

UNIVERSITÀ DEGLI STUDI DEL MOLISE

DIPARTIMENTO DI BIOSCIENZE E TERRITORIO



CORSO DI LAUREA IN INFORMATICA

Tesi di Laurea in
AUTOMATED SOFTWARE DELIVERY

**COMPRENSIONE DELLE PRATICHE DEGLI
SVILUPPATORI PER IL MIGLIORAMENTO
DELLE PRESTAZIONI DI DOCKFILE: UNO
STUDIO EMPIRICO**

Relatore:
Chiar.mo Prof.
Scalabrinio Simone

Correlatore:
Dott.
Rosa Giovanni

Candidato:
Iannone Mattia
Matr. 162697

ANNO ACCADEMICO 2021-2022

*Alla mia amata famiglia, ai miei cari nonni,
a coloro che, anche nei momenti più ardui
mi hanno costantemente incoraggiato a non arrendermi.*

Indice

1	Introduzione	5
1.1	Contesto applicativo	5
1.2	Motivazioni e obiettivi	7
1.3	Risultati raggiunti	8
1.4	Organizzazione tesi	8
2	Fondamenti teorici	9
2.1	Container	9
2.1.1	Cenni evolutivi	9
2.1.2	Container vs Virtual Machine	10
2.2	Architettura Docker	12
2.2.1	Docker Engine	12
2.2.2	Immagini	13
2.2.3	Docker Hub e registri	13
2.2.4	Costruzione immagini e strati	13
2.3	Dockerfile	15
2.3.1	Struttura e istruzioni	15
3	Stato dell'arte	20
3.1	Improvements performance of Dockerfile	20
4	Design dello studio empirico	22
4.1	Obiettivi dello studio	22
4.2	Planning	22
4.3	Contesto	23
4.4	Procedura sperimentale	25
5	Risultati dello studio empirico	26
5.1	Base Image	31
5.2	Modify installation	32
5.3	Cashing operations	33
5.4	Restructure files	35
5.5	Remove unnecessary data-packages-dependencies	37
5.6	Tweaks	38
5.7	More specific	40
5.8	Environment variables	41
5.9	Ignore files	42
6	Conclusioni e sviluppi futuri	43
7	Ringraziamenti	45

1 Introduzione

1.1 Contesto applicativo

In passato, l'infrastruttura software era basata su server fisici dedicati, ognuno con un sistema operativo e un'applicazione specifica. Questo approccio comportava una limitata flessibilità dell'utilizzo delle risorse essendo ogni server progettato per supportare una singola applicazione. Server che funzionavano al di sotto delle loro potenzialità e eccessiva manutenzione hanno portato alla necessità di trovare soluzioni alternative che potessero ottimizzare l'utilizzo delle risorse hardware, fornendo una maggiore flessibilità e un migliore controllo dell'ambiente. La virtualizzazione ha rivoluzionato il mondo software introducendo nuovi paradigmi e soluzioni che hanno permesso di ottimizzare l'utilizzo delle risorse hardware. L'industria software però, ha subito una profonda trasformazione grazie all'adozione di nuove tecnologie che hanno rivoluzionato il modo in cui le applicazioni vengono sviluppate e distribuite. In particolare, la containerizzazione ha avuto un impatto significativo sulla gestione delle applicazioni in ambienti moderni. Docker, uno dei principali strumenti di containerizzazione, è diventato sempre più popolare grazie alla sua capacità di creare e distribuire immagini di container leggere e portabili. Questa tecnologia ha permesso agli sviluppatori di ridurre significativamente i tempi di sviluppo e distribuzione delle applicazioni, semplificando la gestione delle dipendenze e fornendo un ambiente di esecuzione altamente affidabile e scalabile. Oggi, l'adozione di Docker è diventata una pratica comune in molti settori dell'industria software. Aziende di ogni dimensione, dalle start-up alle grandi imprese, stanno adottando la containerizzazione come parte integrante della loro infrastruttura tecnologica. Questo cambiamento di abitudini ha comportato la necessità di formare nuove competenze e di adattarsi a nuovi modelli di sviluppo e gestione delle applicazioni. Tuttavia, l'adozione di Docker e la containerizzazione hanno dimostrato di essere un'opzione vincente per affrontare le sfide dell'evoluzione dell'industria informatica, consentendo di aumentare l'efficienza, la scalabilità e la portabilità delle applicazioni. Recentemente, la piattaforma definita da una numerosa comunità di sviluppatori e programmatore per lo scambio di conoscenze di sviluppo, Stack Overflow, ha pubblicato i risultati del sondaggio 2022 per scoprire nuove tendenze. I container, principalmente Docker, hanno ottenuto risultati notevoli. Docker infatti, è lo strumento di sviluppo n. 1 più amato, ottenendo 8,966 voti(vedi 1).

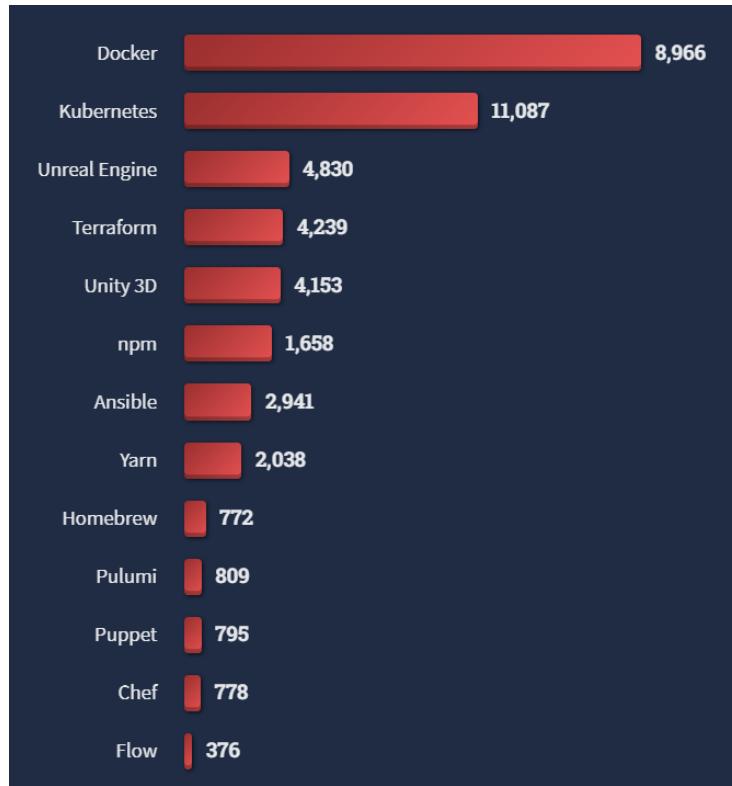


Figura 1: Tecnologie più amate 2022 [1]

Gartner, società fornitrice di analisi e previsioni, ritiene che il 70% delle organizzazioni eseguirà più app containerizzate nell'anno 2023. È innegabile che Docker stia diventando un'importante risorsa per gli sviluppatori professionisti (vedi 2).

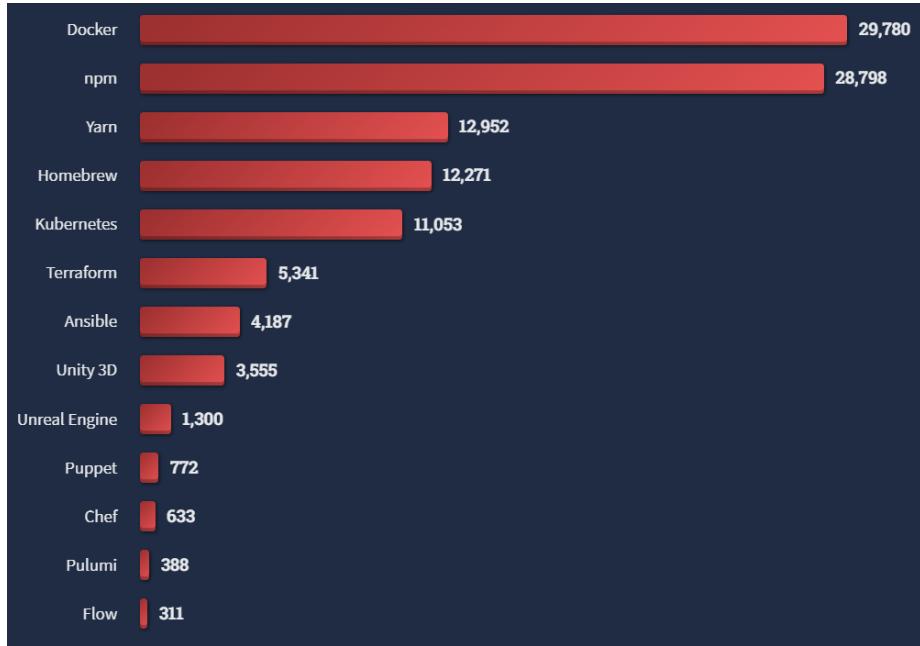


Figura 2: Tecnologie più popolari 2022 [2]

1.2 Motivazioni e obiettivi

Nonostante Docker sia una tecnologia affermata e sempre più utilizzata nella gestione e distribuzione di applicazioni, Dockerfile può rappresentare una sfida per gli sviluppatori. I file Docker possono presentare problematiche legate alle prestazioni. Un Dockerfile mal progettato può causare problemi di scalabilità e di efficienza nell'esecuzione dell'applicazione. Purtroppo, non tutti gli sviluppatori sono consapevoli di queste problematiche, e spesso si concentrano solo sulla funzionalità dell'applicazione, trascurando le questioni di performance. Un Dockerfile ben strutturato infatti, può migliorare significativamente le performance dell'applicazione, ridurre i tempi di avvio e di esecuzione. Sfortunatamente, c'è una lacuna di conoscenza su come i progetti basati su container si evolvono effettivamente e vengono mantenuti.

L'obiettivo della ricerca è quello di capire come gli sviluppatori modificano il Dockerfile quando la loro volontà è quella di migliorare le performance e perché lo fanno.

Le implicazioni di questo studio aiuteranno i professionisti, sviluppatori, costruttori di strumenti e educatori a migliorare la qualità dei progetti Docker.

1.3 Risultati raggiunti

Si è effettuata un'ampia analisi manuale di 400 commit, ognuna di esse comprende modifiche attuate al fine di migliorare le performance del Dockerfile. L'approccio allo studio è stato quello di analizzare commit e annotare le modifiche fatte con dei tag. Studiando questi tag ci si è resi conto che molti appartengono alle stesse aree di interesse di refactoring. L'analisi mostra che il 26,42% degli sviluppatori analizzati nelle 400 commit, ha focalizzato la propria attenzione sull'immagine di base del proprio Dockerfile e come modificarla. Un'altra grande fetta di sviluppatori, il 20,92%, ha ritenuto necessario sfruttare le potenzialità offerte dal caching e focalizzarsi nel ordinare i comandi del Dockerfile al fine di ottenere una migliore prestazione di build. È emerso come molti sviluppatori siano parecchio attenti nel rimuovere le dipendenze, i pacchetti e i dati non necessari alla build, il 21,28% di loro infatti, ha attuato un refactoring indirizzato alla rimozione di elementi superflui per la costruzione dell'immagine. La restante fetta delle commit ha mostrato come esistono molti modi per ottimizzare un Dockerfile che verranno analizzati più da vicino nella sezione 5.

1.4 Organizzazione tesi

Successivamente a questo capitolo introduttivo seguirà una panoramica sul Background.

Nel capitolo 3 sarà descritto lo stato dell'arte.

Nel capitolo 4 sarà presentato lo studio effettuato.

Nel capitolo 5 saranno presentati i risultati dello studio.

Nel capitolo 6 si trarranno le conclusioni e si progetteranno sviluppi futuri che l'analisi potrebbe avere.

2 Fondamenti teorici

In questo capitolo tratteremo la tecnologia Docker, che successivamente al suo avvento, ha rivoluzionato il mondo della distribuzione delle applicazioni. Con il suo innovativo approccio alla creazione, gestione e distribuzione di software ha apportato una significativa rivoluzione, trasformando il modo in cui le applicazioni vengono sviluppate e distribuite. Docker è una piattaforma che permette di pacchettizzare le applicazioni, renderle disponibili verso altri sistemi e gestire applicazioni in modo efficiente ed isolato, senza dover configurare l'infrastruttura sottostante.

Il processo di "pacchettizzazione" è comunemente chiamato containerizzazione.

2.1 Container

Prima di arrivare a Docker, bisogna scavare nella storia di un argomento precursore sul quale Docker si è costruito, i container. Nel mondo del software industriale si è in costante ricerca di tecniche e pratiche di ottimizzazione e riduzione del costo della gestione, manutenzione e portabilità hardware e software.

Quando parliamo di container, citiamo uno dei temi più caldi del cloud computing. Questa tecnologia permette di eseguire un'applicazione e le sue dipendenze in processi separati, isolati da altre risorse. L'idea è quella di virtualizzare solo una piccola parte della macchina, invece che l'intero sistema. Il risultato sono i container. Questo metodo di virtualizzazione innovativo semplifica notevolmente il deployment e i processi di implementazione, rappresentando una concreta alternativa alle Virtual Machine (VMs).

2.1.1 Cenni evolutivi

Il concetto alla base dei container nasce già nel lontano 1979 su chroot UNIX, un sistema in grado di fornire ai processi degli spazi isolati all'interno dello storage in uso. Lo si può vedere come un precursore della tecnologia dei container. Nel 2000, fu introdotta una soluzione che si avvicina di più a quella attualmente in uso: FreeBSD jail. Questa tecnologia permetteva la suddivisione del sistema in numerosi sottosistemi isolati, noti come jail, con la capacità aggiuntiva di isolare anche i file di sistema, gli utenti e la rete. Nel 2001, nasce una soluzione simile a jail, Linux VServer. Questa tecnologia di virtualizzazione si proponeva di

partizionare le risorse di un computer nei "security context", all'interno dei quali si trovava un sistema VPS (Virtual Private Server). In questi anni si gettano le prime basi per gli spazi controllati in Linux. Nasce nel 2008 il progetto LinuX Container, progetto che consentì il controllo e la limitazione di risorse da parte di un processo o più. Nel 2013, avviene un evento significativo: l'arrivo di Docker, il sistema di container Linux più popolare che ha segnato un cambiamento epocale. Docker ha semplificato notevolmente l'uso dei container e ha introdotto nuove funzionalità, come la creazione delle immagini dei container, che semplificano la distribuzione delle applicazioni su diverse piattaforme e ambienti di sviluppo. La tecnologia Docker è diventata uno standard de facto per la gestione dei container e ha avuto un impatto significativo sull'industria dell'informatica e della gestione delle applicazioni.

2.1.2 Container vs Virtual Machine

La virtual machine è un altro modo per creare un ambiente isolato. Una VM è effettivamente un singolo computer che risiede all'interno di una macchina host, come più macchine virtuali possono risiedere all'interno di una singola macchina. Le virtual machine vengono create virtualizzando l'hardware sottostante della macchina host. L'hardware è virtualizzato e suddiviso, con un pezzo che rappresenta l'hardware sottostante, su cui può essere eseguita una macchina virtuale. Il componente chiave per la tecnologia di virtualizzazione è l'hypervisor, anche chiamato Virtual Machine Monitor, è un software di virtualizzazione che consente la creazione delle VM, la loro gestione e la loro fornitura di risorse. In altre parole, senza questo componente non si avrebbe la capacità di eseguire più sistemi operativi su una singola macchina.

Questo processo porta con sé un notevole sovraccarico computazionale. Inoltre, poiché ciascuna VM rappresenta fondamentalmente una macchina indipendente, ognuna ospita il proprio sistema operativo, che richiede decine di gigabyte di spazio di archiviazione e un processo di installazione maggiore, cosa che deve essere eseguita ogni volta si desidera una nuova VM (vedi 3).

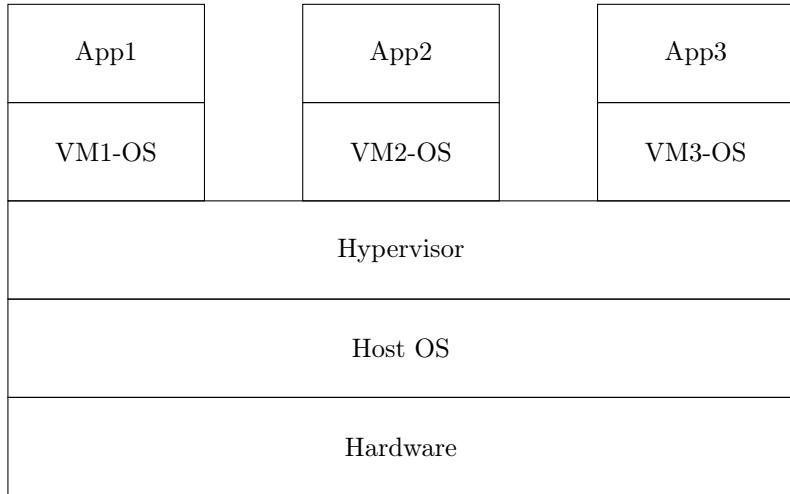


Figura 3: Struttura macchine virtuali

I container, come le virtual machine, risiedono su una macchina host e ne sfruttano le risorse. In particolare con la caratteristica di virtualizzare il sistema operativo host. Ciò significa che i container non necessitano del proprio sistema operativo, il che li rende molto più leggeri delle macchine virtuali e di conseguenza, più veloci da avviare.

Tracciando una differenza a livelli paralleli tra i container e le macchine virtuali, notiamo come il livello hypervisor viene sostituito con il Docker Daemon. Quest'ultimo funge da intermediario tra sistema operativo e i container. (vedi 4). Le macchine virtuali presentano il problema delle duplicazioni, poiché molte funzionalità dei sistemi operativi guest sono già presenti nel sistema operativo host. Al contrario, i container consentono di sfruttare e inserire solo le risorse di cui l'applicazione necessita, evitando di installare tutto l'ambiente del sistema operativo. A conclusione di questo parallelismo container e virtual machine, la tecnologia di containerizzazione offre una soluzione intelligente per garantire un'efficace isolamento o disaccoppiamento delle applicazioni senza compromettere velocità ed efficienza.

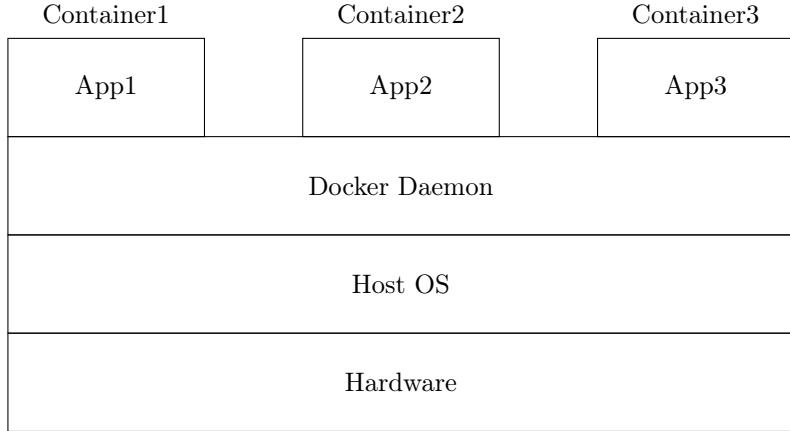


Figura 4: Struttura dei container

2.2 Architettura Docker

2.2.1 Docker Engine

In Docker troviamo la presenza di un'unità strutturata secondo una modulare gerarchia, anche conosciuta come fulcro del sistema stesso, il Docker Engine. Esso lo si può vedere come il componente principale, in quanto permette di eseguire ambienti virtuali isolati in grado di ospitare applicazioni e servizi.

Il Docker Engine, pertanto, assume un ruolo di primaria importanza, garantendo la possibilità di creare, distribuire e gestire i container con la massima affidabilità. Basato secondo lo standard fornito dall'OCI (Open Container Initiative), ha come componenti principali il Docker Client, Docker Daemon, ContainerD e RunC. Questi componenti interagiscono nel modo seguente: La Docker Cli permette di avviare il container tramite un comando apposito, il quale viene poi convertito dal Docker client in un formato comprensibile dal Docker Daemon. Quest'ultimo, a sua volta, in seguito alla ricezione dei comandi per la creazione del container, richiama il componente ContainerD. Tuttavia, è importante sottolineare che la responsabilità effettiva della creazione del container non è demandata direttamente a ContainerD, ma è invece affidata al componente RunC. Quest'ultimo si interfaccia con il kernel del sistema operativo per riunire tutti i componenti necessari alla creazione del container.

2.2.2 Immagini

In precedenza abbiamo fatto riferimento alle immagini Docker, ma non abbiamo ancora esplorato a fondo il loro funzionamento e la loro struttura. Possiamo identificare le immagini come dei container stoppati, dei modelli da cui poter, successivamente, eseguire più container. Ogni immagine contiene una specifica configurazione del sistema operativo, librerie e dipendenze, insieme (eventualmente) all'applicazione stessa. Ciò significa che, con le immagini Docker è possibile creare facilmente e rapidamente l'ambiente sottostante. Inoltre, le immagini Docker sono progettate per essere leggere e efficienti, riducendo i tempi di avvio dei container e il consumo di risorse del sistema.

2.2.3 Docker Hub e registri

Le immagini possono essere scaricate da un registro in cui quest'ultime sono archiviate. Il registro pubblico è comunemente conosciuto come Docker Hub. Questo "raccoglitore" di immagini ci permette di selezionare l'immagine più adatta, in base alle necessità.

Oltre al Docker Hub esistono anche altri registri, ogni utente connesso con il Docker Hub può condividere la propria immagine nel proprio registro attraverso il comando 'docker push'. Questo spiega l'ulteriore distinzione tra registri ufficiali e non ufficiali, quest'ultime infatti potrebbero non essere sicure.

2.2.4 Costruzione immagini e strati

Per scaricare l'immagine da un registro sul Docker locale è possibile utilizzare il comando 'docker pull'. Tale comando effettuerà il download dell'immagine e permetterà l'esecuzione di quest'ultima.

```
$ docker pull redis
Using default tag: latest
latest: Pulling from library/redis
9e3ea8720c6d: Pull complete
7bb0a593ef8e: Pull complete
1563ab48b627: Pull complete
e71aa54519a0: Pull complete
e43bffb29bfd: Pull complete
a8fb6ae3ba1b: Pull complete
```

```
Digest: sha256:ea30bef6a1424d032295b90db20a86  
9fc8db76331091543b7a80175cede7d887  
Status: Downloaded newer image for redis:latest  
docker.io/library/redis:latest
```

Osservando nel caso precedentemente mostrato l'esecuzione del comando 'pull' verso l'immagine 'redis', è possibile notare come essa avvenga a strati impilati uno seguito dall'altro. Al termine, tutti insieme, costituiranno un oggetto unico. Docker verifica sempre se nella cache locale è presente l'immagine o meno, se non è presente effettua l'operazione di pulling.

Il comando 'docker image ls' permette di elencare tutte le immagini presenti localmente, in particolare il comando mostra le informazioni di base per ciascuna immagine, come il nome dell'immagine, il suo ID, la dimensione e il tag (se presente).

```
$ docker image ls  
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE  
redis            latest    116cad43b6af   7 days ago   117MB
```

Per meglio comprendere il processo di avvio di un container, è utile considerare l'esempio che segue attraverso il comando 'docker run'. In questo caso, il logs del container ci permettono di capire che il container è in esecuzione ed è pronto per l'utilizzo.

```
$ docker run redis  
  
1:M 10 May 2023 16:41:44.403 # ..some logs  
  
1:M 10 May 2023 16:41:44.405 * Ready to accept connections
```

Per avere la certezza dell'esecuzione del container, Docker fornisce un comando che permette di controllare i container presenti in locale. In particolare, il flag '-a' specifica la volontà di vedere tutti i container, in esecuzione e non.

```
$ docker ps -a  
CONTAINER ID IMAGE COMMAND      CREATED      STATUS      PORTS  
89c40d828505 redis "docker-entr.." About a min Up..      6379/tcp
```

2.3 Dockerfile

Il Dockerfile è un normale documento di testo per la definizione di passaggi per creare una nuova immagine. Nasce dall'esigenza di convertire un'applicazione e renderla un container. Questo processo, comunemente chiamato "dockerizzazione", significa creare uno snapshot dell'applicazione stessa, in altre parole, la propria immagine. La creazione di immagini personalizzate tramite la definizione di un Dockerfile consente di adattare le immagini Docker alla singola infrastruttura attuale. Come già accennato in precedenza, la costruzione delle immagini avviene in modo stratificato, il concetto ritorna utile anche nella scrittura di un Dockerfile. Ogni istruzione, infatti, crea un livello nell'immagine. Dopo aver modificato il Dockerfile e ricostruito l'immagine, vengono ricostruiti solo i livelli che sono stati effettivamente modificati; questa è una delle ragioni per cui le immagini Docker sono così leggere, compatte e veloci rispetto ad altre tecnologie di virtualizzazione.

2.3.1 Struttura e istruzioni

L'obiettivo di questa sezione è di fornire una descrizione semi-dettagliata delle istruzioni principali per la definizione di un Dockerfile, che è necessario per la build dell'immagine.

Il Dockerfile non fa distinzione tra maiuscole e minuscole, tuttavia, la convenzione è scrivere i comandi in maiuscolo.

Alcune istruzioni creano nuovi livelli, altre aggiungono meta-dati all'immagine.

Istruzione FROM

Docker esegue determinate istruzioni in base all'ordine delle istruzioni nel Dockerfile. Il primo comando con la quale iniziare un Dockerfile è il l'istruzione FROM. Solo in alcuni casi specifici potrebbe non essere la prima istruzione, infatti si potrebbe avere la necessità di definire delle direttive parser, commenti e variabili globali con l'istruzione ARG.

Istruzione ARG

In alcuni casi può avere senso specificare uno o più parametri da passare come argomento al FROM. Questo è possibile attraverso la keyword ARG. Richiamando l'istruzione ARG dopo il FROM rendiamo visibili gli argomenti anche all'interno del processo di costruzione dell'immagine.

Istruzione ENV

Le variabili d'ambiente vengono dichiarate mediante l'istruzione ENV e possono essere anche utilizzate all'interno dell'immagine Docker durante la sua esecuzione.

Istruzione RUN

L'istruzione RUN è una delle istruzioni più utilizzate nel Dockerfile, la sua funzione è quella di installare qualsiasi pacchetto come nuovo livello all'interno dell'immagine durante la compilazione. Sostanzialmente aggiunge un blocco immutabile all'immagine. Durante la fase di build, l'istruzione di build successiva a quella di RUN appena definita, sarà costruita a partire dall'immagine ottenuta dall'applicazione di tutti i layer precedenti. L'ultima istruzione RUN appena definita sarà l'ultima in ordine di comparizione. Il concetto appena osservato è chiave nelle logiche del Dockerfile e in Docker in generale. La build avviene per composizione di layer successivi definiti con buona approssimazione da ogni riga del Dockerfile a partire da FROM.

```
FROM Ubuntu
RUN apt-get update && apt-get install curl
```

Istruzione COPY

L'istruzione COPY viene utilizzata per copiare file e directory dall'host locale da incidere all'interno del contenitore. La sintassi è la seguente:

```
COPY test /newdir/test
```

Istruzione ADD

L'istruzione ADD, esattamente come l'istruzione COPY, viene utilizzata per copiare i file o le directory dal sistema host all'interno del filesystem dell'immagine Docker, ma a differenza dell'istruzione COPY, supporta anche l'estrazione di file tar, il supporto degli url e la gestione delle espressioni regolari.

Istruzione WORKDIR

Per impostare una directory di lavoro per qualsiasi istruzione si impiega l'utilizzo del comando WORKDIR. Per capire nello specifico il suo utilizzo può essere utile valutare un esempio nella quale l'istruzione RUN verrà eseguita nel percorso /var/source.

```
WORKDIR /var  
WORKDIR source  
RUN apt-get update
```

Istruzione VOLUME

L'istruzione VOLUME crea una directory nel file system dell'immagine, che può essere successivamente utilizzata per montare i volumi dall'host Docker o dagli altri container. L'uso di VOLUME è utile per separare i dati del container dall'immagine stessa, in modo che i dati possano essere facilmente gestiti e salvati separatamente dall'immagine. Ciò consente anche di sostituire facilmente i dati del container senza dover ricreare l'intero container.

Istruzione EXPOSE

Il comando EXPOSE viene utilizzato per esporre una o più porte del container. Nasce dall'esigenza di dover far comunicare il container con il mondo esterno, in modo tale che quest'ultimo possa inviare richieste al container. Per impostazione predefinita, il motore Docker assegnerà TCP come protocollo.

Istruzione CMD

L'istruzione CMD è l'istruzione di default per l'esecuzione di un container. Fondamentalmente significa che allo start di un container a partire dall'immagine che stiamo definendo, verrà invocata questa istruzione. In ogni Dockerfile è possibile avere una sola istruzione CMD

Istruzione ENTRYPOINT

DA RIVEDERE L'istruzione ENTRYPOINT consente di configurare un container che verrà eseguito come eseguibile. È simile all'istruzione CMD, perché consente anche di specificare un comando con dei parametri, ma la differenza è che i parametri non verranno ignorati quando il container Docker viene eseguito con i parametri passati tramite riga di comando. Considerando le seguenti istruzioni, Docker stamperà nei log del container "running".

```
CMD "running"  
ENTRYPOINT echo
```

RUN, CMD e ENTRYPOINT a confronto

Una distinzione importante da fare all'interno del Dockerfile è tra le istruzioni RUN e CMD. Mentre entrambe le istruzioni vengono utilizzate per definire un

comando all'interno del container, è importante comprendere la differenza tra di esse per evitare confusione e problemi durante la build.

RUN esegue effettivamente un comando e conferma il risultato.

CMD non esegue nulla al momento della compilazione, ma specifica il comando previsto per l'immagine all momento della creazione del container.

Di fondamentale importanza un'altra distinzione, quella tra CMD e ENTRYPOINT. Come analizzato in precedenza, un Dockerfile deve contenere almeno un comando tra i due, ma con delle importanti differenze.

CMD deve essere utilizzato per definire un comportamento di default.

ENTRYPOINT deve essere utilizzato per eseguire comandi specifici per il container che, salvo precise eccezioni, non dovrebbero essere modificati per personalizzazioni.

A seguire è disponibile un esempio di Dockerfile completo con diverse istruzioni utili per la creazione di un'immagine definisce un'immagine basata su Ubuntu 20.04 LTS, che utilizza Python 3 per eseguire un'applicazione web. Le righe con la presenza del # sono dei commenti e non sono considerati in fase di build.

```

# Definizione dell'immagine di base
FROM ubuntu:latest

# Definizione di un ARG
ARG APP_VERSION=1.0.0

# Definizione di una variabile d'ambiente
ENV APP_HOME=/usr/src/app

# Aggiornamento dei pacchetti
RUN apt-get update && apt-get install -y \
    git \
    python3 \
    python3-pip

# Creazione della directory di lavoro
WORKDIR $APP_HOME

# Copia dei file dell'applicazione nella directory di lavoro
COPY . .

# Aggiunta di un file alla directory di lavoro
ADD config.ini .

# Definizione di un volume
VOLUME /data

# Esposizione della porta 8080
EXPOSE 8080

# Definizione del comando di default
CMD ["python3", "app.py"]

```

3 Stato dell'arte

In questa sezione si è cercato di analizzare lo stato dell'arte nell'ambito del refactoring del Dockerfile e del miglioramento delle performance.

In questa sezione saranno esposti i problemi di performance legati alla scrittura di un Dockerfile, sintomatici di una scarsa conoscenza, da parte degli sviluppatori, delle tecniche di miglioramento del file stesso.

3.1 Improvements performance of Dockerfile

Migliorare le performance di un Dockerfile, indica il processo di ottimizzazione di tecniche di scrittura al fine di ottimizzare il processo di build, ridurre il numero di layer e di dipendenze e in generale ridurre le dimensioni dell'immagine. La creazione di immagini Docker è una pratica comune nell'industria informatica moderna. Grazie alla loro portabilità e alle loro caratteristiche di isolamento, i container Docker sono ampiamente utilizzati per la distribuzione di applicazioni, servizi e microservizi. Tuttavia, la creazione di un'immagine Docker affidabile e ottimizzata può rappresentare una sfida per molti sviluppatori. In particolare, il processo di build del Dockerfile, ovvero il file di configurazione che descrive come creare l'immagine, richiede una buona conoscenza delle migliori pratiche di sviluppo e di configurazione. Lo studio [6] condotto da Zhuo Huang, Song Wu, Song Jiang e Hai Jin1 propone una soluzione al problema della composizione dell'immagine, processo che infatti può essere inaspettatamente lento. La tecnica di risoluzione, chiamata FastBuild è una soluzione efficace testata che, successivamente alle sperimentazioni con immagini e Dockerfile ottenuti da DockerHub, può migliorare la velocità di costruzione fino a 10 volte e ridurre i dati scaricati del 72%.

Essendo Docker una tecnologia di tendenza popolare, un comportamento generalmente registrato è quello di "dockerizzare" le applicazioni e i servizi senza conoscere pattern e logiche di scrittura che ottimizzerebbero il processo. Come risulta dallo studio [9] condotto da Yiwen Wu, Yang Zhang, Tao Wang, e Huaimin Wang, analizzando 6334 progetti, si evince che 5.309 (83,8%) hanno presenza di code smells all'interno del Dockerfile, mentre solo 1.025 (16,2%) sono considerabili "sani". Sono molto comuni quindi casi di Dockerfile infettati da code smells che non rispettano le pratiche di scrittura del Dockerfile stesso. Adottare queste pratiche può avvantaggiare la manutenzione. In accordo con lo studio [7]

condotto da Emna Ksontini, Marouane Kessentimi, Thiago do N. Ferreira e Foyzul Hassan è utile studiare il refactoring dei Dockerfile, analizzando quello che effettivamente avviene quando l'intenzione degli sviluppatori è quella di rifactorizzare il codice. Il suddetto articolo si è focalizzato sullo studio generale, sono stati presi in analisi infatti, Dockerfile, Docker-compose e altri file cambiati nel processo di refactoring. Lo studio è stato effettuato analizzando manualmente 193 commit da diversi progetti unici con l'obiettivo di refactoring del codice. Il processo riporta che 44 commit hanno refactoring correlato a Dockerfile, 51 commit hanno refactoring correlato a Docker-compose e 55 commit di refactoring di codice regolari (ad esempio, Java, C e così via) a causa di modifiche in Dockerfile o Docker-compose. Introducendo 24 categorie di refactoring e debito tecnico specifiche per Docker definendo diverse best practice. La figura 5 riassume le tecniche di refactoring nell'ambiente Dockerfile e le rispettive micro-categorie. Come facilmente intuibile, l'ambito di interesse è notevolmente più ampio rispetto a quello analizzato nel nostro studio, ciò sottolinea la mancanza di una adeguata considerazione approfondita dell'aspetto prestazionale.

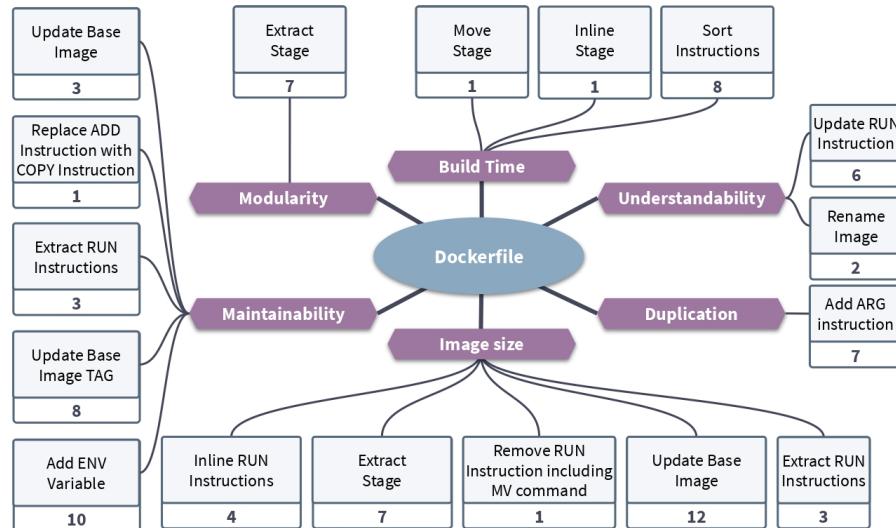


Figura 5: Refactoring specifici relativi al Dockerfile (vedi5)

4 Design dello studio empirico

In questo capitolo, verrà presentato il caso di studio. Si effettuerà un'analisi delle diverse commit presenti sulla piattaforma GitHub, con particolare riguardo ai messaggi di commit che manifestino la volontà di ottimizzare le prestazioni. È di fondamentale importanza che i professionisti dello sviluppo che si occupano della tematica delle prestazioni connesse al Dockerfile acquisiscano, basandosi sull'esperienza di altri sviluppatori, le opportune tecniche atte a migliorare l'efficienza delle prestazioni stesse.

4.1 Obiettivi dello studio

L'obiettivo principale dello studio è quello di collezionare commit su GitHub in cui gli sviluppatori manifestano, attraverso il messaggio di commit, di migliorare le performance dei Dockerfile, individuare le motivazione delle modifiche e studiare i cambiamenti al fine di identificare pattern strategici mirati al consolidamento delle performance.

4.2 Planning

La domanda (research question) che ci si è posti è principalmente è:

RQ: Quali strategie usano gli sviluppatori per migliorare le performance nei Dockerfile?

Per fornire una risposta esaustiva alla domanda posta, è stato indispensabile porci il problema di individuare commit "ottimali" che potessero servire come base per l'analisi delle performance. Per conseguire tale obiettivo, è stato necessario approfondire la vera natura del concetto di "performance" e la sua traduzione pratica in ambito della creazione di immagini. Quando si parla di performance, infatti, la dimensione dell'immagine e il tempo di build sono spesso considerati fattori critici. Ci sono diverse ragioni per cui questi aspetti possono influire sulle prestazioni del Dockerfile. In primo luogo, per ovvie ragioni, più le dimensioni aumentano e più le performance generali tendono a diminuire. Immagini più grandi richiedono più tempo per essere scaricate o distribuite su diverse macchine, e possono richiedere anche più tempo per essere eseguite. Questo può avere un impatto significativo sulla velocità di deploy dell'applicazione e sulla scalabilità del sistema. In secondo luogo, il tempo di build del

Dockerfile può influire sulla velocità di sviluppo, test e rilascio in fase di produzione di un'applicazione. Più il tempo di build è lungo, più tempo ci vorrà per testare e validare le modifiche apportate al Dockerfile, e questo può rallentare il processo di sviluppo. Inoltre, tempi di build più lunghi possono aumentare il costo del processo di sviluppo, in quanto richiedono più risorse di calcolo per eseguire le operazioni di build. Mediante l'apporto di un dataset dello stato dell'arte, oggetto di analisi in studi come [8], composto da 9.4M di Dockerfile unici, è bastato studiare lo schema e interrogare mediante query la base di dati per ottenere le commit desiderate.

4.3 Contesto

Il dataset considerato è interamente dedicato all'ambiente Docker e raccoglie esclusivamente commit riguardanti le modifiche di Dockerfile, Docker-compose e dell'ecosistema Docker in generale. Complessivamente, sono disponibili 9,4M di commit potenzialmente analizzabili. La popolazione presa in considerazione per l'analisi sono le commit che vanno a modificare Dockerfile, in particolare quelle che contengono modifiche a questo file di configurazione avendo come obiettivo cercare i messaggi di commit mostranti intenzionalità di miglioramenti di performance, è stato necessario costruire un algoritmo in grado di gestire al meglio queste evenienze. L'algoritmo sviluppato è stato implementato in Node.js, sfruttando le numerose librerie di supporto disponibili. In particolare, sono state di grande aiuto due librerie: Natural [3], utilizzata per l'elaborazione del linguaggio naturale (Natural Language Processing - NLP), e Snowball [5], anch'essa adottata nell'ambito dell'NLP. La necessità di adottare queste librerie e le loro funzionalità è nata dall'esigenza di avere il più possibile risultati puliti, ovvero quelle commit che esplicitano chiaramente modifiche alle performance ed escludendo tutti i "falsi positivi". Non avendo a disposizione un modo per interpretare il linguaggio naturale per definire delle intenzioni, si è chiesto supporto da tre algoritmi di NPL [4], rispettivamente, Tokenization, rimozione stopword e Stemming.

Tokenization è il processo di divisione/suddivisione dei caratteri o delle parole di input in parti più piccole note come "token". La tokenizzazione è il passaggio iniziale nell'elaborazione del linguaggio naturale, che comporta la raccolta di dati e la loro suddivisione in parti in modo che una macchina possa comprenderli.

Stopword sono parole comuni in un determinato linguaggio, come articoli, preposizioni e congiunzioni, che spesso non forniscono alcuna informazione utile per l'elaborazione del testo. La rimozione di queste parole può migliorare l'efficienza e la precisione di algoritmi di elaborazione del linguaggio naturale, come l'analisi dei sentimenti o l'identificazione di temi. Il modulo "stopwords"

di Natural fornisce un elenco predefinito di stopwords per la lingua inglese.

Stemming è l'atto di ridurre una parola alla sua radice di parola (nota anche come forma base o radice). Lo stemming è una caratteristica del recupero e dell'estrazione dell'intelligenza artificiale, nonché della morfologia linguistica. Viene utilizzato dai motori di ricerca per indicizzare le parole. Ad esempio, parole come cat, catlike e catty verranno derivate dalla radice della parola cat e risulteranno della stessa natura una volta confrontate.

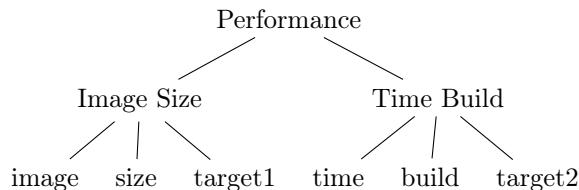
Per cercare di ottenere le commit relative alle performance, dopo aver combinato gli algoritmi di NPL, l'algoritmo ricerca nel dataset, in particolare nella colonna dei messaggi relativi ai commit, solo le commit che soddisfano le query. Le query utilizzate sono 2, una denominata image-size e l'altra time-build.

image-size

```
image AND size AND (decrease OR reduce OR cut OR small OR drop OR optimize)
```

time-build

```
time AND build AND (speed up OR decrease OR reduce OR cut OR small OR drop OR diminution)
```



```
target1 = [ decrease,reduce,cut,small,drop,optimize ]
target2 = [ speed up,decrease,reduce,cut,small,drop,diminution ]
```

I risultati presi in considerazione sono tutte quelle commit che presentano, prendendo come esempio image size, un'occorrenza di image, size, e almeno una chiave tra quelle presenti in target1. Come ultimo step dell'analisi, è stata condotta una ricerca manuale utilizzando le Api di GitHub. Sfruttando lo sha1 di ogni commit per studiare le modifiche apportate e analizzare le differenze tra le diverse versioni del codice e studiare quali cambiamenti hanno apportato una migliorria delle performance. Attività che, in mancanza di strumenti automatici di supporto, risulta laboriosa ma comunque efficace ai fini analitici.

4.4 Procedura sperimentale

L'analisi è stata condotta su circa 1200 commit, tutte analizzate manualmente e utilizzando le Api di GitHub, e tutte risultanti dalle due query precedentemente descritte. Analizzando attentamente queste commit, è stato possibile estrapolare pattern che possono essere applicati all'intera popolazione interessata. La procedura eseguita manualmente potrebbe presentare alcune sfide e potenzialmente comportare valutazioni soggettive, disattenzioni ed errori. Valutando le commit una ad una, etichettandole manualmente con dei tag specifici, sono state raccolte 400 commit utili, 200 per image-size e 200 per build-time. Per ogni commit sono stati salvati su una base di dati, portando il riferimento di url per un facile accesso alla commit, sha1 (codice univoco per ogni commit), messaggio di commit, nome dello sviluppatore e del repository, ranking del repository, tag e query che lo ha raccolto. Sebbene se tra tutti i 1200 messaggi di commit, tutti soddisfacevano i requisiti delle query, avanzandoli ad un analisi manuale si è dovuto escludere molti "falsi positivi" e molti repository chiusi e non più presenti su GitHub, pertanto, non utilizzabili. Questo perché, come specificato precedentemente, non è stato possibile valutare l'intenzionalità di un periodo/testo, ma solo la presenza di alcune keyword che si ipotizza possano essere utilizzate per notificare il miglioramento delle performance da parte di uno sviluppatore.

I tag raccolti sono stati studiati, meticolosamente quantificati e raggruppati secondo le aree di interesse.

5 Risultati dello studio empirico

Attraverso un attento e approfondito studio, sono state individuate ben 9 categorie di modifica che sono emerse come risultato dell'analisi di 400 commit. Questa suddivisione dettagliata ha consentito di ottenere una comprensione più approfondita e strutturata dei dati analizzati. Come è possibile notare nella tabella riassuntiva (vedi 6) ogni categoria rappresenta un insieme di caratteristiche e pattern generici di aree di interesse, essenziali per la comprensione complessiva del fenomeno. Pertanto, questa tabella fornisce una visione chiara e distinta della frequenza con cui queste categorie hanno giocato un ruolo fondamentale nel miglioramento delle performance da parte degli sviluppatori. Il risultato dello studio ha evidenziato un totale di 545 modifiche/tag effettuate.

CATEGORY	OCCURRENCES	%
Base Image	144	26,42%
Remove unnecessary data/packages/dependencies	116	21,28%
Cashing operations	114	20,92%
Modify installation	65	11,93%
Restructure files	53	9,72%
Ignore files	16	2,94%
More specific	15	2,75%
Tweaks	14	2,57%
Environment variables	8	1,47%
TOT	545	100,00%

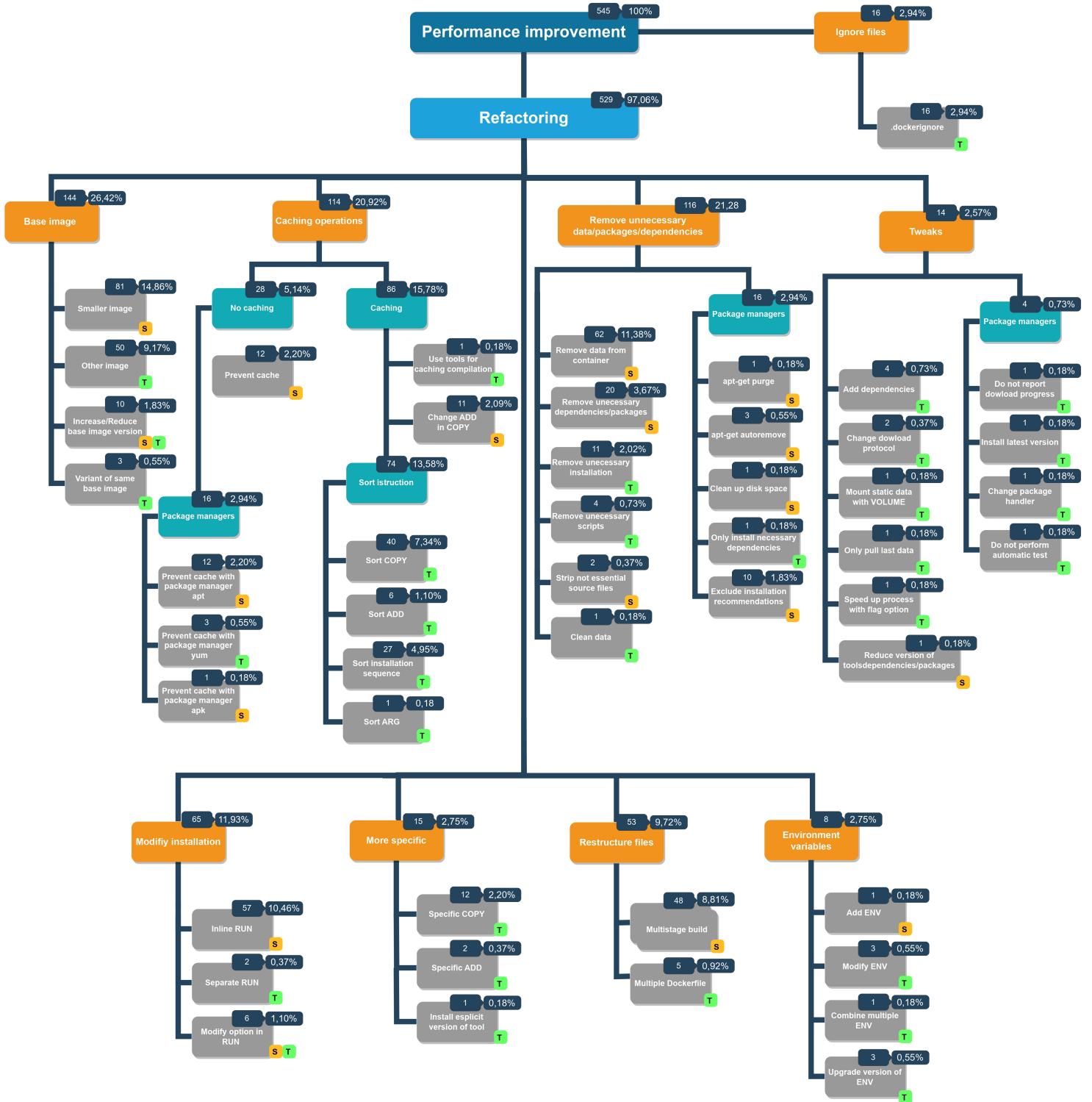
Figura 6: Categorie di refactoring trovate

Sono stati identificati un totale di 48 tag di interesse, utilizzati, ognuno, quando necessario. Complessivamente, questi tag sono stati impiegati per un totale di 545 volte, sottolineando la loro rilevanza e frequenza di utilizzo nel contesto dello studio delle performance. Nelle tabelle successive, c'è il fulcro della ricerca. La prima tabella mostra sotto la colonna tag, tutte le modifiche che sono state individuate nel Dockerfile e modifiche che vengono interessate nel processo di costruzione del Dockerfile, portando il numero di occorrenze al fine di offrire una impattante idea all'approccio di refactoring. La realizzazione della seconda tabella ha avuto l'obiettivo di tenere traccia di quante modifiche sono state raccolte da quale query. Facile intuire, che ci si può accedere per confrontare le occorrenze e valutare cosa migliorare.

TAG	CATEGORY	OCCURRENCES	%
Smaller Image	Base Image	81	14,86%
Other image	Base Image	50	9,17%
Increase/Reduce base image version	Base Image	10	1,83%
Variant of same base image	Base Image	3	0,55%
Inline RUN	Modify installation	57	10,46%
Separate RUN	Modify installation	2	0,37%
Modify option in RUN	Modify installation	6	1,10%
Prevent cache	Cashing operations No caching	12	2,20%
Prevent cache with package manager apt	Cashing operations No caching Package managers	12	2,20%
Prevent cache with package managers yum	Cashing operations No caching Package managers	3	0,55%
Prevent cache with package manager apk	Cashing operations No caching Package managers	1	0,18%
Tools for caching	Cashing operations Caching	1	0,18%
Change ADD in COPY	Cashing operations Caching	11	2,02%
Sort COPY	Cashing operations Caching Sort istruction	40	7,34%
Sort ADD	Cashing operations Caching Sort istruction	6	1,10%
Sort installation sequence	Cashing operations Caching Sort istruction	27	4,95%
Sort ARG	Cashing operations Caching Sort istruction	1	0,18%
Multistage build	Restructure files	48	8,81%
Multiple Dockefile	Restructure files	5	0,92%
Remove data from container	Remove unnecessary data/packages/dependencies	62	11,38%
Remove unecessary dependencies/packages	Remove unnecessary data/packages/dependencies	20	3,67%
Remove unnecessary installation	Remove unnecessary data/packages/dependencies	11	2,02%
Remove unecessary scripts	Remove unnecessary data/packages/dependencies	4	0,73%
Clean data	Remove unnecessary data/packages/dependencies	1	0,18%
Strip not essential source files	Remove unnecessary data/packages/dependencies	2	0,37%
Remove packages with apt-get-purge	Remove unnecessary data/packages/dependencies Package managers	1	0,18%
Remove unused dependencies with apt-get-autoremove	Remove unnecessary data/packages/dependencies Package managers	3	0,55%
Clean up disk space	Remove unnecessary data/packages/dependencies Package managers	1	0,18%
Only install necessary dependencies	Remove unnecessary data/packages/dependencies Package managers	1	0,18%
Exclude installation recommendations	Remove unnecessary data/packages/dependencies Package managers	10	1,83%
Add dependencies	Tweaks	4	0,73%
Change dowload protocol	Tweaks	2	0,37%
Mount static data with VOLUME	Tweaks	1	0,18%
Only pull last data	Tweaks	1	0,18%
Speed up process with flag option	Tweaks	1	0,18%
Reduce version of toolsdependencies/packages	Tweaks	1	0,18%
Do not report dowload progress	Tweaks Package manager	1	0,18%
Install latest version	Tweaks Package manager	1	0,18%
Change package handler	Tweaks Package manager	1	0,18%
Do not perform automatic test	Tweaks Package manager	1	0,18%
Specific COPY	More specific	12	2,20%
Specific ADD	More specific	2	0,37%
Install esplicit version of tool	More specific	1	0,18%
Add ENV	Environment variables	1	0,18%
Modify ENV	Environment variables	3	0,55%
Combine multiple ENV	Environment variables	1	0,18%
Upgrade version of ENV	Environment variables	3	0,55%
Ignore data with .dockignore	Ignore files	16	2,94%

TAG	IMAGE SIZE	S	TIME BUILD	T
Smaller Image	57		23	
Other image	19		31	
Increase/Reduce base image version	5		5	
Variant of same base image	1		2	
Inline RUN	35		22	
Separate RUN	0		2	
Modify option in RUN	3		3	
Prevent cache	9		2	
Prevent cache with package manager apt	12		0	
Prevent cache with package managers yum	1		2	
Prevent cache with package manager apk	1		0	
Tools for caching	0		1	
Change ADD in COPY	6		5	
Sort COPY	1		46	
Sort ADD	4		23	
Sort installation sequence	4		23	
Sort ARG	0		2	
Multistage build	33		14	
Multiple Dockefile	1		4	
Remove data from container	45		17	
Remove unnecessary dependencies/packages	11		9	
Remove unnecessary installation	3		11	
Remove unecessary scripts	1		3	
Clean data	0		1	
Strip not essential source files	2		0	
Remove packages with apt-get-purge	1		0	
Remove unused dependencies with apt-get-autoremove	3		0	
Clean up disk space	1		0	
Only install necessary dependencies	0		1	
Exclude installation recommendations	7		3	
Add dependencies	1		3	
Change dowload protocol	0		2	
Mount static data with VOLUME	0		1	
Only pull last data	0		1	
Speed up process with flag option	0		1	
Reduce version of toolsdependencies/packages	1		0	
Do not report dowload progress	0		1	
Install latest version	0		1	
Change package handler	0		1	
Do not perform automatic test	0		1	
Specific COPY	0		12	
Specific ADD	0		2	
Install esplicit version of tool	0		1	
Add ENV	1		0	
Modify ENV	1		2	
Combine multiple ENV	0		1	
Upgrade version of ENV	1		2	
Ignore data with .dockeignore	7		9	

Nella pagina successiva, viene presentato uno schema generale che raggruppa tutte le categorie e le rispettive tassonomie in modo completo e organizzato. Man mano, nelle sezioni successive verrà analizzato ogni modifica più rilevante e maggiormente effettuata dagli sviluppatori.



5.1 Base Image



Commit : a612e44e5bb99cd9a036b2af79c0b3b30bf9a189 - **Star** : 3k

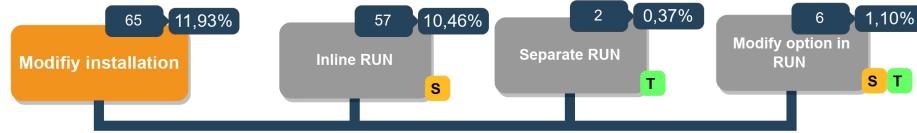
Commit msg : Using an alpine linux base for this image reduces the size of the image..

Goal : Cambiare l'immagine base al fine di diminuire le dimensioni finali dell'immagine

Significato : Scegliere l'immagine giusta con un footprint minimo del sistema operativo risulta un efficiente modifica da applicare ad un Dockerfile. Per rendere i Dockerfile leggere esistono alcune base image come Alpine, BusyBox, Scratch. Alpine, ad esempio, è una distribuzione leggera che usa solo pochi megabayte e possono essere piccole fino a 5.59 MB. Alcune immagini per impostazione predefinita, vengono fornite con la shell sh che aiuta a eseguire il debug del container rendendolo più grande di dimensioni. Per ridurre ulteriormente le dimensioni dell'immagine, si può optare per l'utilizzo di immagini "distroless" disponibili per diversi software. Queste immagini si distinguono per la loro straordinaria compattezza, al punto da non includere nemmeno una shell.

```
50 50 [diff] Dockerfile □
...
1  - FROM centos:7
1  + FROM alpine:3.3
2
3  - ENV TERRAFORM_VERSION 0.6.10
4  - ENV TERRAFORM_STATE_ROOT /state
3  + RUN apk add --no-cache build-base git openssh openssl py-pip python python-dev unzip \
4  +     && git clone https://github.com/CiscoCloud/mantl /mantl \
5  +     && apk add --no-cache build-base python-dev py-pip \
6  +     && pip install --upgrade pip \
7  +     && pip install -r /mantl/requirements.txt \
8  +     && apk del build-base python-dev py-pip
5 9
10 + VOLUME /local
11 + ENV MANTL_CONFIG_DIR /local
12 +
13 + VOLUME /root/.ssh
14 + ENV SSH_KEY /root/.ssh/id_rsa
15 +
16 + ENV TERRAFORM_VERSION 0.6.12
```

5.2 Modify installation



Commit : 8052f98627ba1304f320fa7c42afb51386427657 - **Star** : 15.7k

Commit msg : Refactor all build commands into 1 RUN command to try and reduce the number of cached build layers and thus reduce the size of the finished image.

Goal : Rifattorizzare il comando RUN al fine di ridurre il tempo di build e le dimensioni dell'immagine.

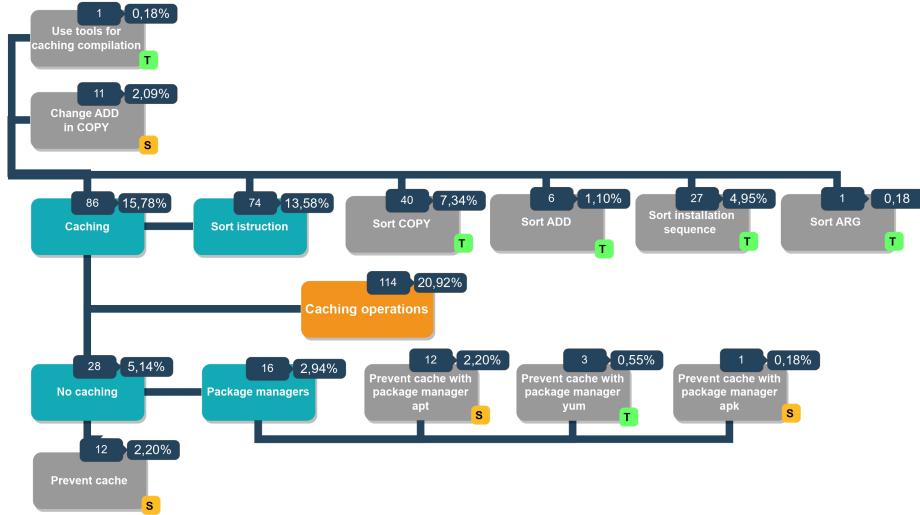
Significato : Come mostrato nelle precedenti sezioni, ogni istruzione RUN aggiunge un nuovo layer(strato), ed è facilmente intuibile che aumentare il numero di layers aumenterebbe la dimensione finale dell'immagine. Rifattorizzare, quando possibile, il comando RUN rendendolo in linea attraverso il costrutto **&&** è un approccio volto alla riduzione del numero di layers creati durante la costruzione dell'immagine. In questa commit, è possibile notare anche l'aggiunta del 'backslash', esso a differenza del **&&** ha lo scopo di rendere il Dockerfile leggibile, comprensibile e manutenibile.

```

12 12 + # Keep this all in one RUN command so that the resulting Docker image is smaller.
13 13 - RUN apk --no-cache add --virtual build-dependencies \
14 14 - build-base git go freetype-dev jpeg-dev ffmpeg-dev ragel libx11-dev libxt-dev libxext-dev
15 15 - RUN apk --no-cache add libgflags-dev@testing glog-dev@testing \
16 16 + build-base git go freetype-dev jpeg-dev ffmpeg-dev ragel libx11-dev libxt-dev libxext-dev \
17 17 + && apk --no-cache add libgflags-dev@testing glog-dev@testing \
18 18 + && mkdir -p build \
19 19     && cd build \
20 20     && make \
21 21     # Alpine's version of `install` doesn't support the `--mode=` format
22 22     && install -m 0755 hiptext /usr/local/bin \
23 23 -     && cd ../../ && rm -rf build
24 24 +     && cd ../../ && rm -rf build \
25 25     && COPY . /app
26 26     # Build the interfacer.go/xzoom code
27 27     && export GOPATH=/go && export GOBIN=/app/interfacer \
28 28     && cd /app/interfacer && go get && go build \
29 29
30 30
31 31
32 32
33 33
34 34
35 35
36 36

```

5.3 Cashing operations



Commit : dbe7d3972e02294f47ff07fc6933bf78ab59a856 - **Star** : 412

Commit msg : Adjust build order to reduce build times

Goal : Riordinare i comandi del Dockerfile al fine di ottenere un miglioramento delle prestazioni.

Significato : Per approfondire la comprensione intrinseca dello sfruttamento della cache, bisogna soffermarsi nuovamente e più nel dettaglio su una questione già incontrata molte volte. Si è detto che Docker, nel momento della costruzione delle immagini, ogni comando genera un nuovo layer. Questo vuol dire che quando si utilizza un'immagine base di node alpine come nell'esempio seguente, si hanno già i layers essendo questa già buildata usando il suo Dockerfile. Tutti i comandi successivi al FROM aggiungeranno nuovi layers a questa immagine.

```
FROM node:17.0.1-alpine
WORKDIR /app
COPY myapp/ app #copy project files into Docker image
RUN npm install --production #install project dependencies
CMD ["node", "src/index.js"]
```

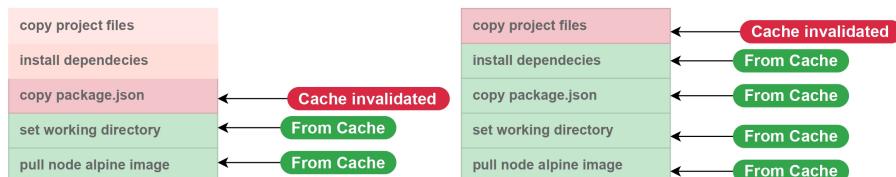
Per caching, si intende quindi, il processo di memorizzazione di un layer nella cache, salvato su un filesystem locale. Se nulla è cambiato nei layers, verrà nuovamente riusata la cache. Questo processo aiuta notevolmente il tempo di

costruzione dell'immagine. Il caching è altrettanto importante quando pulliamo un'immagine. Infatti, se pulliamo una nuova versione dell'immagine della stessa applicazione, dei nuovi layer verranno aggiunti, gli altri non necessitano il download e verranno riutilizzati dalla cache.

La categoria che è stata trovata durante lo studio (Sort istruction), raggruppa un concetto e delle operazioni essenziali per la scrittura di un Dockerfile. Costruendo il Dockerfile precedentemente mostrato notiamo come il comando COPY, viene giustamente ribuildato, per alcuni cambiamenti inseriti nell'applicazione. È possibile notare come anche il comando RUN viene ribuildato, questo perché quando un layer cambia, tutti quelli successivi sono ri-buildati. Una soluzione potrebbe essere ristrutturare il Dockerfile.

```
FROM node:17.0.1-alpine
WORKDIR /app
COPY package.json package-lock.json .
RUN npm install --production #install project dependencies
COPY myapp/ app
CMD ["node", "src/index.js"]
```

Ordinare i comandi Dockerfile da quello che viene modificato più raramente a quello che viene modificato più frequentemente è un'ottima pratica di miglioramento delle performance, in particolare il tempo di build.



L'idea è quella di non usare la cache sul comando RUN solo quando il package.json ha dei cambiamenti nelle sue dipendenze.

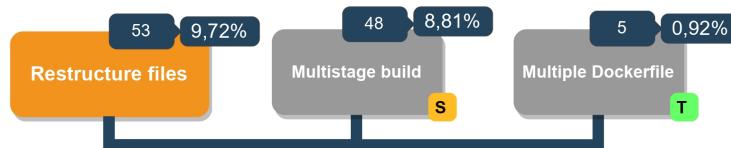
Quindi, spostando il comando COPY di tutti i file alla fine del Dockerfile. In generale, bisogna mantenere questa regola: prima i comandi che cambiano raramente. Dopo, i comandi che subiscono variazioni frequenti.

```

- COPY ./clouddriver-web/build/install/clouddriver /opt/clouddriver
-
5 RUN apk --no-cache add --update bash wget unzip 'python2>2.7.9' && \
6 wget -nv https://dl.google.com/dl/cloudsdk/release/google-cloud-sdk.zip && \
7 unzip -qq google-cloud-sdk.zip -d /opt && \
@@ -35,4 +33,7 @@ RUN adduser -D -S spinnaker
33 USER spinnaker
34
35 WORKDIR /home/spinnaker
36 +
37 + COPY ./clouddriver-web/build/install/clouddriver /opt/clouddriver
38 +
39 CMD ["/opt/clouddriver/bin/clouddriver"]

```

5.4 Restructure files



Commit : 67411257b35c42de7e2a9160a0b71fe35ecf8891 - **Star** : 417

Commit msg : use multistage builds to optimize the image size

Goal : Ristrutturare il Dockerfile, utilizzando la tecnica del multistage build al fine di ottenere un risultato migliore sulle dimensioni dell'immagine.

Significato : La costruzione multistage di Docker permette di creare immagini Docker in più fasi separate(stage), riducendo la dimensione finale dell'immagine. Durante la costruzione multistage, vengono utilizzate diverse fasi per eseguire operazioni specifiche, come compilazione e creazione di binari. Questo approccio permette di eliminare file temporanei e dipendenze non necessarie, riducendo l'overhead dell'immagine finale. In sintesi, la costruzione multistage di Docker ottimizza il processo di creazione delle immagini, migliorando le prestazioni e riducendo le dimensioni complessive dell'immagine. Inoltre, il multistage build facilita la gestione delle dipendenze e dei requisiti specifici per ciascuna fase, migliorando la modularità e la manutenibilità del Dockerfile. Di seguito viene fornito un esempio illustrativo di utilizzo di multisgtage build e di come esso provoca una ristrutturazione completa del Dockerfile. Bisogna chiarire che uno stage può essere buildato singolarmente: `docker build -target stage-name`

```

FROM maven: 3.6-jdk-alpine AS Builder
WORKDIR /app
COPY pom.xml .
RUN mvn -e -b dependecy:resolve
COPY src ./ src
RUN mvn -e -b package
CMD ["java","jar","app/app.jar"] #move to last stage

FROM openjdk: 8-jdr-alpine
COPY -from = builder /app/target/app.jar /
CMD ["java","jar","app/app.jar"]

```

1	- FROM bitnami/node:8
1	+ FROM bitnami/node:8 as builder
2	2
3	- LABEL maintainer "Bitnami Team <containers@bitnami.com>"
3	+ COPY . /app
4	+
5	+ WORKDIR /app
4	6
5	7 ENV NODE_ENV=production
8	+ RUN npm install yarn --global && \ 9 + yarn install && \ 10 + npm rebuild node-sass && \ 11 + yarn run build
6	12
7	- RUN npm install yarn --global
13	+ FROM bitnami/node:8-prod
8	14
9	- COPY . /app
15	+ LABEL maintainer "Bitnami Team <containers@bitnami.com>"
10	16
11	17 WORKDIR /app
12	18
13	- RUN yarn install && \ 14 - npm rebuild node-sass && \ 15 - yarn run build
19	+ COPY --from=builder /app /app
20	+
21	+ ENV NODE_ENV=production
22	+ RUN npm install yarn --global

5.5 Remove unnecessary data-packages-dependencies



Commit : a55d3b333625df58e07a0eed4e13b11af0dee95d - **Star** : 2.6k

Commit msg : Removed files used during build process to reduce image size

Goal : Rimuovere alcuni dati dalla build al fine di ridurre l'immagine finale

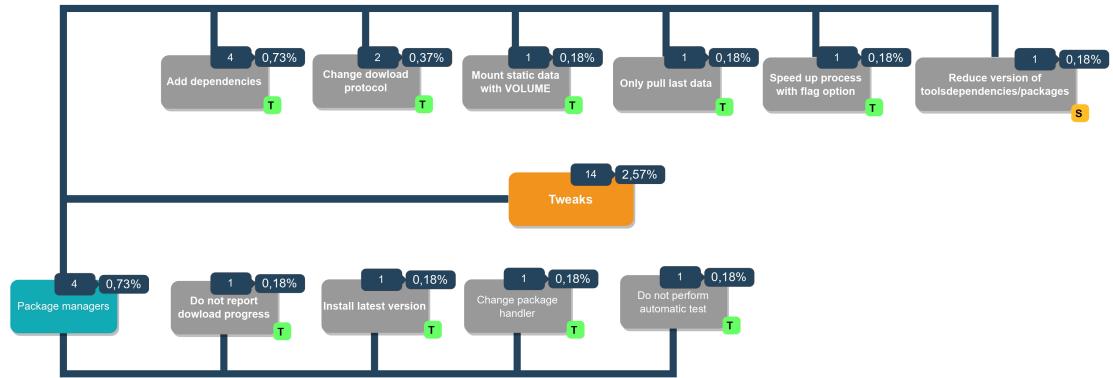
Significato : Il comando RUN rm può essere usato all'interno di un Dockerfile per rimuovere file o directory all'interno dell'immagine Docker. Tuttavia, l'uso di questo comando deve essere fatto con cautela, in quanto può comportare rischi per la sicurezza e l'efficienza del container. Può essere utile in diversi contesti, un esempio tra questi è la rimozione dei file temporanei. Se il Dockerfile crea file temporanei o di log che non sono necessari per l'esecuzione del container, rimuoverli con il comando RUN rm può ridurre la dimensione dell'immagine Docker finale.

```

1      1      FROM openjdk:8u181
2      2
3      - RUN apt-get update && apt-get install -y \
4      - make \
5      - python3-pip
3      + RUN apt-get update && apt-get install --no-install-recommends -y \
4      + make && \
5      + rm -rf /var/lib/apt/lists/*

```

5.6 Tweaks



Commit : 19de7f7b11629a36a3c72b4ec58353478ebd9a7f - **Star** : 8

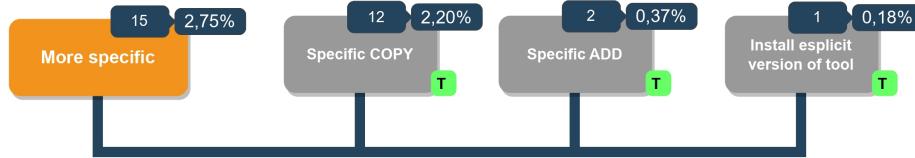
Commit msg : Adding more deps to here to reduce build time

Goal : Aggiungere alcune dipendenze nel Dockerfile al fine di migliorare il tempo di build.

Significato : Alcune dipendenze, come le librerie comuni, possono essere pre-compilate o avere file di cache che consentono di accelerare la compilazione dell'applicazione. Utilizzando queste dipendenze, l'ambiente di build può evitare di ripetere operazioni costose come la compilazione da zero, riducendo così il tempo di build complessivo. In generale però, sembra incoerente l'idea che aggiungere un qualcosa renda il processo più veloce. Ovviamente dipende dal contesto, dalle volontà e le esigenze dello sviluppatore.

↑	@@ -11,6 +11,7 @@ RUN apt-get install -y \
11	11 libboost-all-dev \
12	12 libfftw3-3 \
13	13 libfftw3-dev \
14	+ libgcrypt-dev \
14	15 libqwt-dev \
15	16 libqwt5-qt4 \
16	17 libusb-1.0-0 \
↓	@@ -19,19 +20,38 @@ RUN apt-get install -y \
19	20 libuhd-dev \
20	21 libuhd003 \
21	22 libcppunit-dev \
23	+ libsqlite3-dev \
24	+ mercurial \
22	25 uhd-host \
23	26 pkg-config \
27	+ python-cairo \
24	28 python-cairo-dev \
25	29 python-cheetah \
30	+ python-crypto \
31	+ python-dev \
26	32 python-gtk2 \
27	33 python-lxml \
28	34 python-mako \
35	+ python-matplotlib \
29	36 python-numpy \
37	+ python-serial \
30	38 python-qt4 \
31	39 python-qwt5-qt4 \
32	40 python-pip \
41	+ python-usb \
33	42 python-zmq \
34	- swig
43	+ swig \
44	+ sqlite3 \
45	+ tcpdump \
46	+ vim \

5.7 More specific



Commit : a24695206b1ff4a47375ef84e1a5bbcc0811202a - **Star** : 47

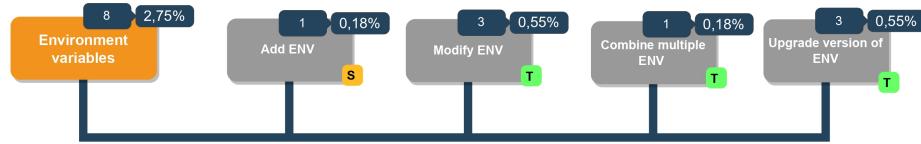
Commit msg : only copy requirements to reduce build time

Goal : Modificare il comando COPY al fine di ottenere miglioramenti dal punto di vista delle performance.

Significato : Modificare il comando COPY in modo più specifico può essere utile per selezionare esattamente i file necessari da copiare nel container, riducendo così la dimensione dell'immagine e il tempo di build. Invece di copiare l'intera directory, è possibile specificare solo i file o le cartelle rilevanti per l'applicazione, evitando così di includere file non necessari. Questa pratica consente di ottimizzare le risorse e garantire un'immagine Docker più efficiente e snella.

```
15      15      && rm -rf ta-lib*
16      16
17      17
18      - COPY . /app
18      + # copy only the requirements to prevent rebuild for any changes
19      + COPY requirements.txt /app/requirements.txt
19      20
20      21      # ensure numpy installed before ta-lib, matplotlib, etc
21      22      RUN pip install 'numpy==1.14.3'
```

5.8 Environment variables



Commit : 28d1252e12809812274ab6bdb1b5ef9dc7e9537 - **Star** : 132

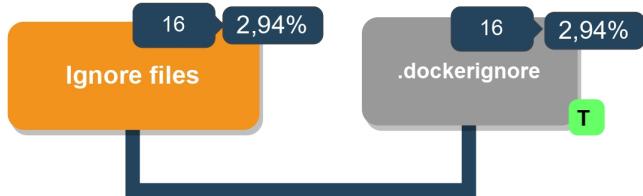
Commit msg : Some updates to the Dockerfile to reduce its size and the time needed to build it (in part, to speed up runs of tests).

Goal : Aumentare/Modificare la versione della variabile d'ambiente al fine di migliorare le performance.

Significato : Anche se l'analisi ha incluso tra le operazioni di refactoring anche l'utilizzo della ENV, sappiamo che generalmente l'utilizzo del comando ENV in un Dockerfile non influisce direttamente sulle performance dell'applicazione o dell'immagine Docker stessa. Il comando ENV viene utilizzato per impostare variabili d'ambiente all'interno del container, che possono essere utili per configurare e personalizzare il comportamento dell'applicazione al suo interno. Ad esempio, è possibile utilizzare le variabili d'ambiente per passare informazioni di configurazione o parametri all'applicazione in esecuzione nel container. L'utilizzo di variabili d'ambiente può rendere il Dockerfile più flessibile e facilmente configurabile, ma non ha un impatto diretto sulle prestazioni dell'applicazione o sulla velocità di build dell'immagine Docker. La rilevanza dell'utilizzo del comando ENV nel Dockerfile per le performance non dovrebbe influire, ma varia a seconda del contesto e dell'uso specifico.

58	- ENV PREINSTALLED_RUBY_VERSIONS 2.0.0-p648 2.1.8 2.2.3 2.2.4 2.3.0
58	+ ENV PREINSTALLED_RUBY_VERSIONS 2.1.8 2.2.4 2.3.0
59	+ ENV RUBY_CONFIGURE_OPTS --disable-install-doc

5.9 Ignore files



Commit : f6d1f76b25aac2950b26f332f33f6986cbd8e35d - **Star** : 2

Commit msg : Added Dockerignores to decrease docker-build times

Goal : Migliorare le performance riducendo il numero di elementi considerati nella fase di build.

Significato : Il file .dockeignore consente di specificare i file e le cartelle da escludere durante la fase di build dell'immagine Docker. In tal modo, i file indicati nel .dockeignore non vengono considerati né inclusi nell'immagine finale. Questo meccanismo contribuisce a ridurre il tempo e le risorse necessarie per la build, consentendo di ottimizzare le prestazioni complessive del Dockerfile. L'utilizzo del file .dockeignore consente di selezionare in modo più mirato i contenuti da copiare e includere nell'immagine, offrendo un approccio efficiente per migliorare l'efficacia delle operazioni di build nel contesto specifico di utilizzo.

```
▼ 2 ████ client/.dockerignore □
...
... @@ -0,0 +1,2 @@
1 + node_modules
2 + .git
```

6 Conclusioni e sviluppi futuri

In questa tesi si condotto un'indagine mirata all'analisi approfondita dei Dockerfile e come i sviluppatori li modificano al fine di migliorare le performance. Ogni Dockerfile è stato analizzato manualmente e studiato nei suoi cambiamenti al fine di trovare l'operazione cardine che ha mosso lo sviluppatore a generare un messaggio di commit riferito alle performance. Attraverso lo studio approfondito delle pratiche di refactoring adottate dagli sviluppatori, è emerso che il refactoring del Dockerfile può effettivamente fornire vantaggi significativi in termini di prestazioni, e che tante altre volte vengono messe in ballo operazioni che invece non dovrebbero essere presenti nel contesto delle performance. I risultati ottenuti hanno dimostrato che l'ottimizzazione del Dockerfile può contribuire a ridurre il tempo di build, la dimensione dell'immagine e l'utilizzo delle risorse. Ciò si traduce in una maggiore efficienza operativa e prestazioni ottimizzate per le applicazioni containerizzate. In definitiva, questa tesi dimostra che il refactoring del Dockerfile è un'attività preziosa e necessaria per gli sviluppatori che desiderano ottimizzare le performance delle proprie applicazioni containerizzate. Incorporando i pattern identificati in questa ricerca, gli sviluppatori possono creare Dockerfile efficienti, riducendo i tempi di build e garantendo prestazioni ottimali delle loro applicazioni nell'ambiente containerizzato.

Con lo scopo di ampliare il numero di pattern per il refactoring, potrebbe essere utile lavorare su un algoritmo più efficiente di analisi delle commit. Un problema di questo studio, infatti, è stato riuscire ad ottenere solo messaggi di commit utili, ma essendo loro mischiati per ovvie ragioni tra merge e falsi positivi, è stato un processo laborioso. Pertanto, sarebbe utile avere commit ottimali per l'analisi, scartando a priori quelle non necessarie. Ampliando sempre di più lo studio, si valuterebbe come più individui approcciando al refactoring per trarne così sempre più conoscenze valide allo scopo di raccogliere informazioni più approfondite, per elaborare una guida esaustiva per una corretta implementazione dei Dockerfile, tenendo conto delle performance.

Riferimenti bibliografici

- [1] Data courtesy of stack overflow - most loved tools.
- [2] Data courtesy of stack overflow - most popular technologies.
- [3] Natural.
- [4] Natural language processing with node.js.
- [5] Snowball.
- [6] Zhuo Huang, Song Wu, Song Jiang, and Hai Jin. Fastbuild: Accelerating docker image building for efficient development and deployment of container. In 2019 35th Symposium on Mass Storage Systems and Technologies (MSST), pages 28–37, 2019.
- [7] Emna Ksontini, Marouane Kessentini, Thiago do N. Ferreira, and Foyzul Hassan. Refactorings and technical debt in docker projects: An empirical study. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 781–791, 2021.
- [8] GIOVANNI ROSA, SIMONE SCALABRINO, GABRIELE BAVOTA, and ROCCO OLIVETO. What quality aspects influence the adoption of docker images? 2018.
- [9] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. Characterizing the occurrence of dockerfile smells in open-source software: An empirical study. IEEE Access, 8:34127–34139, 2020.

7 Ringraziamenti

Desidero rivolgere un sentito ringraziamento alla mia famiglia per il costante sostegno e l'affetto che mi hanno donato lungo tutto il percorso di laurea. A mamma, papà e mia sorella Chiara, vi ringrazio per essere stati sempre al mio fianco, per avermi incoraggiato, per avermi ascoltato nelle mie continue lamentazioni e sostenuto in ogni passo tortuoso del mio percorso. La vostra presenza costante, il vostro amore incondizionato e la vostra fiducia è stata spesso la mia forza.

Ai nonni, zii, zie e cugini, voglio ringraziarvi per il vostro affetto e per il sostegno che mi avete dimostrato lungo tutto il percorso della mia formazione. Le vostre parole, i consigli e il vostro costante incoraggiamento mi hanno ispirato a dare sempre il massimo e ricordare il motivo per la quale lo stessi facendo.

Vorrei estendere il mio ringraziamento a tutti gli amici e i colleghi che mi hanno sostenuto nel percorso accademico. Le vostre parole spronanti, le discussioni stimolanti, l'amicizia e il sostegno ricevuto sono stati un regalo prezioso che mi ha reso più forte e determinato. Ritengo necessario sottolineare l'indiscutibile valore dei momenti di gioia che abbiamo condiviso nel corso di questi anni. Sono quei fugaci istanti di spensieratezza vissuti insieme a voi che hanno donato leggerezza al cammino percorso. Ritrovarsi in quei brevi attimi di divertimento, in grado di donare un sorriso e sollevare il peso dei giorni difficili, è stato un dono prezioso.

Desidero, ancora, rivolgere la mia gratitudine a tutte quelle persone speciali, sono tutte quelle persone che sanno di esserlo per me, senza esitare un secondo. Vi ringrazio per tutto ciò che siete stati e per aver reso la mia vita arricchita semplicemente facendo parte di essa.

Vorrei ringraziare, infine, il relatore Simone Scalabrino e il correlatore Giovanni Rosa per la loro preziosa guida e supporto costante durante la creazione della mia tesi.

Questo messaggio è stato generato da un'intelligenza artificiale addestrata.