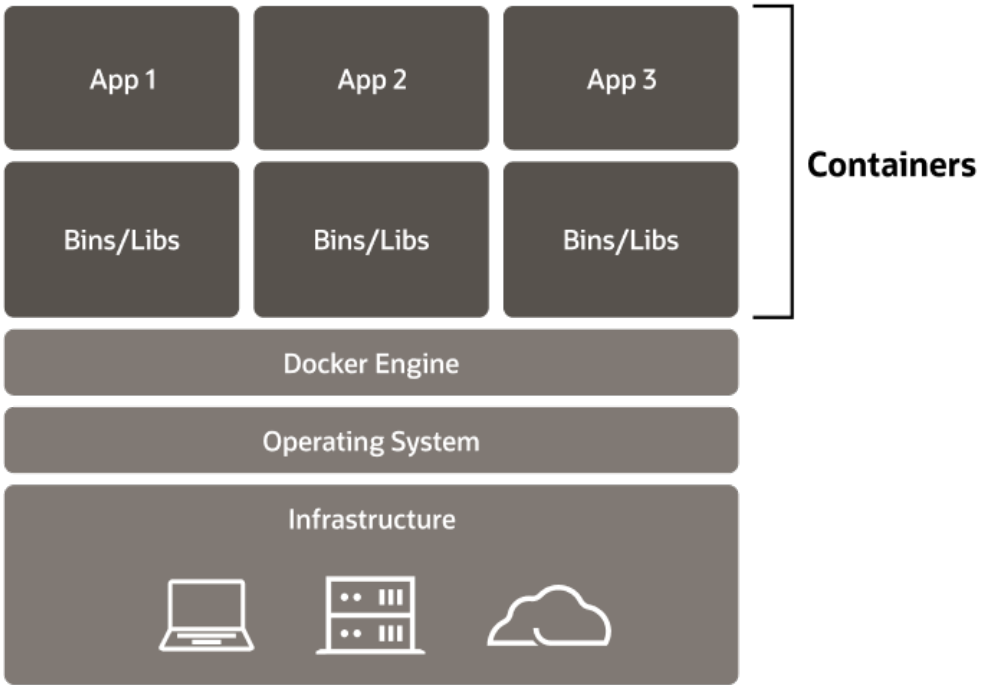


**Figura 2**: Rappresentazione del deployment con i container

**Fonte**: https://www.oracle.com/it/cloud/cloud-native/container-registry/what-is-docker/

Consiste nell'uso dei container, unità software eseguiti dal sistema operativo dell'host fisico, simili alle macchine virtuali, con la differenza che i container prendono un approccio più "leggero", contenendo solo i componenti necessari perché l'applicazione all'interno possa essere eseguita in modo corretto ed efficiente. I container condividono il kernel del sistema host con altri container.

Per la corretta esecuzione delle applicazioni un container può contenere:

- codice
- runtime
- tool di sistema
- librerie di sistema
- dependecies del software

I container isolano il software e permettono l'esecuzione delle applicazioni in qualsiasi ambiente informatico, indipendentemente dalla struttura sottostante, senza arresti anomali e con prestazioni coerenti tra le macchine.

Grazie all'impiego dei container gli sviluppatori non devono concentrarsi a debug o riscrittura del codice per adattare l'applicazione ai vari ambienti informatici, permettendo una transizione perfetta ed efficiente da un ambiente all'altro.

### 2.4 Come sono costruiti i container

I container sono creati tramite l'esecuzione di un file chiamato immagine, consistono in file immutabili(read-only) con all'interno codice eseguibile che può creare container in un ambiente informatico, non contengono sistemi operativi.

Un'immagine contiene il necessario per l'esecuzione del container:

- librerie di sistema
- tool di sistema
- utilities
- comandi
- etc..

Le immagini impiegano un file system incrementale (overlay), organizzato a livelli (layer) impilati uno sopra l'altro e  forniscono solo permessi di lettura. Ogni livello dipende dal livello immediatamente inferiore. Quando un container viene eseguito viene aggiunto in cima, un livello con permessi di scrittura chiamato “container layer” (i layer inferiori rimangono immutati), in questo livello vengono salvati tutti i cambiamenti fatti all'interno del container in esecuzione. Ogni livello rappresenta lo stato del file system dopo una certa operazione in un momento specifico.

Un container può essere salvato come immagine, e il livello superiore diventerà immutabile permettendo solo la lettura.

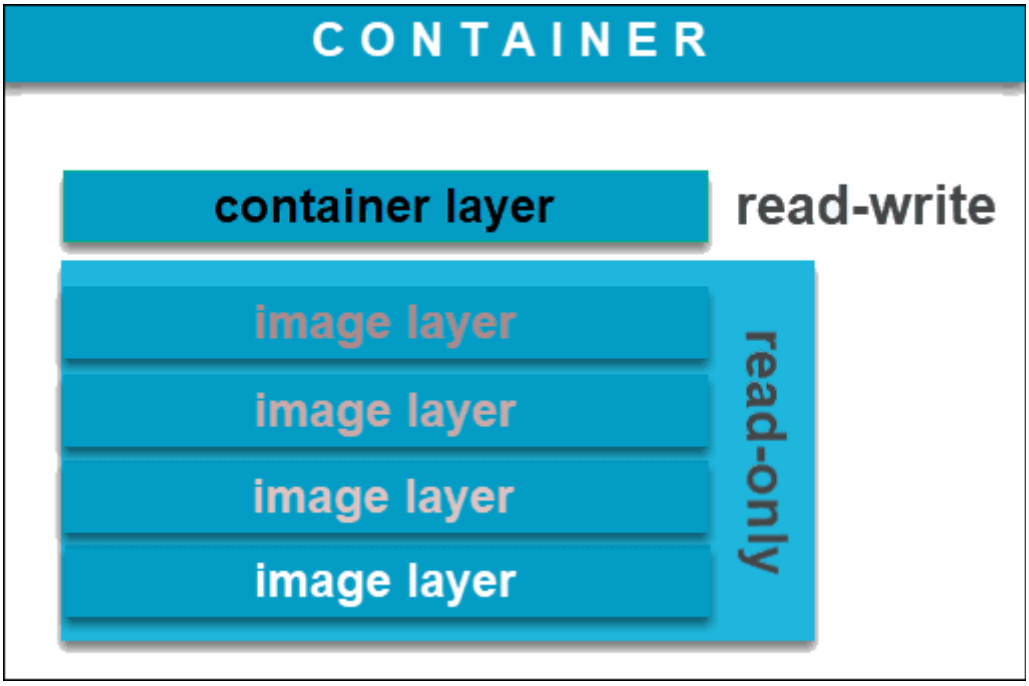


**Figura 3**: Struttura del file system di un container, mostra gli strati dell'immagine read-only e lo strato del container  in esecuzione read-write **Fonte**:https://phoenixnap.com/kb/docker-image-vs-container

Nel container la strategia di scrittura è di tipo copy-on-write, i file del layer inferiore sono accessibili per la sola lettura, quando è richiesta un'operazione di modifica, verrà creata una copia aggiornata nel layer superiore e l'originale rimarrà immutato (come raffigurato nella figura 4).

Un livello resta in cima del file system esistente, combina gli alberi di layer superiori a quelli inferiori e li presenta come un unico direttorio. I layer inferiori rimangono immutati e ogni layer superiore aggiunge solo la differenza(diff) dal layer inferiore.

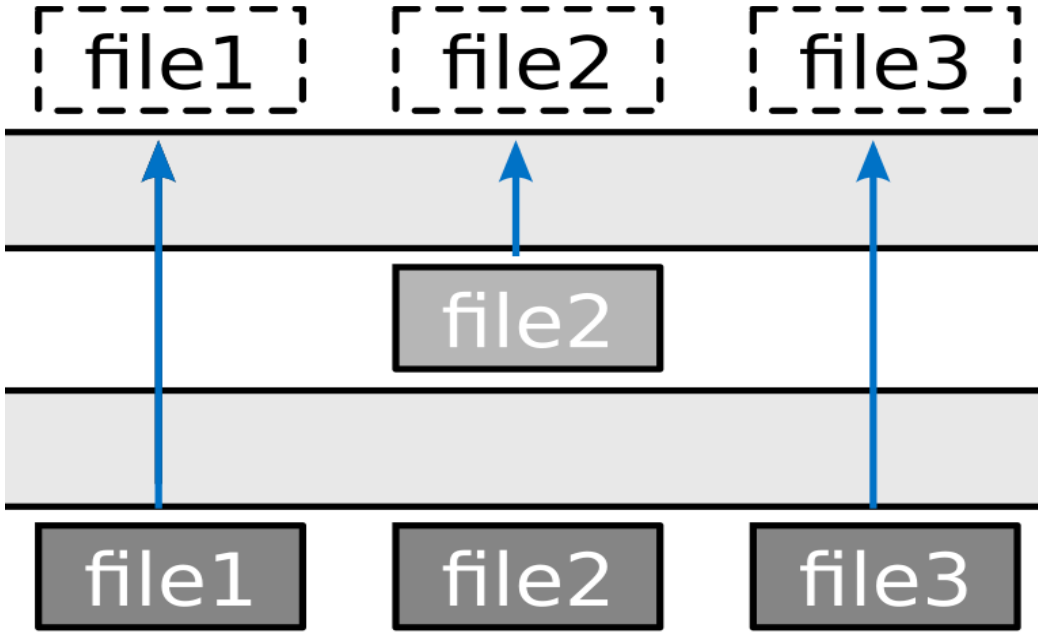


**Figura 4**: Rappresentazione grafica del file system incrementale (overlay)

**Fonte**: https://opensource.com/article/21/8/container-image

Le immagini vengono costruite tramite l'aggiunta di livelli (layer) al di sopra del Parent o Base:

- **Base (o Scratch):** Immagine vuota per la creazione da "zero", sconsigliato per utenti principianti.
- **Parent :** Immagine preconfigurata con funzioni di base, come tool per la gestione del container o librerie. In alcuni casi si possono trovare immagini Parent preconfigurati per un linguaggio di programmazione specifico.

Questo concetto di Base e Parent promuove il riutilizzo delle componenti, in modo tale che l'utente non debba creare immagini da zero per ogni progetto, comportando un risparmio di tempo e spazio di memoria riducendo la mole dei container. Per questo motivo il peso delle immagini è minimo (esistono eccezioni) solitamente intorno alla decine di bytes.

Numerose immagini Parent sono distribuite su DockerHub o in altri repository pubblici pronti all'uso.

In Docker è possibile definire immagini da zero, tramite la scrittura di codice all'interno di un file in formato YAML, denominato "Dockerfile", una volta compilato il file, si avrà in output l'immagine. Parleremo di questo concetto nei capitoli seguenti

### 2.5 Distribuzione delle immagini

Le immagini una volta "create" vengono salvate nel registro di una repository privata o pubblica, come Docker Hub. Gli autori delle immagini pubblicano le immagini nella repository (push) e gli utenti scaricano l'immagine(pull) quando vogliono eseguire un container.

L'uso di immagini container, può portare al rischio di imbattersi a immagini pubbliche corrotte, false e fraudolente a volte fingendosi di essere distributori ufficiali.

Esistono servizi come Docker Content Trust prevedono la firma digitale in modo da verificare che le immagini scaricate da repository pubbliche siano originali e non alterate.

Però questo aggiunta di verifica di autenticità non previene totalmente la creazione e distribuzione di Malware.

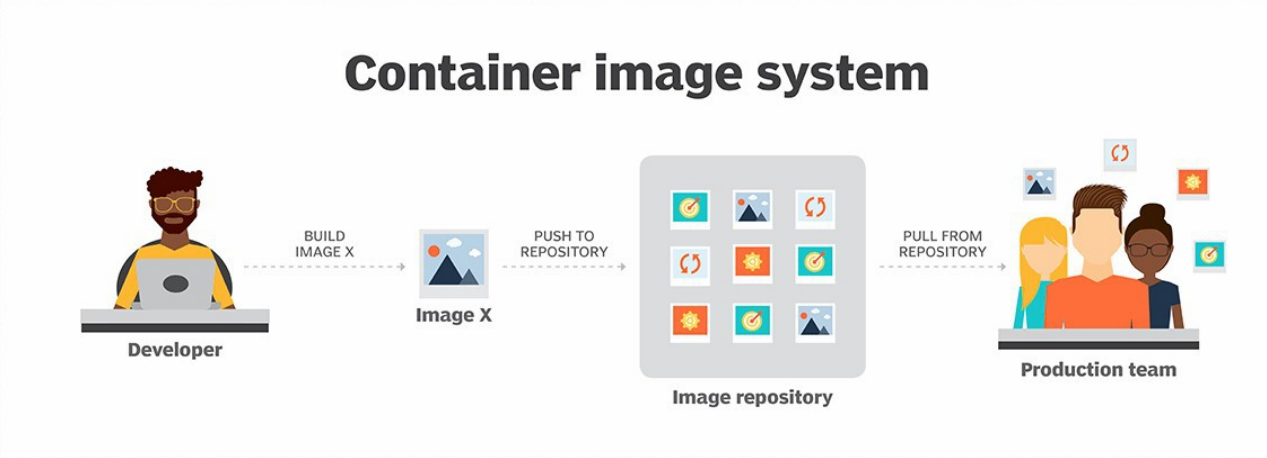


**Figura 5**: Processo di distribuzione delle immagini, dallo sviluppatore al consumatore

**Fonte**: https://www.techtarget.com/searchitoperations/definition/container-image

### 2.6 Breve storia dei container

Il concetto di container era nato intorno agli anni '70, impiegato nei sistemi UNIX per isolare servizi e applicazioni in modo tale da non poter interferire con altri processi, per quanto vantaggiosi per lo sviluppo e distribuzione delle applicazioni, non erano ancora portatili.

Negli anni 2000, la tecnologia dei container subì una crescita, all'inizio fu introdotto Free BSD Jails, che permetteva la partizione dei sistemi FreeBSD(sistema operativo open source basato su Linux distribuito dalla Berkeley Software Distribution) in mini-sistemi chiamati jails, questo sistema è stato poi raffinato nel 2001 con Linux Vserver, che poteva partizionare file system, memoria e indirizzi di rete.

Nel 2006 la tecnologia dei container, progredì di nuovo grazie all'introduzione di Control Groups (cgroups), una nuova funzione del kernel Linux per l'allocazione delle risorse come memoria e CPU.

Nel 2008 nacque LXC(LinuX Containers), la versione più stabile e affidabilità della tecnologia di container in quel momento, LXC operava al livello di sistema operativo e permetteva di eseguire container su una macchina con reale con kernel Linux.

La qualità di LXC porto alla nascita di tecnologie basate su di essa, come Warden nel 2011 e Docker nel 2013.

Docker presentava una GUI semplice da usare, ed era capace di eseguire applicazioni che necessitavano diversi sistemi operativi in uno solo.

Grazie a queste qualità Docker ebbe successo con 100 milioni di download in un anno.

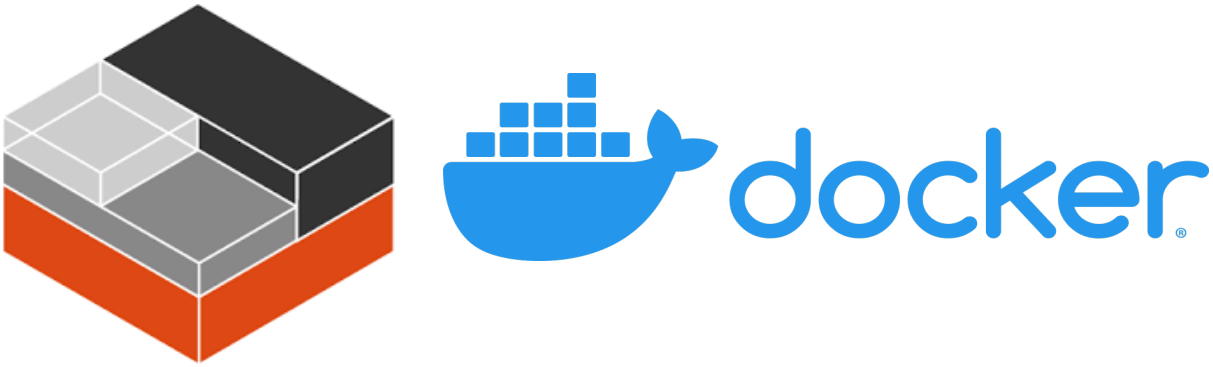

**Figura 6**: Loghi di LXC e Docker **Fonte Logo LXC**: https://en.wikipedia.org/wiki/LXC
**Fonte Logo Docker:** https://francoconidi.it/installare-docker-su-debian-11-e-syslinuxos/docker-logo/

L'interesse nello sviluppo e implementazione delle applicazioni containerizzate continuò crescere anche negli anni successivi.

Nel 2016 Microsoft cominciò a supportare l'esecuzione nativa dei container(inizialmente basati su Linux) su Windows. Microsoft ebbe una grande influenza sul settore dei container, con l'uscita dei container Process e Hyper-V.

Nel 2017 Docker cominciò a supportare Kubernetes, che era una tecnologia per l'orchestrazione di container molto efficiente.

Al giorno d'oggi la tecnologia dei container è in continua crescita senza segno di fermarsi.

### 2.7 Container vs Macchine virtuali

In confronto alle macchine virtuali,  i container contengono solo ciò che è necessario alla corretta esecuzione delle applicazioni, di conseguenza sono più leggeri e il processo di creazione/distruzione dei container è semplice e rapido. I container inoltre astraggono lo spazio utente (quindi il kernel è condiviso), e sono compatibili con varietà di ambienti informatici che promuove la portabilità e consistenza del carico di lavoro nell'ambiente di sviluppo. Mentre le macchine virtuali impiegano molte più risorse di computazione e memoria, spesso utilizzando interi sistemi operativi.

In questo momento però le due tecnologie non necessariamente sono in competizione tra di loro, infatti esistono casi d'uso dove è previsto l'utilizzo congiunto di esse. Come l'impiego di applicazioni container all'interno delle macchine virtuali per applicazioni che richiedono un'infrastruttura solida, che sfrutti al meglio la virtualizzazione delle risorse necessarie per il supporto dei container.

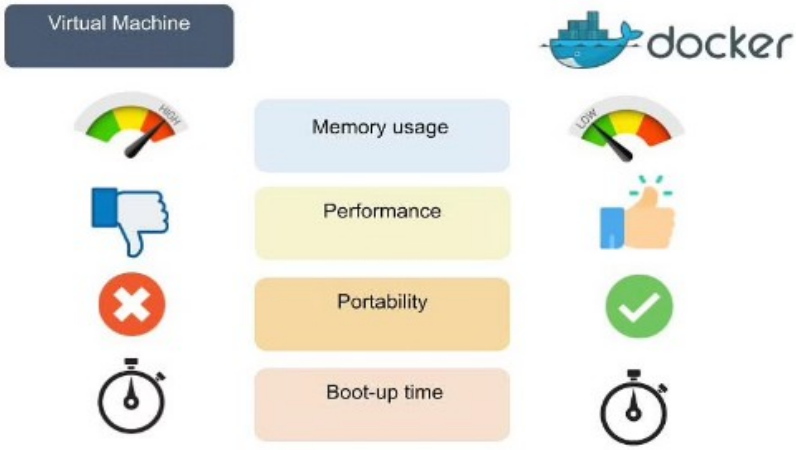


**Figura 7**: Schema riassuntivo del confronto tra container e macchine virtuali
**Fonte**:https://bryanavery.co.uk/docker-basic-concepts-components-and-advantages/

In conclusione, i container non sostituiscono necessariamente le macchine virtuali in tutti i casi d'uso, è consigliata una valutazione per determinare quale approccio è il più adatto.

### 2.8 Settori che impiegano la containerizzazione

Molte aziende stanno cominciando ad impiegare la strategia della containerizzazione,  grazie ai cicli di sviluppo più brevi, quindi spese minori,e  risparmio delle risorse e downtime non pianificati sempre più minori.

Esempi di settori che beneficiano dall'impiego di container:

- **Retail**: Le piccole e medie imprese, necessitano di servizi cloud per creare e gestire siti di eCommerce. Tecnologie di containerizzazione come Docker, entrano in gioco quando serve una grande quantità di container per l'esecuzione di più  processi in un breve lasso di tempo.
- **Internazionale:** necessità di dover fornire un servizio con affidabilità e prestazioni coerenti in ogni area geografica
- **Assistenza sanitaria**:Grazie alla flessibilità e portabilità della tecnologia, il personale curante può prendere decisioni più informate. Un esempio è la distribuzione a più ospedali di software dedicato all'imaging, inoltre grazie al testing e l'integrazione continua (riferimento al CI/CD) si avrà sempre la versione più recente del servizio.
- **Media e intrattenimento**: Servizi come Netflix necessitano di strumenti per la distribuzione di contenuto a una grande quantità di utenti. La tecnologia dei container, permette di creare e gestire infrastrutture sofisticate in modo efficiente. Grazie a tecnologie come Docker è possibile definire le politiche di aggiornamento e self-healing automatizzate.

# 3. Docker

Le fonti da cui il contenuto di questo capitolo è tratto, sono indicate nel capitolo 8 Bibliografia.

Docker è una piattaforma che comprende un insieme di prodotti per lo sviluppo, esecuzione e distribuzione di applicazioni e microservizi containerizzati.

I vari componenti di Docker si possono trovare compresi nell'applicazione Docker Desktop.

## 3.1 Componenti principali di Docker

### 3.1.1 Software

L'engine Docker è definibile come il nucleo di Docker, cioè un'applicazione installata sopra una macchina chiamata Docker Host, si basa su un'architettura di tipo client-server.

Ha tre componenti principali:

- **server:** daemon che rimane in ascolto
- **l'api rest:** permette la comunicazione tra client e server
- **client**

Il programma **client** denominato **docker**, fornisce un interfaccia a linea di comando(CLI) che permette d'interagire con il daemon, questo programma è la modalità principale per il quale l'utente comunica con il server.

Il client comunica con il server tramite l'API REST, ogni comando fatto a linea di comando viene inviato dal client al daemon che li esegue.

Il client può far interagire con gli oggetti(registri, immagini, container....) tramite comandi come **pull**.

Un client può comunicare con più di un daemon.

Il **daemon docker** chiamato **dockerd,** è un processo di background che rimane costantemente in ascolto per richieste attraverso l'API, il suo compito è gestire gli oggetti ed eseguire le operazioni richieste dal client.

I daemon nei casi richiesti, possono comunicare tra di loro(gestione di servizi...).

Il daemon può essere avviato manualmente con il comando **dockerd.**

```
$ dockerd

INFO[0000] +job init_networkdriver()
INFO[0000] +job serveapi(unix:///var/run/docker.sock)
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
```

**Figura 8**: Esecuzione del comando dockerd nel CLI **Fonte**:https://docs.docker.com/config/daemon/

Una volta installato ed avviato Docker, **dockerd** và in esecuzione con la configurazione di default, e può essere configurato manualmente, pratica consigliata perché semplifica il processo di diagnostica e debug in caso di problemi.

La configurazione manuale può essere eseguita in 2 modi:

1. **file di configurazione tramite file JSON:**

```
{
  "debug": true,
  "tls": true,
  "tlscert": "/var/docker/server.pem",
  "tlskey": "/var/docker/serverkey.pem",
  "hosts": ["tcp://192.168.59.3:2376"]
}
```

**Figura 9**: Esempio di file JSON **Fonte**:https://docs.docker.com/config/daemon/

2. **l'uso di flag all'avvio del daemon**

```
$ dockerd --debug \
  --tls=true \
  --tlscert=/var/docker/server.pem \
  --tlskey=/var/docker/serverkey.pem \
  --host tcp://192.168.59.3:2376
```

**Figura 10**: Esempio dell'uso dei flag **Fonte**:https://docs.docker.com/config/daemon/

In entrambi gli esempi succede la stessa cosa, il daemon viene avviato in modalità debug, usano il TLS come protocollo di crittografia, e rimangono in ascolto per il traffico che và verso 192.168.59.3 nella porta 2376.

La prima opzione è consigliata perché mantiene tutte le configurazioni salvate in un singolo posto.

I file di configurazione vengono salvati all'interno di una directory di default, ma si può configurare in modo tale che vengano salvati altrove.

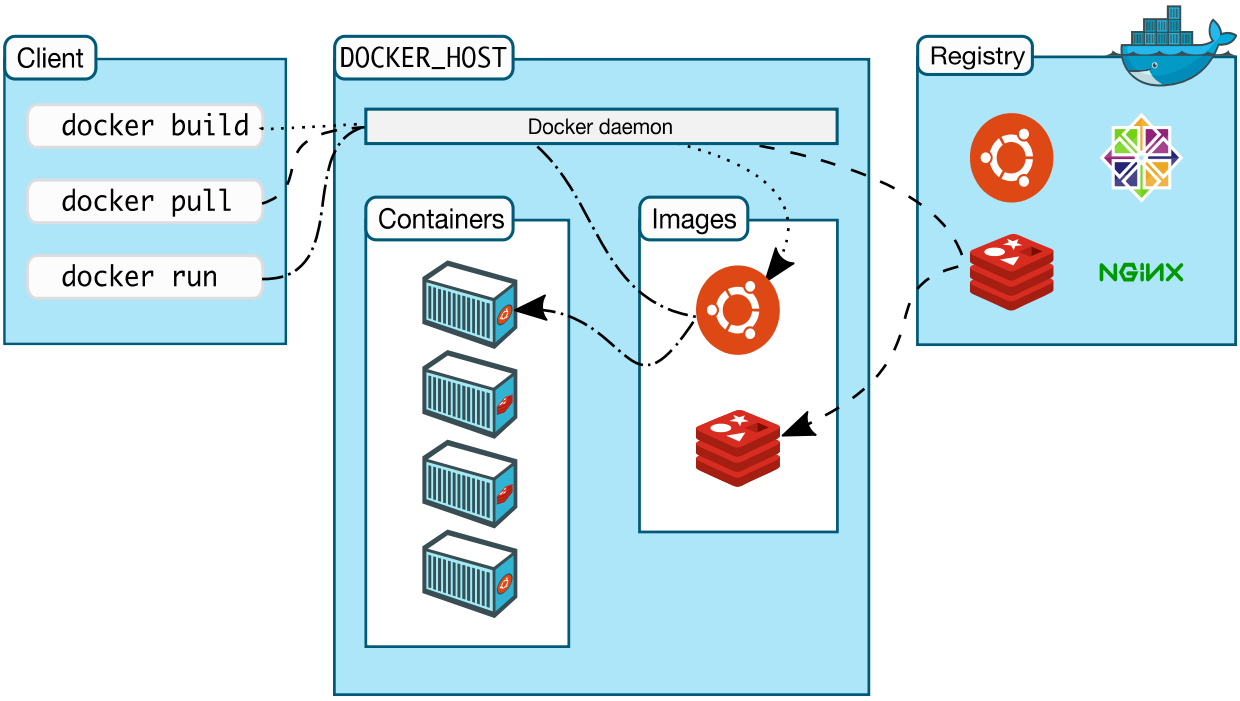


**Figura 11**: Schema che rappresenta l'architettura di Docker **Fonte**:https://docs.docker.com/get-started/overview/

### 3.1.2 OGGETTI

#### 3.1.2.1 Immagini

Gli oggetti di Docker sono le entità usate per assemblare un ambiente containerizzato per Docker.

Le **immagini Docker** sono modelli read-only con all'interno istruzioni per la creazione di un container Docker.

Solitamente le immagini Docker sono basate su altre immagini Docker prese da un registro pubblico (o privato), con impostazioni aggiuntive. Oppure si possono creare immagini proprie tramite un dockerfile, cioè file di testo con istruzioni all'interno che definisce i passi per la creazione di un'immagine.

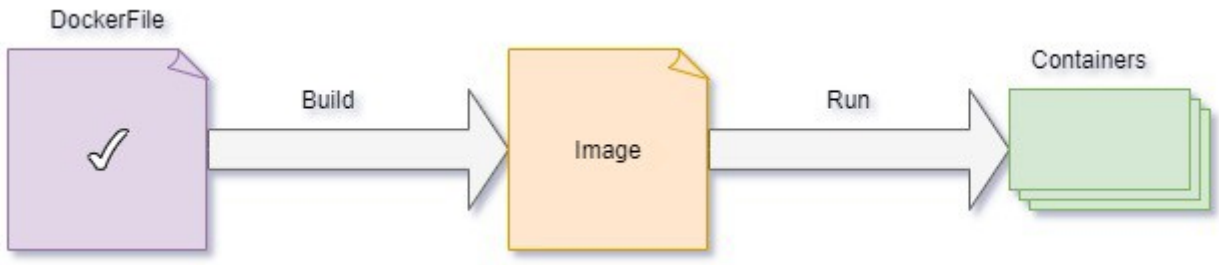


**Figura 12**: Processo di build, dal dockerfile al container **Fonte**:https://www.geeksforgeeks.org/what-is-docker-images/

Per costruire immagini Docker proveniente da un dockerfile con il comando "docker build".

Ogni istruzione nel dockerfile crea un "layer" nell'immagine.

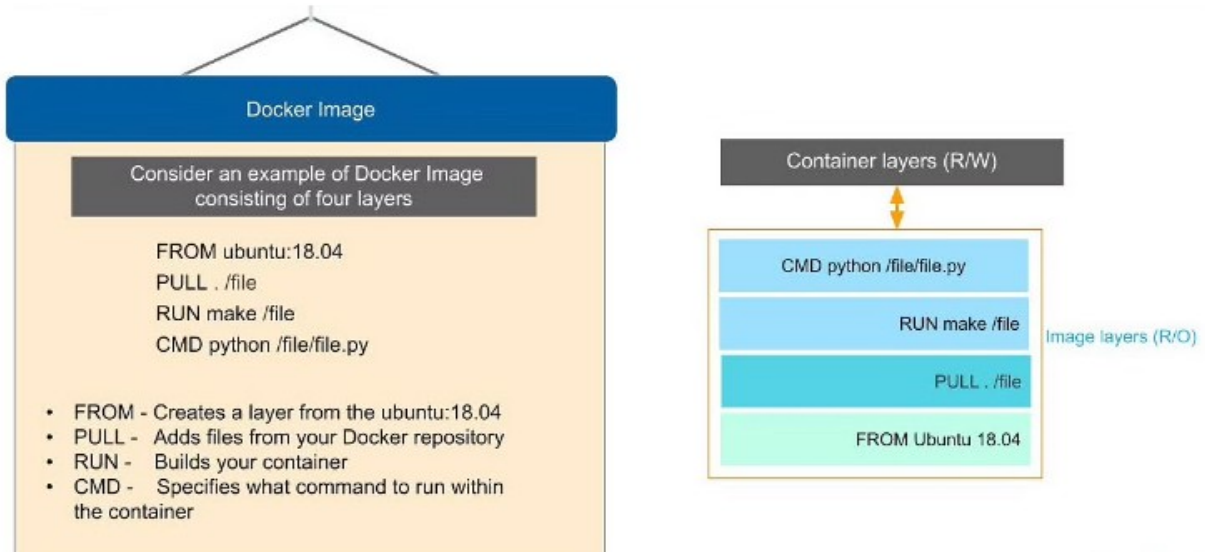


**Figura 13**: Esempio di un'immagine Docker, mostra i comandi del dockerfile all'interno di ogni layer
**Fonte**:https://bryanavery.co.uk/docker-basic-concepts-components-and-advantages/

Quando si modifica il dockerfile e si ricostruisce l'immagine, solo i layer che sono stati modificati vengono ricostruiti(o compilati, credo), in questo modo rimangono leggeri, di piccola taglia e veloci.

I layer superiori dipendono da quelli inferiori, si parte con il layer di base e poi il layer delle dependecies al di sopra di essa.

I dockerfile sono semplici file di testo con istruzioni, con una sintassi semplice:

```
# Comment
INSTRUCTION arguments
```

**Figura 14**: Formato comando all'interno di un dockerfile

**Fonte**:https://docs.docker.com/engine/reference/builder/

Il comando **FROM <image**> imposta l'immagine che verrà usata come base per il processo di creazione di una nuova immagine, se l'immagine specificata non è presente nel sistema in cui viene eseguito il processo di compilazione Docker, il motore Docker cercherà di scaricare l'immagine da un registro d'immagini pubblico o privato.

Se compiliamo il comando dell'esempio "**FROM ubuntu:18.04**" , l'immagine risultate deriva e presenta una dipendenza dall'immagine del sistema operativo di base Ubuntu.

Il comando **RUN <command>** specifica i comandi da eseguire nella nuova immagine del container, può includere installazione di software, creazione di file e directory  etc...

il comando **CMD <command>** imposta le istruzioni predefinite da eseguire quando si distribuisce un'istanza dell'immagine.

Altri comandi noti:

- COPY
- ADD
- MANTAINER

### 3.1.2.2 Container

I **container Docker** sono istanze eseguibili di un'immagine, si possono creare, arrestare, spostare ed eliminare usando  tramite client (CLI) e l'API. Sono progettati per l'esecuzione l'esecuzione di applicazioni e servizi, contenendo tutte le dependencies necessarie per la corretta esecuzione.

Docker permette di personalizzare vari aspetti del container:

- **Livello d'isolamento**: di default un container è isolato dal resto del mondo, ma si può configurare il container stesso in modo tale che si possa connettere a reti e comunicare con altri container e servizi.
- **Montaggio di storage**: Docker permette di "montare" file e directory al container(come volume, bind mount...) e possono essere condivisi con altri container.

I container Docker sono definiti dalla loro immagine e le configurazioni fornite dall'utente durante la creazione ed esecuzione, nel caso d'arresto tutte le configurazioni non salvate nello storage sono perse. Si possono creare immagini basate sullo stato di un container.

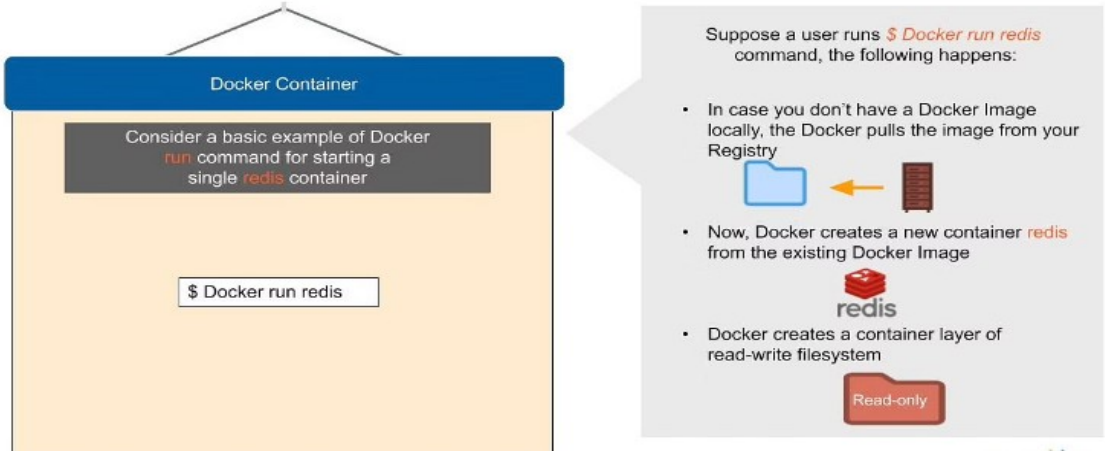


**Figura 15**: esempio comando run **Fonte**:https://bryanavery.co.uk/docker-basic-concepts-components-and-advantages/

Questo è un esempio di base del comando run per avviare un container chiamato redis.

Se l'immagine non è salvata localmente, verrà scaricata (pull) dal registry, una volta fatto, il nuovo container Redis sarà disponibile nell'ambiente utente pronto all'uso.

### 3.1.2.3 VOLUMES

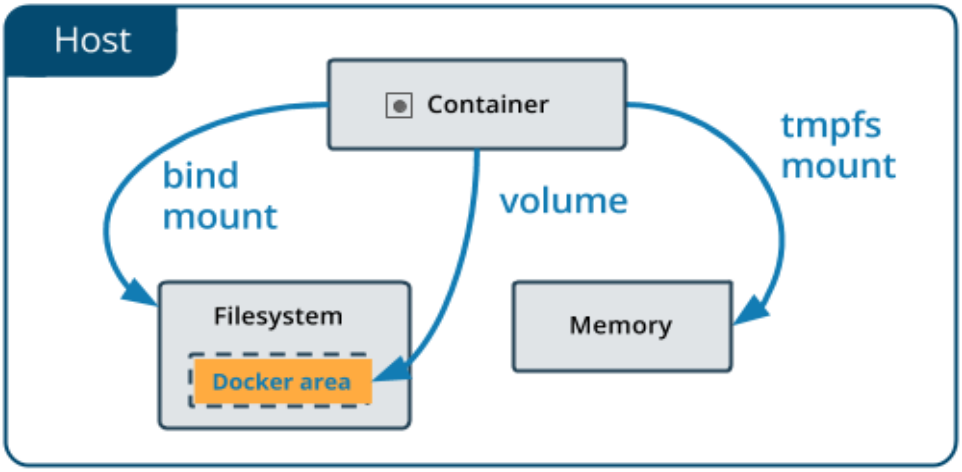


**Figura 16**: Schema che rappresenta i volume e mount "attaccati" a un container

**Fonte**:https://docs.docker.com/storage/volumes/

I container Docker necessitano di storage per i dati, Docker fornisce varie modalità:

- **bind mount**:sono file o directory della macchina host "montati" al container vengono identificati tramite il loro path assoluto nella macchina host. L'uso dei bind mount è sconsigliato dato che i processi estranei a Docker possono modificarne i contenuti.
- **volume**: è il meccanismo predefinito per i dati persistenti, a differenza dei bind mount, i volumes sono gestiti esclusivamente da Docker, cioè la directory o file nel file system dell'host fisico a cui il volume fà riferimento, non può essere modificato da processi che non appartengono a Docker. I volumes possono essere creati con il comando **volume create**, un volume può essere montato a più container. Se un volume è inutilizzato, rimane all'interno del file system host finché non vine eliminato con il comando **volume prune.** Si possono assegnare nomi ai volume, altrimenti viene assegnato un nome random univoco.
- **tmpfs mount**: alternativa disponibile su Linux, consiste nell'uso di una locazione al di fuori del layer modificabile del container, è utile per la memorizzazione di file temporanei(spesso sensibili), il tmpfs mount può essere creato alla creazione o durante l'esecuzione del container, una volta che il container viene arrestato, il tmpfs mount sparisce per sempre.

L'uso dei volume presenta una moltitudine di vantaggi rispetto ai bind oltre all'isolamento dalle funzioni principali della macchina host, i volumes permettono processi do migrazione e backup più semplici, possono essere salvati in host remoti o su provider cloud, e molti altri vantaggi.

#### 3.1.2.4 NETWORKS

Un aspetto dell'engine Docker che lo rende potente, è la possibilità di creare e gestire reti di container e servizi Docker. Le reti permettono l'interconnessione dei container Docker con altri container non necessariamente appartenenti allo stesso host e altri dispositivi.
Il sistema di networking Docker è definito in base alla scelta del **driver**:

- **Bridge**: E' il driver di rete impostato di default, si comporta come un bridge software al quale i container connessi ad esso possono comunicare tra di loro, permette anche l'isolamento dai container non connessi al bridge. Il bridge si applica può comunicare con altri bridge al di fuori della macchina host tramite routing al livello di sistema operativo.
- **Host**: Questo driver prevede l'utilizzo dello stack di rete dell'host fisico da parte del container, rimuovendo l'isolamento a livello di rete tra l'host e il container. Si può utilizzare per ottimizzare le performance nelle situazioni dove il container deve gestire un gran numero di porte, dato che non necessita di mecanismi per la traduzione d'indirizzi(come il NAT)
- **Overlay**: Questo driver crea una rete distribuita tra più daemon all'interno di diversi host docker.
- **None**: Questo driver disabilita tutto il networking.
- **Macvlan**: Prevede l'assegnamento di un mac-address ai container, in questo modo tale i container sembreranno dei dispositivi fisici nella rete. È una soluzione più adatta per la gestione di applicazioni legacy che si connettono direttamente con la rete fisica invece di essere instradati attraverso lo stack di rete dell'host docker.
- **Ipvlan**: Questo driver da l'intero controllo dell'addressing ipv4 e ipv6 agli utenti.
- **Plugin di rete**: Si possono installare plugin provenienti da terze parti in docker. Disponibili su DockerHub o vendors di terze parti.

#### 3.1.2.5 REGISTRY

Un registro Docker è una repository dedicato all' hosting e distribuzione delle immagini Docker.
Gli utenti usando il client possono connettersi al registry tramite i comandi:

- **pull**: per il download d'immagini dal registry
- **push**: per salvare nel registry un ambienti docker containerizzati che l'utente ha creato nel nodo locale
- **create**: se l'immagine non risulta presente localmente, manda prima una richiesta "pull" al registry

Il client Docker usa di default il registry pubblico **Docker Hub**, un registro pubblico disponibile a tutti, fornisce immagini di base conformi alle buone pratiche di sviluppo delle immagini.
Un modo per facilitare la ricerca delle immagini Docker nel registry, è assegnare dei name tag alle immagini.

#### 3.1.2.6 PLUGIN

Docker supporta l'impiego di plugin forniti da terze parti, promuovendo l'espansione delle funzionalità dell'engine Docker.

#### 3.1.2.7 SECRET

I secret sono dei BLOB(binary large object), cioè dati in formato binario tipicamente impiegati nei database per la memorizzazione di dati di grandi dimensioni. Il ruolo dei secret è quello di contenere dati sensibili e trasmetterli in modo sicuro ai servizi autorizzati.

Un secret può essere:

- credenziali username e password
- certificati TLS e chiavi
- chiavi SSH
- Stringhe generiche
- Dati importanti come nome  di un database o server interno
- file di configurazione

I secret sono principalmente impiegati nei cluster Swarm e Kubernetes.

# 4. Tools di Docker

Le fonti da cui il contenuto di questo capitolo è tratto, sono indicate nel capitolo 8 Bibliografia.

Docker è una suite di tools per lo sviluppo software per creare, condividere e lanciare container.

La piattaforma Docker offre tool per creare, condividere ed eseguire container.

Si dividono in 2 categorie:

• Tool integrati nella piattaforma

• Tool forniti da sviluppatori open-source

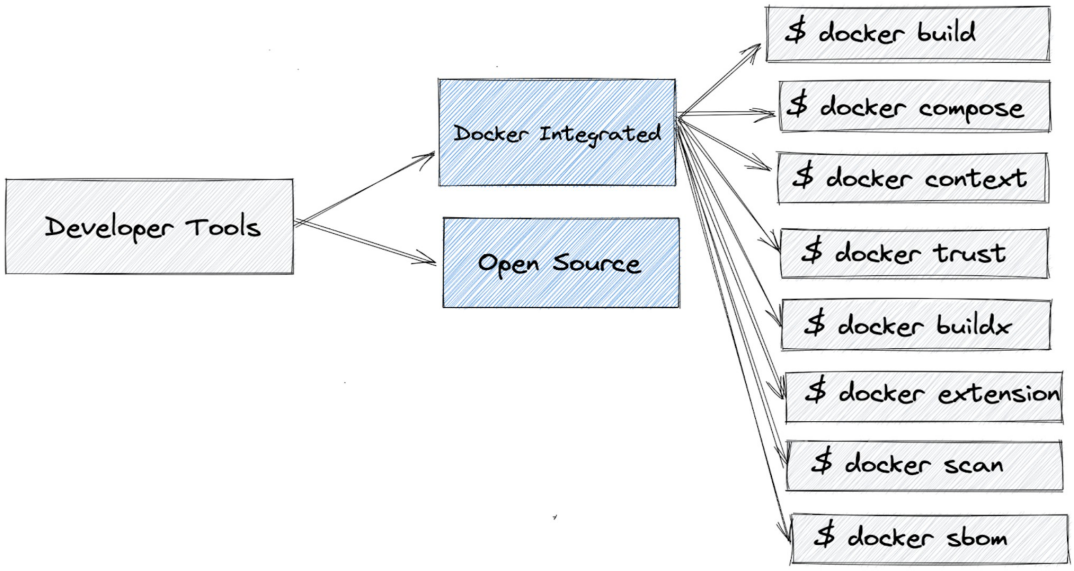


**Figura 17**: Schema riassuntivo, mostra i tool principali di Docker

**Fonte**:https://collabnix.github.io/dockertools/

## 4.1 Tool integrati

### 4.1.1 Docker Build

Uno dei tool più impiegati nell'engine Docker, permette facilmente di creare e condividere immagini Docker. Grazie a Build, si può confezionare il codice e condividerlo dove si vuole, un aspetto chiave dello sviluppo. Data l'architettura client-server su l'engine si basa, l'impiego più comune di Build è tramite linea di comando "CLI", che invia ogni comando al "server" per l'esecuzione.

Con le nuove versioni di Docker, è stato introdotto Buildx offrendo un maggior numero di funzioni per i scenari più complessi.

Lista di alcune funzioni Dockerbuild:

- Scegliere un driver per scegliere con quali configurazioni eseguire Buildx, in per adattarsi all'ambiente di lavoro
- Impiego di memoria cache condivisa per migliorare le prestazioni
- Creare build supportati da piattaforme diverse
- Integrazione di Github e supporto di  operazioni automatizzate
- Scegliere il formato dell'output di Build
- Gestione di secrets delle build per garantire l'accesso sicuro a repository e risorse al tempo di build senza fughe di dati all'interno della build finale o memoria cache.

### 4.1.2 Docker Scan

Tool utile per individuare le debolezze nelle immagini Docker locali, permette agli sviluppatori(e team di sviluppo) di controllare lo stato di sicurezza delle immagini e prendere le misure necessarie per risolvere i problemi individuati, portando di conseguenza a deployment più sicuri.

### 4.1.3 Docker SBOM

Questo tool è utile per la generazione di SBOM di un immagine. Promuove la visibilità di ciò che è all'interno di un immagine container, migliorando la sicurezza della supply chain

L'utente può vedere lo SBOM di ogni immagine Docker tramite comando con il CLI.

Un SBOM(Software Bill Of Materials) mostra ogni componente che compone il software o è stato utilizzato per costruirlo, include OS installati, può anche includere altre informazioni, come la versione e la provenienza, identifica ogni artefatto all'interno dell'immagine nel container, permettendo di determinare le vulnerabilità nella supply chain del software e rimediare tempestivamente in qualsiasi stadio dello sviluppo del software.

SBOM migliora la sicurezza di Docker in vari aspetti:

- Generare un unica fonte di verità per tutti gli artefatti dei container
- Aggregare SBOM per immagini individuali per garantire un panorama a livello applicativo
- Scoprire dependecies ed errori inaspettati, e manipolazioni di natura fraudolenta con intenzione di infiltrare la build
- Permette di eseguire scan per individuare vulnerabilità in tutti gli stadi del ciclo di vita
- Creare e imporre policy in base ai dati forniti dalle SBOM
- Adattarsi a requisiti imposti da clienti o autorità

Il ruolo di SBOM è molto importante perché permette di sapere ciò che è all'interno dell'immagine e quando e/o  dove è stato introdotto.

### 4.1.4 Docker context

Docker Context è un tool per la gestione di multipli cluster(Swarm e Kubernetes) e nodi Docker, tramite un singolo CLI Docker.

Un elemento fondamentale di questo tool è il context, che include le informazioni necessarie per la gestione del cluster o nodo.

Un singolo CLI Docker può gestire più di un context, e muoversi da uno a un altro con facilità.

In questo modo è possibile gestire diversi cluster Swarm/Kubernetes da un singolo Docker client

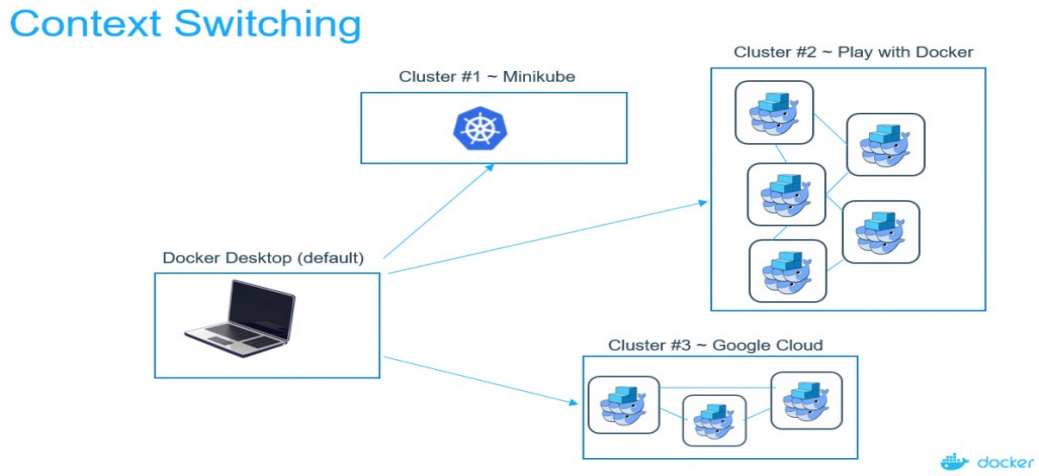


**Figura 18**: Esempio grafico del context switching

**Fonte**:https://collabnix.com/docker-19-03-0-fast-context-switching-rootless-docker-sysctl-support-for-swarm-services/

### 4.1.5 Docker Compose

Docker Compose è tool che consente, di gestire i propri container salvandone le impostazioni in un unico file di configurazione in formato YAML.

Compose consente la rapida e pratica gestione dei container, ogni volta che si necessita di aggiungere un container o aggiornare un impostazione, basta modificare il codice del file YAML.

I container definiti nel file fanno parte di una "stack", ogni container ha le impostazioni specificate nel file di configurazione come volumes, variabili d'ambiente e modalità di rete.

Esempio di file YAML **docker-compose.yaml**, in questo caso possiamo vedere come si dichiara un servizio e il livello di personalizzazione di esso. Si possono specificare le porte, modalità di rete, volumes da montare e politiche di riavvio.

```
version: '3'
services:
  home-assistant:
    container_name: home-assistant
    image: homeassistant/raspberrypi4-homeassistant:stable
    volumes:
    - /home/homeassistant/.homeassistant:/config
    environment:
    - TZ=Europe/Rome
    network_mode: host
    restart: always
  caddy:
    image: caddy:latest
    container_name: caddy
    ports:
    - "443:443"
    environment:
    - "TZ=Europe/Rome"
    volumes:
    - "/home/NOME_UTENTE/caddy/Caddyfile:/etc/caddy/Caddyfile"
    - "/home/NOME_UTENTE/caddy/data:/data"
    restart: always
```

**Funzioni di compose**

- **Esecuzione di 2 o più ambienti in un singolo host**: per mantenere gli ambienti isolati tra di loro in Compose, vengono impiegati i project name. In questo modo si possono creare copie dello stesso ambiente, come per eseguire due o più feature branch di un progetto oppure eseguire build diverse senza che interferiscano tra di loro.
- **Conservare i dati dei volume**: Con Compose è possibile salvare dati usati dai servizi, in questo modo i dati creati dai container non vengono persi, e quando un nuovo container viene eseguito, Compose cercherà i dati delle vecchie esecuzioni e fornirà al nuovo container una copia del volume.
- **Supporto alle variabili d'ambiente**: Si possono definire variabili d'ambiente per personalizzare la composizione in base all'ambiente o utenti diversi. Ciò da una certa flessibilità quando s'impostano i container con Compose. Le variabili d'ambiente possono avere ruoli diversi, come una password o indicare la versione di un software.
- **Riutilizzo di container esistenti**: Compose salva nella cache la configurazione usata per creare container. Quando un servizio viene riavviato che non ha subito cambiamenti, Compose riutilizza il container esistente.

**Casi d'uso di compose:**

- **Ambienti di sviluppo**:Data la l'importanza di poter avviare e interagire con un'applicazione in un ambiente isolato. La linea di comando compose può essere usata per creare e interagire con l'ambiente. Il file Compose fornisce un modo per documentare e configurare le dependencies del servizio(databases, queues, caches, web service API's. Etc..)
- **Ambienti di testing automatico**: Una parte importante del processo CI/CD è la suite di test automatizzato. Un test automatizzato end-to-end necessita un ambiente dove eseguire i test. Compose fornisce un metodo conveniente per creare e distruggere ambienti di testing per la propria suite di test.
- **Deployment multipli su un singolo host**: In Compose, i container sono stati pensati per l'esecuzione in un singolo host con il focus sullo sviluppo e testing.

### 4.1.6 Docker Content Trust(DCT)

Il Docker Trust fornisce agli individui e organizzazioni, il servizio di firma digitale per i dati da e per i registri Docker, in tal modo è possibile verificare l'integrità delle immagini e il pubblisher di un'immagine Docker.

Ogni immagine ha associato dei tag, che possono indicare vari aspetti come la versione della build,i tag di un'immagine Docker sono gestiti tramite un set chiavi, generato quando un operazione DCT(spesso l'operazione di push) è invocata per la prima volta .

Il set di chiavi è composto da:

- **chiave offline:** chiave da tenere al sicuro.
- **chiave repository o tagging:** chiave utilizzata per firmare i contenuti della repository.
- **chiave timestamp:** chiave gestita server-side come timestamp per garantire la versione di un'immagine sia la più recente.

Tramite queste chiavi, Docker Content trust può controllare la firma crittografica quando un utente manda una richiesta di pull. Se le chiavi coincidono, il contenuto è considerato autentico, in caso contrario l'utente verrà avvertito. I pubblisher possono decidere se firmare un tag specifico o meno, è possibile imbattersi a immagini Docker con non tutti i tag firmati.

Grazie a DCT si ha un metodo aggiuntivo per individuare immagini false, non aggiornate(spesso danneggiano le prestazioni) o contenenti malware. Tuttavia non è una misura di sicurezza infallibile, ogni utente può caricare immagini firmate nel registry pubblico, quindi la responsabilità nel testare le immagini cade nelle mani degli utenti.

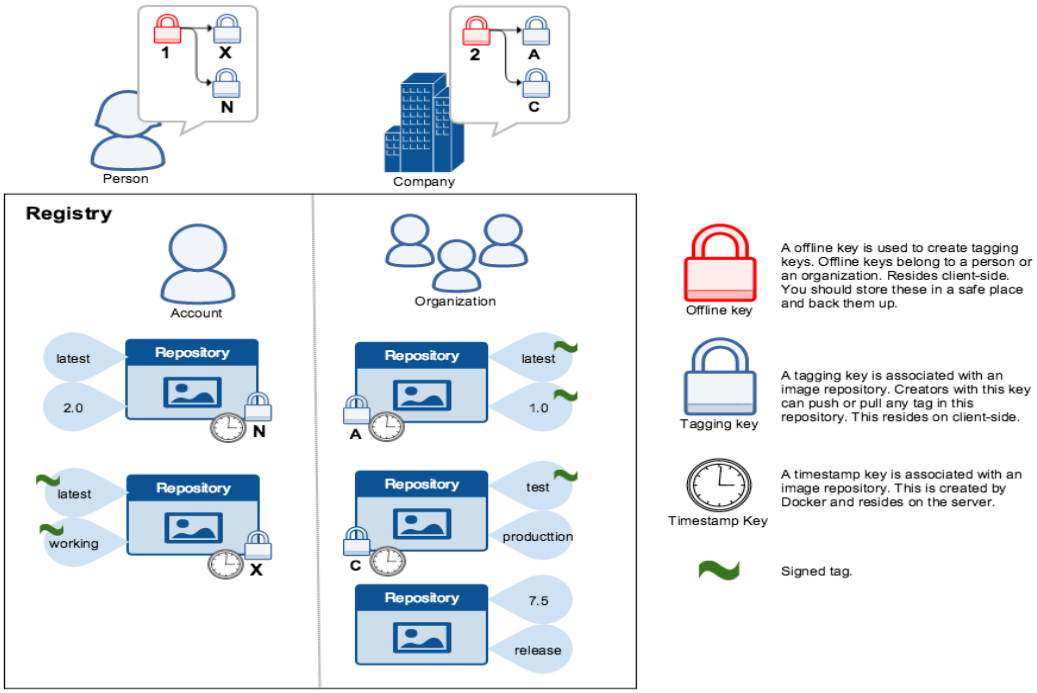


**Figura 19**: Esempio grafico del sistema Content Trust

**Fonte**:https://docs.docker.com/engine/security/trust/

### 4.1.7 Docker extensions

Permette di utilizzare tool di terze parti nel Docker Desktop per estendere la sue funzionalità, i tool sono sviluppati dai membri della comunità Docker tramite il kit di sviluppo(SDK) e pubblicati nel marketplace dopo un controllo per l'idoneità fatta dall'organizzazione.

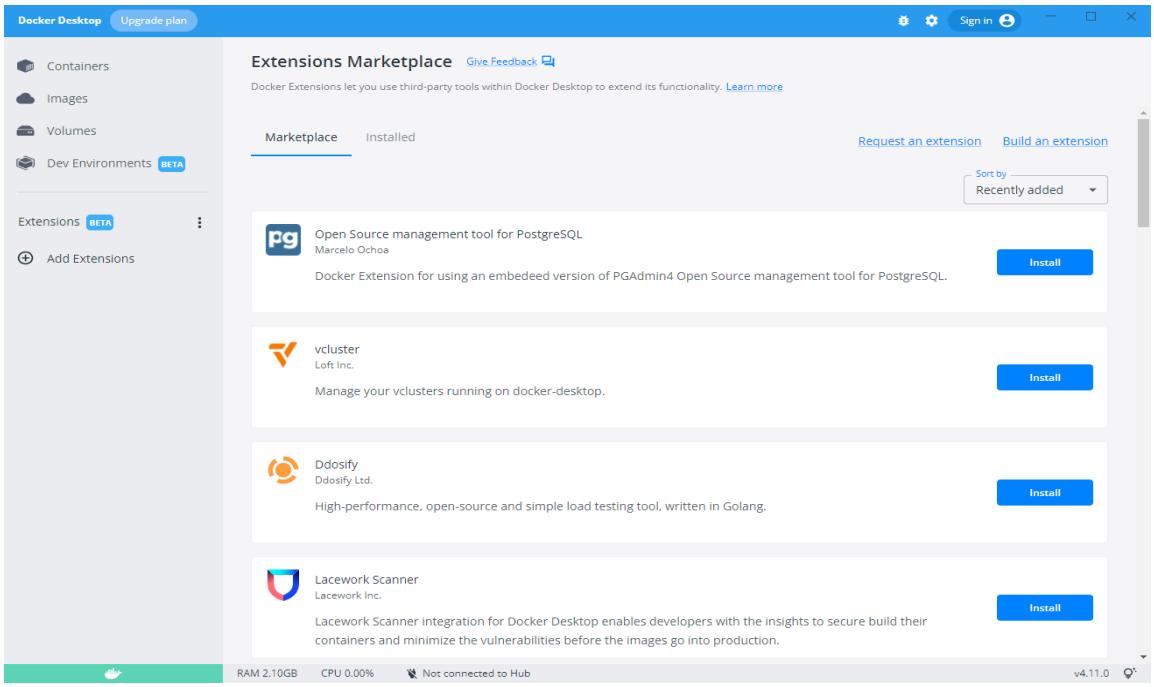


**Figura 20**: Schermata dell'extensions marketplace

**Fonte**:https://docs.docker.com/desktop/extensions/

### 4.2 Tool forniti da sviluppatori open-source

La piattaforma Docker supporta i tool open-source offerte dalla comunità, esistono una grande varietà di progetti che coprono diverse esigenze e casi d'uso. Come la gestione di volumes, tool d'orchestrazione (come Swarm e Kubernetes che tratteremo nel prossimo capitolo), Build Test di applicazioni Android e molti altri ancora.

# 5. I limiti di Docker

Le fonti da cui il contenuto di questo capitolo è tratto, sono indicate nel capitolo 8 Bibliografia.

## 5.1 Orchestrazione

Con Docker si può gestire in modo efficiente un numero limitato di container. Quando invece il numero di container e applicazioni containerizzate(che a loro volta sono scomposte in numerose componenti) diventa molto grande, la gestione e orchestrazione diventa molto complessa, e si necessitano tool che permettono di eseguire tali operazioni in modo automatizzato.

Docker supporta Kubernetes e Docker swarm

### 5.1.1 Docker Swarm

Le ultime versioni di Docker supportano la modalità "swarm", che comprende funzioni per gestire cluster di host Docker, denominati swarm.

Questa modalità permette di creare swarm, gestirli e distribuire servizi in essa tramite linea di comando(CLI).

**Architettura**

La modalità swarm organizza il cluster di host(swarm) con la modalità attiva, assegnando un ruolo ai singoli nodi(generalmente macchine fisiche nel cluster):

- **manager**: hanno il compito di gestire lo swarm e la distribuzione dei servizi
- **worker**: eseguono i servizi assegnati dal manager

Un nodo può avere entrambi ruoli assegnati.

Tipicamente gli swarm sono nodi distribuiti in diverse macchine fisiche e cloud.

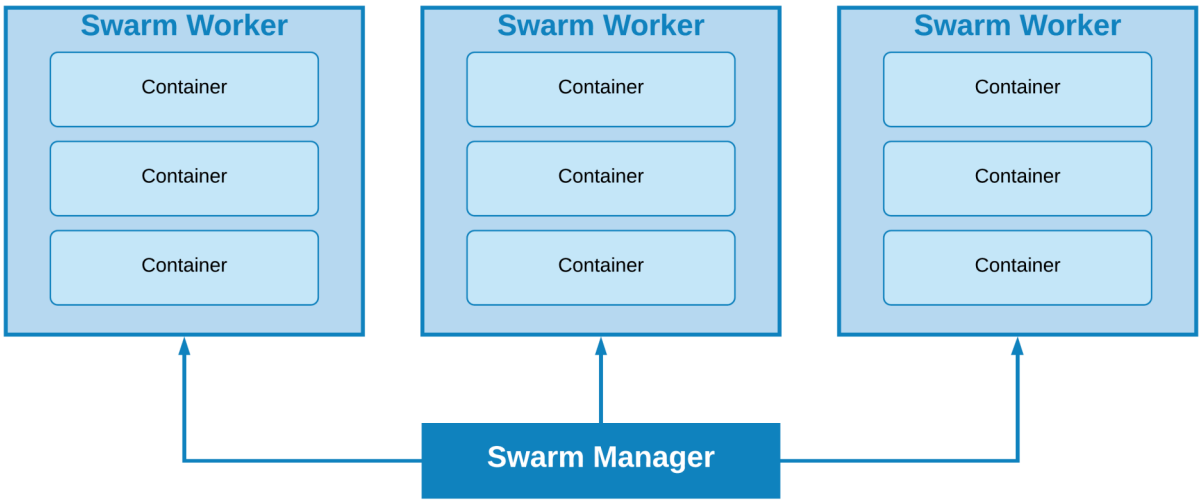


**Figura 21**: Architettura del cluster Swarm

**Fonte**:https://www.linkedin.com/pulse/why-docker-swarm-gitonga-bretton?trk=articles_directory

**Task e Servizi**

I **task** sono l'unità atomica di lavoro, contiene l'immagine del container e i comandi ad eseguire in esso.

Un **servizio** è la definizione del task da eseguire nei nodi, e inoltre specifica un insieme di requisiti detto **stato desiderato**:

- numero di repliche da eseguire nello swarm
- risorse necessarie(storage, CPU...)
- regole di aggiornamento
- la rete(di tipo overlay) a cui il servizio può connettersi per comunicare nello swarm
- porta di rete dove è possibile accedere al servizio dall'esterno

## 5.1.2 Kubernetes

È una piattaforma d'orchestrazione dei container open-source, progettato per la gestione, deployment e scaling in modo automatizzato del software. Vanta di un grande ecosistema in rapida crescita, con un ampia gamma di servizi, supporto e strumenti disponibili.

**Architettura**

Kubernetes presenta un architettura a grandi linee simile a quella di Docker Swarm, ma più complessa, il cluster è composto da nodi:

- **worker**
- **master(detto anche Control panel)**

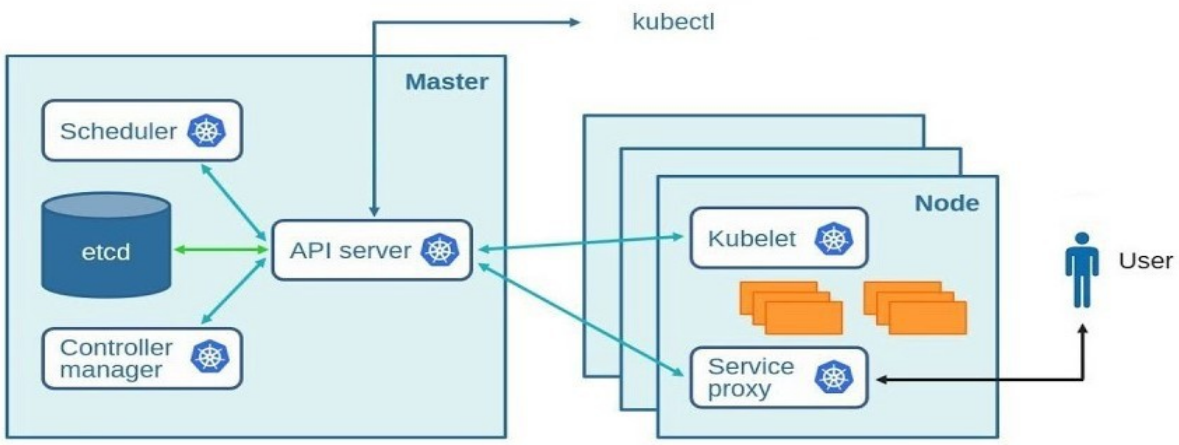


**Figura 22**: Architettura cluster Kubernetes

**Fonte**:https://medium.com/@shubhamdeshmukh8531/basics-of-kubernetes-architecture-4ed4110caf70

**Worker**

Ogni nodo worker ha:

- **pod:**L'unità minima di elaborazione, incapsulano un container astraendo il container runtime come Docker. Contengono tutto ciò che è necessario per l'esatta esecuzione del container, come le dependencies o persino volume per lo storage di dati. I pod possono comunicare tra di loro e avere il proprio indirizzo ip per la comunicazione della rete virtuale del cluster.
- **kubelet**: Il "node agent" principale, si assicura che l'esecuzione del container all'interno di un pod rispetti le specifiche fornite dal Master.
- **Kube-proxy**: Processo in esecuzione su ogni nodo del cluster, gestisce la comunicazione nella rete del cluster e mantiene le regole di networking. Può gestire la comunicazione per i pod e l'host.
- **container runtime:**software responsabile per l'esecuzione dei container, come Docker.

**Master(Control panel)**

Eseguono i processi necessari per la gestione del cluster, gestiscono i nodi worker e tutto quello che accade nel cluster per garantire le prestazioni, disponibilità e resilienza dei nodi worker. Un nodo master ha le seguenti componenti:

- **apiserver(kube-api server):**è la componente che espone la kubernetes api, agisce come il front end del Master Node, si possono eseguire multiple istanze dell'apiserver e bilanciare il traffico tra di esse(soluzione consigliata per avere un cluster più resistente ai down time).
- **controller manager:** componente del che gestisce un insieme di software chiamati controllers, ogni controller ha un ruolo differente.
- **scheduler(kube-scheduler):** Componente con l'incarico di assegnare i pod ai nodi, la scelta è basata su vari criteri.
- **etcd:**Database "coppia chiave-valore", usato da kubernetes per salvare le informazioni del cluster. È considerata come la fonte di dati principale nel cluster(source of truth).
- **kubectl:** è un tool CLI che permette di eseguire i comandi nel cluster, mandando le istruzioni all'api-server.

**Deployment e ReplicaSet**

In Kubernetes il termine "Deployment" si riferisce a un modello dichiarativo, principalmente impiegato per la creazione di un pod. La creazione dei Deployment avviene in 2 modi:

- Tramite comandi specifici con il tool **kubectl** (modello imperativo)
- Impiegando file YAML o JSON (modello dichiarativo)

La creazione di un deployment tramite kubectl ha il vantaggio di essere immediato, ma non permette di aggiornare facilmente le configurazioni, mentre nel caso del metodo dichiarativo è possibile tramite la semplice modifica dei file di configurazione.

Il Deployment permette di definire il ciclo di vita dell'applicazione da distribuire nel cluster, come le immagini usare, il numero di pod necessari e le politiche d'aggiornamento.

Con i Deployement è possibile definire un tipo controller chiamato **replicaSet**,il loro compito è garantire che in un determinato momento, all'interno del cluster Kubernetes, sia in esecuzione un numero specificato di Pod.

## 5.1.3 Docker swarm vs Kubernetes

**Scheduling e Deployment delle applicazioni**

In **Docker Swarm** il deployment è semplice e immediato, il processo ha inizio con la definizione di un task che specifica lo stato desiderato, tramite linea di comando o file YAML. La definizione del task deve essere mandata al manager del cluster, che si occuperà del processo di assegnamento. Il manager assegnerà i task in base allo stato desiderato, se nessun nodo disponibile è in grado di soddisfare il servizio, esso rimarrà in stato di **pending** finché un nodo adatto sarà disponibile. Una volta assegnati i task ai nodi, i worker eseguiranno in modo indipendente e comunicheranno lo stato corrente del task al manager tramite invio di report.

**Kubernetes** tramite i Deployment fornisce una scelta maggiore di opzioni, permette la definizione dei vari aspetti del ciclo di vita dell'applicazione come il numero di pod, immagini da usare e le politiche d'aggiornamento.

Il processo ha inizio con l'invio del deployment al kube-api server da parte dell'utente, una volta verificata l'autenticità della richiesta, lo scheduler per la definizione dello stato desiderato e selezione dei nodi adatti. Una volta che determinati i nodi del cluster, il carico di lavoro verrà distribuito ai nodi selezionati, che eseguiranno i pod.

Nel frattempo all'interno dell'etcd, verrano scritte dati importanti come il desired state, lo stato dei

pod e richieste comunicate tra gli elementi del cluster.

**Availability e self-healing**

Per loro natura, Docker e Kubernetes forniscono una grande disponibilità dei microservizi, attraverso meccaniche di replicazione, e di conseguenza una grande resistenza ai fallimenti.

In **Kubernetes** vengono impiegati degli "health checks" periodici mandati ai container, in caso di mancata risposta, vengono prese le misure necessarie come il riavvio o la terminazione dei container non funzionanti. Inoltre il traffico destinato al container difettoso verrà deviato finché non sarà pronto a rispondere correttamente.

Grazie all'impiego dei ReplicaSet, è possibile garantire il livello desiderato di disponibilità dei servizi, tramite i parametri definiti durante il deployement.

**Docker Swarm** ha un meccanismo simile basato sul "Desired state reconciliation", lo swarm manager monitora costantemente lo stato del cluster, e cerca di mantenere lo stato attuale del cluster uguale allo stato desiderato. Se per esempio un container si è arrestato in modo anomalo, il manager assegnerà una replica del container un altro worker disponibile.

Per quanto riguarda la distribuzione di più repliche dello stesso container, Swarm prevede la distribuzione dei container in 2 modi:

- distribuzione **replicata:** il manager distribuisce un numero specifico di task tra i nodi che eseguiranno lo stesso task, il numero di repliche è definito dall'utente nello stato desiderato del servizio.
- distribuzione **globale**: il task del servizio viene eseguito da ogni nodo disponibile nello swarm, ogni volta che un nodo viene aggiunto allo swarm, il manager assegnerà ad esso il servizio.

**Rolling update**

Il "rolling update" è una tecnica di distribuzione del software incrementale, pensato per ridurre al minimo il down-time del cluster. L'aggiornamento del servizio nel cluster viene eseguito con un deploy del container avente il servizio aggiornato in modo incrementale e non simultaneo, il load balancer viene configurato in modo tale da deviare il carico di lavoro ai container disponibili. Nel caso dove l'aggiornamento non và a buon fine o si è verificato un problema, su Swarm il servizio viene riportato alla versione precedente(rollback) automaticamente, mentre in Kubernetes deve essere richiesto in modo esplicito perché sia il pod attuale e quello aggiornato vengono terminati.

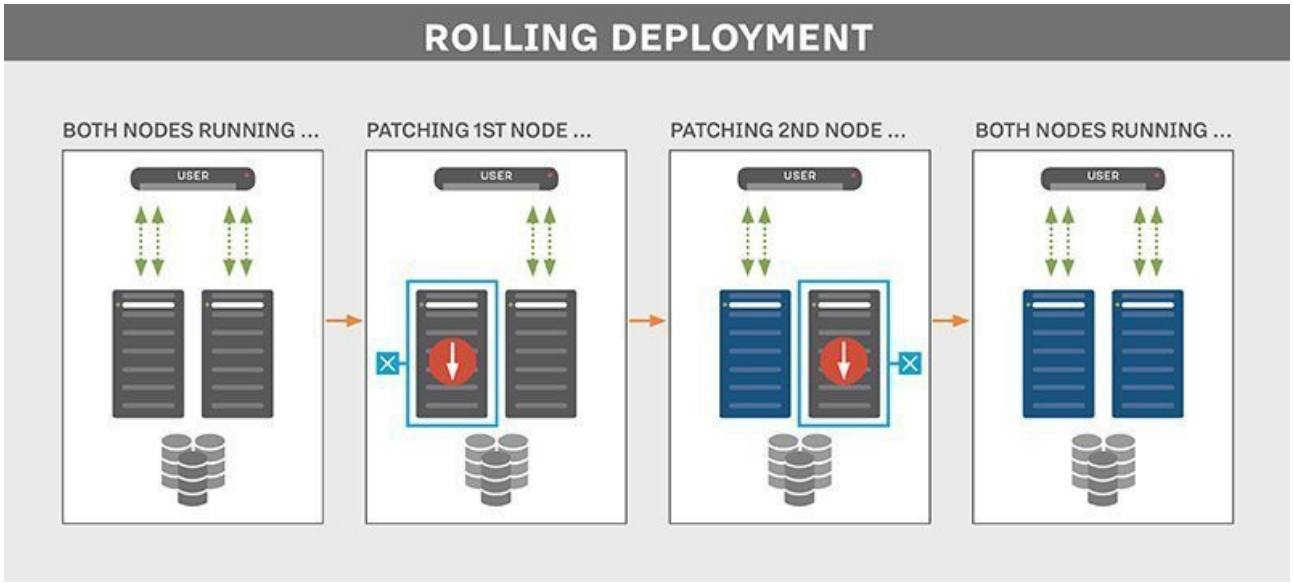


**Figura 23**: Rappresentazione grafica del meccanismo d'aggiornamento incrementale

**Fonte**:https://www.techtarget.com/searchitoperations/definition/rolling-deployment

**Scaling**

Swarm ha velocità di deployment più elevate ma non fornisce scaling automatico, mentre Kubernetes fornisce meccanismi automatizzati di scaling in base alla domanda, lo scaling avviene tramite l'assegnamento di nuovi pod ai nodi disponibili.

**Gestione di informazioni sensibili e della configurazione**

Sia Swarm che Kubernetes consente di memorizzare e gestire informazioni sensibili, come le password, i token OAuth e le chiavi SSH, sotto forma di **secret**. Si possono distribuire e aggiornare le informazioni sensibili e le configurazioni delle applicazioni, senza dover ricostruire le immagini dei container e senza svelare le informazioni sensibili nella configurazione del sistema.

**Load Balancing**

Il load balancing è un meccanismo per gestire il flusso d'informazione nel cluster tra i server e i nodi, è una soluzione che può presentarsi sia in forma software che hardware, e ottimizza il l'uso delle risorse di rete prevenendo l'overload della rete.

Docker Swarm supporta servizi di Load Balancing automatizzato, mentre Kubernetes necessita di tool esterni.

**Monitoring**

Quando si deve gestire un cluster, il monitoraggio può essere vitale. Il monitoring consiste nel tracciare l'attività e le metriche del cluster, per capirne il comportamento e avvertire tempestivamente gli sviluppatori nel caso di comportamenti anomali.

Docker Swarm non offre servizi di monitoring integrati, solo tramite applicazioni di terze parti, mentre Kubernetes include già meccaniche di monitoraggio oltre al supporto per i tool di terze parti.

### 5.1.4 Quale tool scegliere?

La scelta di quale tool d'orchestrazione dipende dalle necessità dell'organizzazione, esperienza del team, complessità del progetto e la frequenza dei deployment delle applicazioni. Kubernetes senza ombra di dubbio è lo strumento più potente, godendo del supporto dei provider cloud principali(AWS, Azure e Google), con una vasta gamma di servizi integrati come l'auto-scaling e il self-healing. Ma è uno strumento complesso, pesante e difficile da imparare. Docker Swarm invece è uno strumento più leggero, già incluso nell'engine e "beginner friendly", consigliato per progetti semplici e frequenza bassa di deployment.

### 5.1.5 Nota su K3s

Un'alternativa valida è K3s, è un versione più leggera di Kubernetes che elimina la complessità e fornisce un'esperienza più accessibile. La leggerezza di K3s deriva dal fatto che è un binary, includendo l'api di K8s ma elimina i pacchetti non neccessari.

Si può estendere le funzionalità tramite add-on di terze parti, e si presenta come un buon puntod di partenza per transizionare gli utenti di Swarm a Kubernetes.

## 5.2 Sicurezza

Docker non necessariamente è una piattaforma insicura, ma bisogna comunque tenere conto delle possibili minacce e impiegare buone pratiche per creare un cluster sicuro.

### 5.2.1 Possibili pericoli

• **Attacchi DOS(Denial of Service)**:Dato che i container condividono le risorse del kernel, un container può monopolizzare le risorse dell'host, questo apre il cluster a possibili attacchi contro servizi, mirati a rendere inutilizzabile un gran numero di risorse, rendendo il sistema inaccessibile.

• **Secrets compromessi**: Quando un container accede nel database, solitamente avrà bisogno di un secret, che può essere una chiave API oppure una coppia Username/Password, se un attacker ha accesso al secret avrà accesso anche al servizio

• **Comunicazione non sicura e senza restrizioni**: La comunicazione centralizzata alla macchina host di default può esporre i dati ai container sbagliati.

• **Impiego d'immagini fraudolente e/o vulnerabili**

• **Vulnerabilità nel Kernel dell'host**:dato che il kernel della macchina host è esposto ai container, possibili vulnerabilità nel kernel vanno mitigate per evitare lo scenario dove uno o più container possano causare un kernel panic, che può mandare l'intero cluster in down.

• **Container compromessi**:rari ma non impossibili, potenzialmente dannoso per tutto il cluster se il container compromesso ha permessi di root.

### 5.2.2 Buone pratiche di sicurezza per assicurare la massima sicurezza

• **Evitare i permessi di root**:Ci sono poche giustificazioni per eseguire container con come root nell'ambiente di lavoro. Su Docker i container eseguono come root di default, vanno prima configurati per avere i permessi, cosa da evitare, conviene configurare i permessi in base alle necessità del container.

• **Usare solo registri sicuri**: Come sappiamo i registri ci permettono di scaricare facilmente immagini da una repository. Ciò pero comporta un rischio, e per questo conviene scaricare immagini da registri fidati.

• **Evitare build non testate e/o innaffidabili**: È consigliato utilizzare immagini certificati da Docker per evitare l'introduzione di vulnerabilità e codice fraudolento.

• **Permettere di operare nei registry solo a utenti autorizzati**: È consigliato un controllo degli accessi basati su ruoli, dove ogni utente ha permessi specifici, che sia scaricare immagini o accedere a risorse. È una pratica potenzialmente laboriosa ma previene possibili breccie nel registro.

• **Limitare l'uso delle risorse**: Con Docker è possibile regolare l'uso di CPU e memoria di ogni singolo container.

• **Fare scan delle immagini**: Un'altra buona pratica è scannerizzare periodicamente le proprie immagini per mantenerli aggiornati e individuare possibili vulnerabilità.

• **Monitoraggio della rete e API**: I container usano reti e API per comunicare tra di loro, quindi è buona pratica impostare policy di sicurezza e monitorare periodicamente.

• **Monitoraggio dei container**:È una pratica essenziale per la sicurezza, tool specializzati potrebbero essere richiesti.

• **Mantenere il sistema operativo della macchina host aggiornato**: Bisogna assicurarsi che il sistema sia aggiornato, soprattutto per quanto riguarda la sicurezza. È consigliato l'impiego di sistemi operativi minimi.

• **Impiegare VM**: questa pratica è opzionale ideale per scenari specifici, ma protegge il kernel da attacchi, imponendo 2 strati in più da superare, il kernel della macchina virtuale e l'hypervisor.

• **Impostare reti in base alle esigenze e usare protocolli sicuri**: Creare una rete con accesso ristretto solo ai container essenziali, permette una maggire sicurezza e meno angoli dove possono avere luogo attacchi. Inoltre è buona pratica criptare le comunicazioni tra container per proteggere il traffico.

• **Salvare in modo sicuro i dati sensibili(secret)**

# 6. Casi d'uso di Docker

Le fonti da cui il contenuto di questo capitolo è tratto, sono indicate nel capitolo 8 Bibliografia.

Uno dei tanti motivi che hanno portato Docker ad essere una delle tecnologie di alto successo è la moltitudine dei casi d'uso, qualità riconosciuta dai distributori software più grandi al mondo come Microsoft e Google, ciò ha comportato un affluenza di fondi e investimenti.

## 6.1 Microservizi

Uno dei principali casi d'uso di Docker è la suddivisione di applicazioni monolitiche in microservizi indipendenti. Ogni microservizio è in esecuzione nel proprio container in modo indipendente con un database personale e comunica con gli altri microservizi tramite API.

L'impiego dei microservizi facilita la modifica senza dover riscrivere, testare e distribuire l'applicazione nella sua interezza, in caso di bug ne risentirà solo il microservizio coinvolto.

L'impiego dei container è adatto per i microservizi grazie al basso consumo delle risorse e il breve tempo di avvio.

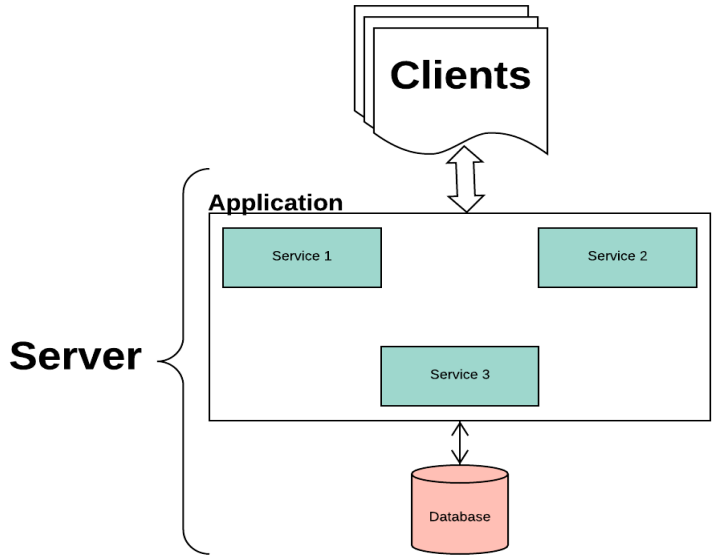


**Figura 24**: Architettura sistema che eroga microservizi

**Fonte**:https://timber.io/blog/docker-and-the-rise-of-microservices/

## 6.2 CI/CD

Il CI/CD è un approccio per lo sviluppo software, focalizzato sull'automazione delle fasi di sviluppo, questo metodo è fatto apposta per risolvere il problema legato all'integrazione di nuovo codice. Per CI è l'acronimo di "Continous Integration" cioè la metodologia d'intregrazione, continua mentre CD può fare riferimento a "Continuous Distribution" o "Continuous Deployment"(spesso usati in modo intercambiabile).

Le metodologie s'integrano in unico flusso in questo modo:

- **CI**: le modifiche al software vengono compilate, testate e unite in un repository condiviso
- **Continuous Distribution**: le modifiche apportate da uno sviluppatore all'applicazione già testate e caricate nella repository, vengono distribuite nell'ambiente di produzione dei team operativi. In questo modo l'intervento manuale per distribuire il nuovo codice è ridotto al minimo.
- **Continuous Deployment**:si riferisce al rilascio automatico delle modifiche apportate dallo sviluppatore al repository, diventando fruibili ai clienti. In tal modo si evita di sovraccaricare i team operativi con procedure manuali.

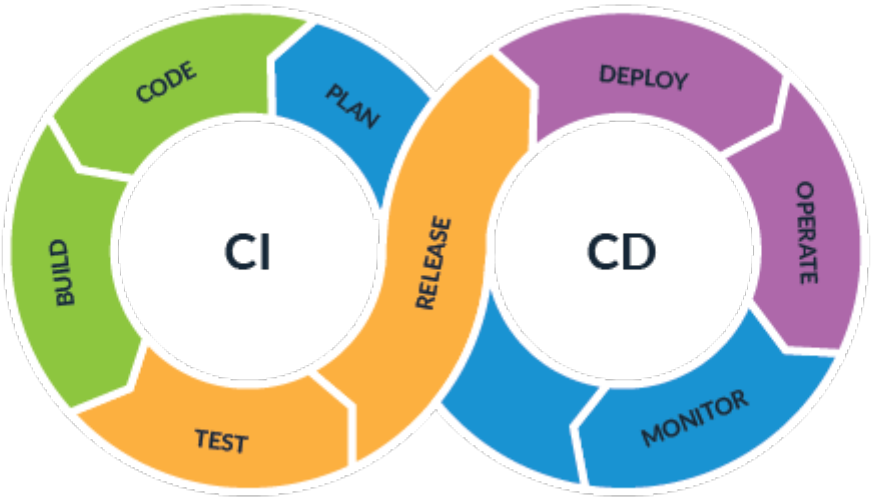


**Figura 25**: Rappresentazione grafica del ciclo CI/CD

**Fonte**: https://medium.com/@maheshchouhan1995/continuous-integration-ci-continuous-delivery-cd-fundamentals-3de82931c21e

Con Docker grazie alla leggerezza dei container che possono essere avviati in una manciata di secondi, permettendo di fare il deploy veloce delle modifiche o di nuovo codice.

Docker inoltre può evitare il dover ricaricare le dependecies della pipeline CI/CD, grazie alla cache sia in locale che in remoto con i comandi specifici, evitando rallentamenti del deployment.

Un metodo per integrare Docker nel "Continuous Integration" è il seguente:

1. I developer fanno il commit del codice verso la repository
2. Un trigger innesca il processo di build nel CI server, tipicamente viene fatta la compilazione, test e la build dell'applicazione.
3. CI server fa il deploy dell'applicazione in un application server

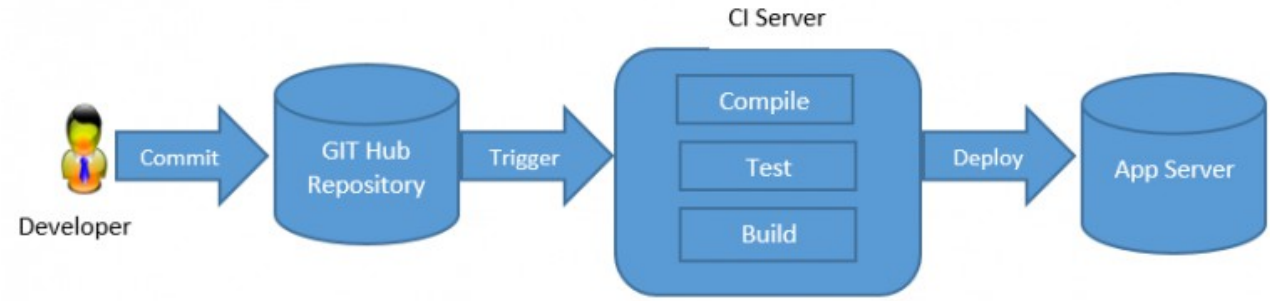


**Figura 26**: Esempio di Deployment CI senza l'utilizzo di container

**Fonte**:https://www.cigniti.com/blog/need-use-dockers-ci-cd/

Mentre con Docker il CI server crea un'immagine Docker con all'interno l'applicazione e salvata nell'hub di Docker tramite push, e i vari host che necessitano della nuova build, scaricano l'immagine con una richiesta pull ed eseguono il container.

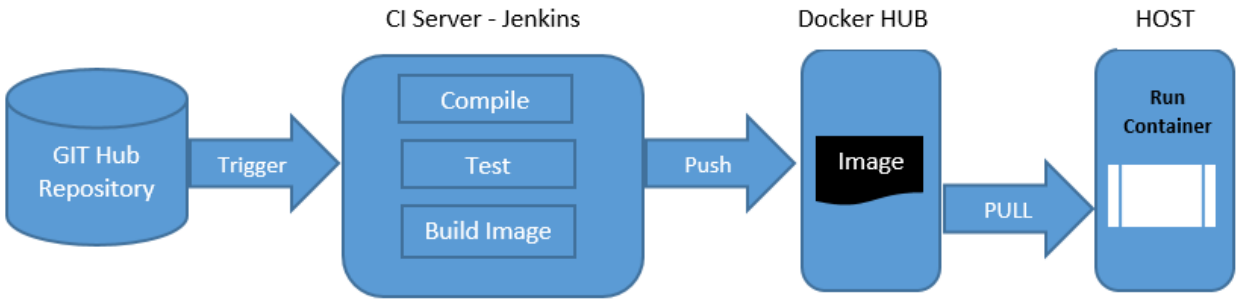


**Figura 27**: Deployment CI impiegando Docker

**Fonte**:https://www.cigniti.com/blog/need-use-dockers-ci-cd/

Altri vantaggi che Docker porta al CI/CD:

- la possibilità di fare multipli test in parallelo.
- rimuove il fattore dell'inconsistenza tra i vari ambienti di sviluppo.
- ogni macchina con Docker può usare le immagini Docker.
- separazione dei ruoli migliorata, gli sviluppatori possono preoccuparsi dello sviluppo e gli amministratori possono occuparsi del deployment.
- La leggerezza dei container comportano test rapidi e deploy del codice velocizzato.
- Version control migliorato

### 6.3 Applicazioni Cloud

Grazie alla portabilità, si possono spostare facilmente container Docker da un server o ambiente cloud ad un altro con piccoli aggiustamenti nella configurazione riducendo la complessità del processo di deployment.

Grazie al fatto che i container Docker sono eseguiti allo stesso modo sia in cloud che locale, il dilemma di dover trovare un compromesso tra costi e set di funzioni per quando si vuole spostare il carico di lavoro attraverso diversi ambienti è eliminato.

Un altro vantaggio importante è la possibilità di costruire ambienti ibridi e multi-cloud comprendendo 2 o più cloud privati o pubblici provenienti distributori diversi. Quindi si può scegliere un servizio cloud dove distribuire il carico di lavoro basandosi sui protocolli di sicurezza e i service-level agreement(SLA).

Al giorno d'oggi i principali provider cloud come AWS,GCP e Azure supportano funzioni Docker.

### 6.4 Debugging

Docker facilita il processo di debug con funzioni, utili come il poter trovare la differenza tra due versioni di container diverse o impostare una specifica versione di un'immagine come "checkpoint", funzioni molto utili per trovare la causa dei bug.

### 6.5 Disaster recovery

I container Docker possono essere facilmente creati e distrutti in modo istantaneo, quindi quando un container smette di funzionare può essere rimpiazzato da un altra istanza basata sulla stessa immagine riducendo il downtime in modo significativo.

### 6.6 Maggiore produttività

Grazie alle immagini Docker l'impostazione e gestione dell'ambiente di sviluppo per la produzione facile, veloce e con performance consistenti.

I vantaggi principali di Docker nella produzione sono:

- Servizio di cloud-management integrato per i registry, togliendo la necessità di dover gestire il proprio registry che può risultare costoso nel corso del tempo.
- Supporto per l'integrazione di repository Bitbucket e Github.
- Supporta build automatizzate, test automatizzati e e webhooks.
- La possibilità di poter configurare accessi role-based per permettere l'accesso sicuro alle immagini Docker.
- Integrazione del tool Slack per la comunicazione istantanea tra membri del team di sviluppo, promuovendo la perfetta collaborazione e coordinazione durante il ciclo di vita del prodotto.
- Supporto di una grande varietà di tool, framework e tecnologie usate nell'ambiente di sviluppo.

### 6.7 Standardizzazione multi ambiente

Uno dei problemi principali durante la produzione è il passaggio del codice da un ambiente di sviluppo a un altro, piccole differenze come la versione di una libreria o persino un editor di codice differente può portare un grande impatto alla gestione della pipeline del codice.

Con i container Docker è possibile imporre lo stesso ambiente di sviluppo nelle macchine, e grazie all'immutabilità delle immagini Docker il funzionamento del codice rimane consistente in ogni macchina senza conflitti.

### 6.8 Applicazioni multi-tenant

Il modello di deployment multi-tenant prevede una sola istanza di un'applicazione in esecuzione nel server, che serve più utenti contemporaneamente e i dati di ogni utenti sono isolati tra di loro.

L'utente(in questo caso sono chiamati "tenant") vede il servizio erogato ad esso come esclusivo e personalizzabile.

La gestione del codice che sta alla base delle applicazioni multi-tenant può essere complicato e difficile, la modifica può risultare lunga e costosa.

Docker permette di eseguire istanze dell'applicazione per ogni tenant in modo facile e conveniente.

### 6.9 Configurazione Semplificata

Uno dei vantaggi delle macchine virtuali era quello di poter eseguire qualsiasi piattaforma con la propria configurazione sopra la propria infrastruttura, Docker è capace di fare la stessa cosa in modo più leggero ed efficiente, permettendo di mettere l'ambiente e la configurazione in codice e distribuirlo, ed è utilizzazbile in vari ambienti di sviluppo. Ciò permette di l'ambiente dell'applicazione dalle restrizioni della instrattura.

### 6.10 Deployment rapido

Grazie a Docker è possibile creare un container in secondi, non necessitando di hardware nuovo o di avviare un intero sistema operativo, ciò permette di avere il  deployment più veloce e leggero.

La natura dei container rende possibile la creazione e distruzione delle risorse senza doversi preoccupare dei costi, e grazie all'immutabilità delle immagini l'uguale funzionamento dei container è garantito.

### 6.11 Infrastrutture di larga scala

Con l'avvento dei microservizi, architetture cloud e applicazioni distribuite, le organizzazioni hanno cominciato a sfruttare le funzionalità di Docker eseguendo multiple istanze dello steso container e aumentare la scala dell'infrastruttura in orizzontale.

Con l'aumentare del numero dei container diventa necessario l'impiego di tool d'orchestrazione come Kubernetes e Swarm, che includono funzioni per aumentare la scala dell'infrastruttura in modo automatizzato, ottimizzare i costi eliminando i container non necessari e monitorare il ciclo di vita dei container.

### 6.12 Isolamento delle applicazioni

L'esecuzione di più applicazioni in una sola macchina è motivata da varie ragioni, come la riduzione dei costi dei server o la separazione di applicazioni monolitiche in servizi di dimensioni più piccole.

La separazione permette anche di evitare conflitti di dependecies tra applicazioni diverse, come ad esempio due versioni diverse delle stessa libreria(dependency hell).

Ogni container è impacchettato con tutte le dependecies, quindi non c'è bisogno di preoccuparsi di conflitti tra dependecies. Ciò minimizza ogni downtime provocato da cambi d'infrastruttura.

# 7. Conclusione

Alla luce di ciò che è stato detto, Docker, si dimostra di essere un ottimo strumento per l'impostazione di sistemi basati sul deployment containerizzato. Grazie alla sua semplicità e facilità d'accesso, è un ottimo punto di partenza per sviluppatori ancora alle prime armi nell'ambito dei container. Docker è in grado di affrontare le sfide che risiedono nella gestione delle infrastutture di grandi dimensioni e il mantenimento della sicurezza, grazie al supporto dei tool che estendono le capacità della piattaforma permettendo di creare soluzioni su  misura. Un'altra ragione del successo di Docker è senza ombra di dubbio il supporto da parte della comunità, formata nel corso degli anni e tuttora in continua crescita. Docker è uno strumento potente, versatile e leggero, qualità che ha promosso il suo impiego in molti ambiti nel settore dei servizi distribuiti.

# 8.Bibliografia

Questa sezione è dedicata per fornire una lista delle fonti utilizzate durante la compilazione dell'elaborato.

### [2. Tipologie di Deployment]

I contenuti di questo capitolo sono tratti da:


- https://www.oracle.com/it/cloud/cloud-native/container-registry/what-is-docker/
- https://kubernetes.io/it/docs/concepts/overview/what-is-kubernetes/
- https://phoenixnap.com/kb/docker-image-vs-container
- https://www.techtarget.com/searchitoperations/definition/container-image
- https://www.aquasec.com/cloud-native-academy/container-security/container-images/
- https://opensource.com/article/21/8/container-image
- https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016
- https://www.infoworld.com/article/3204171/what-is-docker-the-spark-for-the-container-revolution.html
- https://www.techtarget.com/searchitoperations/feature/Dive-into-the-decades-long-history-of-container-technology
- https://www.vmware.com/it/topics/glossary/content/virtual-machine.html
- https://it.wikipedia.org/wiki/Macchina_virtuale
- https://it.wikipedia.org/wiki/Virtualizzazione_a_livello_di_sistema_operativo
- https://it.wikipedia.org/wiki/Virtualizzazione
- https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/7.0_release_notes/sect-red_hat_enterprise_linux-7.0_release_notes-linux_containers_with_docker_format-comparison_with_virtual_machines
- https://azure.microsoft.com/it-it/resources/cloud-computing-dictionary/what-is-a-virtual-machine/#overview
- https://docs.vmware.com/en/vCenter-Converter-Standalone/6.2/com.vmware.convsa.guide/GUID-105598AA-F443-4339-AC6D-E7664C1A9B4F.html
- https://bryanavery.co.uk/docker-basic-concepts-components-and-advantages/
- https://www.omniagroup.it/containerizzazione-delle-applicazioni/

- https://www.hpe.com/it/it/what-is/containers.html
- https://www.simform.com/blog/docker-use-cases/
- https://www.suse.com/c/rancher_blog/how-to-build-and-run-your-own-container-images/
- https://jfrog.com/knowledge-base/a-beginners-guide-to-understanding-and-building-docker-images/
- https://www.dataset.com/blog/create-docker-image/
- https://opensource.com/article/20/12/containers-101

## [3. Docker]

I contenuti di questo capitolo sono tratti da:

- https://docs.docker.com/get-started/overview/
- https://docs.docker.com/config/daemon/
- https://docs.docker.com/engine/reference/builder/
- https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
- https://docs.docker.com/storage/volumes/
- https://docs.docker.com/engine/swarm/secrets/
- https://it.wikipedia.org/wiki/Binary_large_object
- https://en.wikipedia.org/wiki/Docker_(software)
- https://docs.docker.com/network/
- https://docs.docker.com/registry/
- https://docs.docker.com/engine/extend/legacy_plugins/
- https://www.geeksforgeeks.org/docker-concept-of-dockerfile/
- https://www.geeksforgeeks.org/what-is-docker-images/
- https://bryanavery.co.uk/docker-basic-concepts-components-and-advantages/
- https://www.redhat.com/it/topics/containers/what-is-docker

## [4. Tools di Docker]

I contenuti di questo capitolo sono tratti da:

- https://collabnix.github.io/dockertools/
- https://docs.docker.com/engine/reference/commandline/build/
- https://docs.docker.com/engine/sbom/
- https://docs.docker.com/desktop/extensions/
- https://docs.docker.com/engine/scan/
- https://docs.docker.com/compose/
- https://indomus.it/formazione/docker-compose-cose-a-cosa-serve-e-perche-si-usa/
- https://phoenixnap.com/kb/docker-compose
- https://github.com/collabnix/dockertools#open-source-developer-tools

## [5. Limiti di Docker]

I contenuti di questo capitolo sono tratti da:

- https://docs.docker.com/engine/swarm/

- https://www.youtube.com/watch?v=X48VuDVv0do
- https://www.youtube.com/watch?v=BgrQ16r84pM
- https://kubernetes.io/it/docs/concepts/overview/what-is-kubernetes/
- https://circleci.com/blog/docker-swarm-vs-kubernetes/
- https://www.bmc.com/blogs/kubernetes-vs-docker-swarm/
- https://www.ibm.com/cloud/blog/docker-swarm-vs-kubernetes-a-comparison
- https://phoenixnap.com/blog/kubernetes-vs-docker-swarm
- https://www.redhat.com/it/topics/containers/what-is-docker
- https://www.redhat.com/it/topics/containers/what-is-kubernetes
- https://www.tigera.io/learn/guides/container-security/docker-security/
- https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/
- https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
- https://kubernetes.io/docs/concepts/architecture/
- https://kubernetes.io/docs/concepts/architecture/nodes/
- https://kubernetes.io/docs/concepts/architecture/control-plane-node-communication/

## [6. Casi d'uso di Docker]

I contenuti di questo capitolo sono tratti da:

- https://www.airpair.com/docker/posts/8-proven-real-world-ways-to-use-docker
- https://www.techtarget.com/searchitoperations/tip/6-use-cases-for-Docker-containers-and-when-to-pass
- https://www.clickittech.com/devops/docker-use-cases/
- https://www.simform.com/blog/docker-use-cases/
- https://data-flair.training/blogs/docker-use-cases/
- https://en.wikipedia.org/wiki/CI/CD
- https://www.geekandjob.com/wiki/ci-cd