Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Docker: analisi sull'uso e sulla diffusione

Tesi di Laurea in Informatica

Relatore:
Prof. Paolo Ciancarini
Correlatore:
Dott. Francesco Poggi

Presentata da:
Ciro,William,Giovanni Popolo

MAR, 2018

Indice

Elenco delle figure

Capitolo 1

Introduzione

Questa tesi studia i docker, una delle principali e più recenti tecnologie di virtualizzazione per automatizzare il deployment di applicazioni all'interno di container software. L'obiettivo di questo lavoro è analizzare la diffusione e caratterizzare l'uso dei docker attraverso l'analisi dei progetti della piattaforma github. Con la parola docker si intende un container software generato dalla piattaforma docker. Docker è una piattaforma che sfrutta una virtualizzazione del livello sistema operativo, per rispondere in modo efficiente, usando un ambiente leggero ed isolato, a problemi quali dipendenze software, versioning e altri problemi legati al software deployment. Un container e invece un ambiente con proprie dipendenze, librerie e file di configurazione; quindi un container è un pacchetto che consiste in un'applicazione, la quale viene eseguita dal sistema operativo che crea un ambiente isolato rispetto a tutti gli altri processi e vi installa librerie e configurazioni indicate.
Il lavoro che ho svolto si è concentrato prima sul capire cos'è docker e sul concetto di container, successivamente sono andato ad analizzare l'uso e la diffusione della tecnologia. Per poter capire docker e il concetto di container in primo luogo mi sono concentrato sullo studio della virtualizzazione, in particolare sui motivi che hanno portato alla nascita ed alla rapida diffusione di tale tecnologia, alle varie tecniche disponibili per attuare una virtualizzazione come la virtualizzazione hardware o la virtualizzazione ‘ system-level

', e a un confronto tra le tecniche evidenziandone le differenze.Il passo successivo è stato analizzare le tecnologie e le caratteristiche della piattaforma docker. Dopo aver studiato la storia della compagnia Docker Inc. che ha sviluppato la piattaforma, ho analizzato le principali motivazioni che hanno portato allo sviluppo di questa nuova tecnologia (ad esempio l'inferno delle dipendenze, la documentazione imprecisa, il codice obsoleto o i problemi di portabilità) e gli elementi principali dell'architettura docker (docker-engine, filesystem e ambiente usato, dockerfile e docker-compose). Successivamente, ho confrontato l'approccio proposto basato su docker con quelli delle principali tecnologie di virtualizzazione esistenti. Una volta fatto questo quadro completo sulla piattaforma mi sono concentrato sull'analisi dei progetti che usano la tecnologia docker caricati sulla piattaforma github. A questo scopo ho sviluppato un'applicazione per filtrare fra i 79 milioni di progetti quelli che fanno uso di docker, e fare mining delle principali informazioni ad essi relative. Github per non sovraccaricare i propri server di richieste da parte di utenti limita a questi ultimi l'uso delle api, in termini di tempo con il rate-limit e di risultati con la paginazione. Oltre ad uno studio della documentazione di github ho effettuato uno studio sulla piattaforma, usandola e sfogliando alcuni repository interessanti per la mia analisi. Per popolare il dataset da andare ad analizzare ho progettato un algoritmo che interrogasse il sistema di github tramite l'uso delle api REST, rispettando le limitazioni da esso imposte. In particolare l'algoritmo non si limita ad analizzare tutti gli elementi della lista JSON ma anche tutte le pagine html ad essi riferite. Il programma da me progettato una volta analizzati tutti gli elementi della lista appena citata e le relative pagine html elabora i dati raccolti e stila una serie di statistiche per poi andare a stamparle in un foglio di calcolo. Successiva alla realizzazione dello script è stata la sua esecuzione e il relativo studio dei risultati. In totale per l'analisi sono stati analizzati circa 200000 repository e solo per l'esecuzione del programma ci sono voluti ben 5 giorni. Tra le statistiche stilate ci sono: numero repository, media stelle mensile, media partecipanti per repository, numero di repository creati per ogni lin-

guaggio, media commit mensile e media commit per numero di repository. Con il seguente programma sarà inoltre possibile personalizzare la propria ricerca infatti è implementato per accettare vari argomenti tra cui la parola chiave, che nel caso della mia analisi sarà settata a docker, oltre alla parola chiave è possibile inserire l'anno in cui far iniziare la ricerca e l'anno in cui far terminare la ricerca, che nella mia analisi saranno 2013 e 2017. Sarà inoltre possibile inserire il percorso riguardante la directory nella quale lavorare.

1.1 Analisi dei risultati

Una volta terminata l'esecuzione ho potuto visionare i risultati riguardanti le statistiche stilate e ho potuto organizzarli in grafici, per andare a sottolineare alcuni aspetti che ho trovato interessanti, soprattutto per quanto riguarda i linguaggi utilizzati, le stelle e i commit. Alcune statistiche sono risultate alla fine poco interessanti, come la media dei partecipanti ad un repository, quindi ho deciso di oscurarle nei grafici ma comunque faranno parte delle statistiche raccolte nel foglio di calcolo. Per concludere l'analisi mi sono focalizzato su alcuni repository che usano docker per capire a che scopo lo fanno e come docker rende il raggiungimento di tale scopo migliore rispetto ad un' altra tecnologia. Ho quindi visionato e studiato circa 50 repository, manualmente, per cercare di carpire gli usi che vengono fatti di docker da parte degli utenti della community. Dopo questo faticoso lavoro sono riuscito ad individuare degli usi tipici che vengono fatti di docker e sono così potuto andare a categorizzare tali usi come: soluzione alle dipendenze, ambiente di test, gestore container, composizione container, estensione docker. Inoltre ho anche individuato il fatto che molti progetti che fanno uso di docker ricadano in varie categorie delle appena citate e che alcune categorie consistano nello stesso uso di docker da parte dell'utente ma con un fine diverso, sarà il caso di soluzione alle dipendenze e ambiente di test. Concludo la tesi con una discussione finale sull' analisi e sul lavoro svolto. In particolare la tesi è articolata come segue:

- **Capitolo 2:** introduce la virtualizzazione, descrive le tecniche per attuarla e le confronta.
- **Capitolo 3:** presenta la storia riguardante la nascita e lo sviluppo della virtualizzazione e della piattaforma docker, descrive le motivazioni che portano alla nascita della piattaforma, le principali caratteristiche e i principali elementi dell'architettura della piattaforma.
- **Capitolo 4:** introduce gli strumenti adoperati per l'analisi tra cui github, le api che mette a disposizione e le limitazioni su esse, lo scraping e il linguaggio xpath, descrive la progettazione del programma usato per svolgere l'analisi, dall'implementazione all'esecuzione alla personalizzazione.
- **Capitolo 5:** descrive l'analisi effettuata, i risultati raccolti sotto forma di grafici, una categorizzazione dell'uso di docker.
- **Conclusioni:** descrive una discussione sull'analisi effettuata e sulle più interessanti osservazioni emerse.

Capitolo 2

Ambienti di virtualizzazione

Nel seguente capitolo sarà introdotto il concetto di virtualizzazione in concomitanza con l'introduzione dei problemi che hanno portato alla necessità di una qualche tecnica di virtualizzazione. Oltre al concetto di virtualizzazione sarà introdotto quello di hypervisor, componente fondamentale quando si parla di virtualizzazione e soprattutto di tecniche di virtualizzazione, e dei due tipi di hypervisor diffusi ai giorni nostri. Successivamente saranno introdotte e confrontate le principali tecniche per attuare una virtualizzazione e saranno sottolineati vantaggi e svantaggi a cui una diversa tecnica di virtualizzazione può condurre.

2.1 Ambiti della virtualizzazione

Per avere una panoramica generale sulla virtualizzazione vado ad introdurre alcuni dei principali motivi che portano al bisogno di questa tecnica, tra questi ci sono: affidabilità del sistema operativo e dei servizi, alti costi di manutenzione hardware, impossibilità di eseguire applicazioni legacy, mancanza di sicurezza dati in caso di guasti, difficoltà nello scalare le risorse. Ognuno di questi problemi viene risolto o limitato dalla virtualizzazione in modo efficacie, tanto che al giorno d'oggi è una delle pratiche più usate in ambito

test o server. Per quanto riguarda i problemi citati prima la virtualizzazione offre delle soluzioni semplici ed efficaci :

- **Affidabilità:** l'uso della virtualizzazione limita l'inaffidabilità del sistema operativo e/o dei servizi infatti è possibile configurare una macchina virtuale affinché esegua solo pochi servizi, che non vadano in conflitto tra loro, ed in caso di problemi sarà compromesso solo l'uso di quella macchina virtuale e non di altri sistemi(altre macchine virtuali, sistema host).

- **Manutenibilità:** i costi legati alla manutenzione dell'hardware vengono ridotti tramite l'uso della virtualizzazione, in quanto è possibile eseguire più sistemi operativi contemporaneamente su una sola macchina fisica, ciò significa che il bisogno di nuovo hardware si riduce, riducendo notevolmente i costi. Oltre ad una riduzione dei costi relativi a nuovo hardware la virtualizzazione garantisce una riduzione dei costi derivanti dall'hardware stesso, quali: energia elettrica e spazio e manutenzione.

- **Compatibilità:** la virtualizzazione rende possibile l'esecuzione di applicazioni legacy. Molte applicazioni funzionano solo su hardware obsoleto, per permettere il loro funzionamento su un hardware recente o semplicemente non supportato c'è bisogno di una migrazione della stessa applicazione verso un'architettura attuale, con relativi costi di porting e debug. Con l'uso della virtualizzazione sarà possibile riprodurre qualsiasi hardware, eliminando il problema dell'impossibilità di eseguire un'applicazione non supportata da una certa architettura e inoltre eliminando i costi di porting e debug.

- **Sicurezza:** l'uso della virtualizzazione garantisce maggiore sicurezza dei dati in caso di guasti. Eseguire il backup di una macchina virtuale, a differenza dell'eseguirlo di una macchina fisica, è una compito che richiede poco impegno sia per quanto riguarda il tempo che le risorse impiegate. Anche ripristinare una macchina virtuale, a differenza

del ripristinare una macchina fisica , richiederà un impegno basso,un disaster recovery può essere immediato.

- **Scalabilità e ottimizzazione risorse:** la difficoltà nello scalare le risorse è notevolmente limitata dall'uso della virtualizzazione. Aggiungere o rimuovere delle risorse ad una macchina fisica può essere difficoltoso, ciò comporta il dover mettere mano all'hardware della macchina in modo opportuno. Con l'uso di una macchina virtuale questa difficoltà è eliminata in quanto sarà possibile aggiungere o rimuovere risorse con un semplice click, ovviamente limitatamente alle risorse della macchina fisica che ospita la macchina virtuale in questione.

2.2 Tecniche per la virtualizzazione

La virtualizzazione è un meccanismo che permette di eseguire più sistemi operativi su una sola macchina contemporaneamente. È una tecnica che nasce nei lontani anni '60 in IBM. Questa tecnica nell' ultimo ventennio ha avuto una forte ascesa e praticamente tutte le più grandi compagnie nel campo informatico hanno investito ingenti capitali nello sviluppo della tecnologia. La virtualizzazione rappresenta un concetto chiave quando si parla di Docker, ci sono diverse tecniche per la virtualizzazione ed essa è usata nei più disparati ambiti oramai, dati i notevoli vantaggi a cui essa può portare. Prima di stabilire quali siano le diverse tecniche per attuare virtualizzazione bisogna introdurre il concetto di **Virtual Machine Monitor (VMM) o Hipervisor:** può essere un software, un firmware o un hardware capace di comunicare sia con il sistema operativo che con l'hardware di una macchina fisica. Gestisce l'allocazione delle risorse, come memoria o larghezza di banda , tra la macchina fisica e le macchine virtuali permettendo così la coesione di più sistemi operativi contemporaneamente su una sola macchina fisica. Il concetto di hypervisor nasce tra la fine degli anni '60 e l'inizio degli anni '70 in IBM ma la maggior diffusione si ha intorno agli anni 2000 in

concomitanza con la diffusione della virtualizzazione e delle varie tecniche di virtualizzazione. Oggi si distinguono due tipi di hypervisor:

- **Type 1 Hypervisor:** lavora direttamente sul livello hardware della macchina fisica e gestisce i sistemi operativi dei livelli più alti come mostrato in figura 2.1. Risiedendo sul livello hardware questo tipo di hypervisor è completamente indipendente dal sistema operativo. L'hypervisor ha dimensioni molto ridotte e il suo compito principale è quello di gestire la condivisione delle risorse hardware tra i sistemi operativi ospiti. Il vantaggio principale di questo tipo di hypervisor è che se si dovesse avere un guasto ad una macchina virtuale o a uno dei sistemi operativi guest, tale guasto non viene propagato alle altre macchine virtuali né al sistema operativo host. Microsoft Hyper-V, VMWare ESXi Server, Citrix/Xen Sever sono tra le piattaforme che usano hypervisor type 1 le più diffuse.

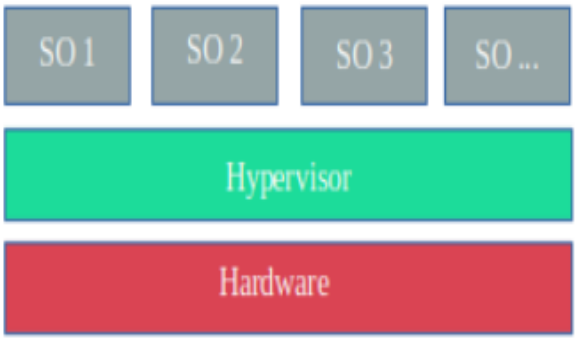

Figura 2.1: Esempio di Hypervisor Type 1

- **Type 2 Hypervisor:** questo tipo di hypervisor a differenza del precedente lavora un livello sopra il sistema operativo host e supporta la presenza di altri sistemi operativi a livelli superiori. Il fatto che l'hypervisor risieda sul sistema operativo host, come mostrato in figura 2.2,

rende l'hypervisor completamente dipendente da tale sistema. Se da un lato adoperando questo tipo di hypervisor si ha un controllo maggiore, risiedente nel sistema operativo host, dall'altro lato ciò comporta che un guasto nel sistema operativo host si ripercuote nei sistemi operativi guest. Tra le piattaforme più diffuse che usano hypervisor type 2 ci sono VMWare Workstation,Microsoft Virtual PC, Oracle Virtual Box.

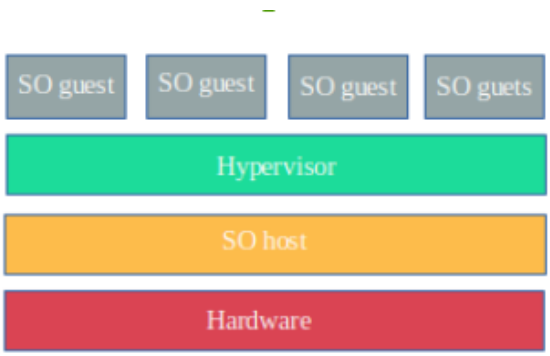

Figura 2.2: Esempio di Hypervisor Type 2

Era doveroso introdurre il concetto di hypervisor prima di parlare delle tecniche di virtualizzazione, dato che le differenze tra le tecniche saranno dovute anche alla presenza di questo componente. Le tecniche di virtualizzazione più diffuse oggi sono la virtualizzazione del livello hardware e la virtualizzazione del livello sistema operativo,con differenze sostanziali che mostro di seguito:

- **Virtualizzazione Hardware:** consiste nell'emulare un certo hardware ed installarvici un sistema operativo , quindi costruire una macchina virtuale, viene mostrato in figura2.3. I due sistemi operativi riescono a coesistere contemporaneamente grazie al hypervisor, che può essere sia di tipo 1 che di tipo 2, il quale si occupa di gestire l'allocazione delle risorse ai vari sistemi operativi, isolare le diverse macchine

virtuali e garantire la stabilità dei sistemi operativi, il tutto inserendo un overhead. La virtualizzazione consente tra le altre cose di ridurre il numero di macchine fisiche in funzione abbattendo i costi di energia elettrica e di hardware. In genere bisogna attivare la possibilità di avere una macchina virtuale nel BIOS della macchina fisica. La tecnica di virtualizzazione del hardware è la più diffusa ma ci sono dei punti deboli. Tra i software più diffusi che usano questa tecnica di virtualizzazione troviamo Virtual Box(Oracle), Android Studio(Google), Vmware (Dell), Virtual PC (Microsoft) tutte con le loro precise caratteristiche ma in generale per lo stesso scopo.

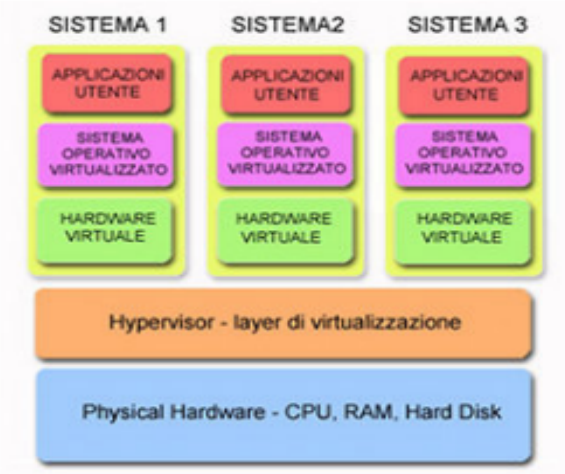

Figura 2.3: Virtualizzazione Hardware

- **Virtualizzazione 'System-Level':** sfrutta il concetto di container, cioè un contenitore indipendente dal sistema operativo con proprie librerie e configurazioni. Il sistema operativo ospitante crea uno spazio isolato dal resto dei processi e ci esegue il container. Questa tecnica, detta anche "containerizzazione", come la precedente garantisce un risparmio in termini di costi e un uso più efficiente delle risorse ma

stavolta senza l'intervento di un VMM. Questo tipo di virtualizzazione è molto usata in ambito di hosting virtuale dove è utile per allocare risorse finite tra un elevato numero di utenti. Un altro tipico uso che si fa di questa tecnica e da parte degli amministratori di sistema i quali possono decidere di incapsulare i vari servizi di un server in vari contenitori diversi su quel server, con il fine di rafforzare l'hardware del server. Le differenze con la virtualizzazione hardware sono molteplici, la più significativa è che mentre nella prima ogni sistema ha il suo kernel e l'hypervisor gestisce le risorse, nella seconda il kernel è uno solo ed è quello del sistema host apportando vantaggi significativi. In figura 2.4 mostro un esempio:

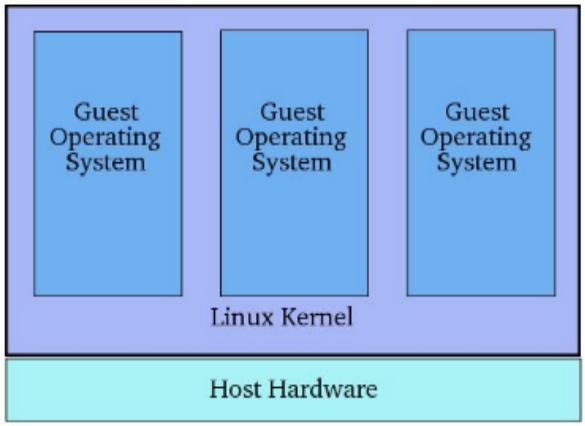

Figura 2.4: Virtualizzazione System-Level

2.3 Tecniche a confronto

Entrambe le tecniche, in base all'uso che se ne deve fare, possono avere i propri vantaggi; installare un sistema operativo può essere un'operazione lunga e faticosa,inoltre ogni livello dell'architettura introduce un overhead.

Quindi se da un lato abbiamo due (o più) sistemi operativi e i sistemi virtuali meno efficienti del sistema host, la possibilità di creare più macchine virtuali ,limitatamente alle risorse fisiche disponibili. Dall' altro lato abbiamo un sistema operativo incapsulato in un processo del nostro sistema host, con il quale condivide il kernel, cancellando overhead introdotto dai livelli hipervisor e OS host e eliminando difficoltà varie di installazione e gestione di un sistema operativo. Un container racchiude in sè solo librerie richieste dalla funzione che esso stesso deve svolgere, quindi le sue dimensioni sono molto ridotte rispetto ad una macchina virtuale. Le dimensioni ridotte di un container consentono l'esecuzione di un numero di container maggiore rispetto al numero di macchine virtuali eseguibili contemporaneamente a parità di hardware fisico. In figura 2.5 mostro un confronto fra le due tecniche:

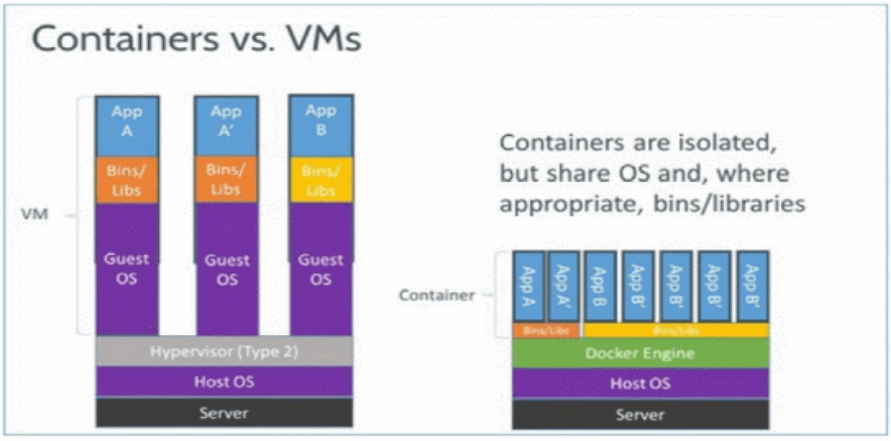

Figura 2.5: Virtualizzazione Hardware vs System-Level

Docker quindi è una piattaforma che utilizza una tecnologia di virtualizzazione system level e la rende 'user-friendly' [1]. Un concetto fondamentale quando si parla di docker, o di virtualizzazione in generale, è l'immagine. Un'immagine è un file che contiene l'istantanea di un contenitore, o di un intero sistema operativo. Sia un contenitore che una macchina virtuale sono

l'istanza di un' immagine, le azioni più diffuse per quanto riguarda un'immagine sono salvataggio e condivisione. Nel capitolo successivo sarà evidenziata prima la diffusione delle tecniche di virtualizzazione e poi quella di docker, successivamente sarà introdotto il contesto nel quale un container docker entra in funzione, le principali componenti della sua architettura e alcune caratteristiche. Per terminare sarà fatto un accenno sul funzionamento e sull'uso di un container docker.

Capitolo 3

Docker: storia e caratteristiche principali

Il seguente capitolo espone una panoramica generale sui container docker, partendo dalla storia della compagnia Docker Inc., non prima di aver mostrato la fulminea ascesa della virtualizzazione come tecnica general purpose. Verranno successivamente analizzati alcuni dei principali problemi legati al software deployment e i modi di affrontarli usando docker. Successivamente sarà introdotta l'architettura e le principali componenti che popolano la piattaforma. Nelle sezioni successive si osserverà il funzionamento della piattaforma, alcuni esempi di comandi per usare i container docker e per concludere uno sguardo alla modalità swarm e alla possibilità di comporre container per realizzare applicazioni basate su micro-servizi. Bisogna infatti fare una distinzione che sarà fondamentale nel parlare di container software, tra applicazioni monolitiche e applicazioni basate su microservizi.

- **Applicazioni monolitiche:** hanno la principale caratteristica di avere tutte le funzioni in un unico componente. Questa caratteristica tende a generare componenti sempre più complessi e difficili da gestire con il passare del tempo. Il problema principale di questo approccio è che, per applicazioni di grandi dimensioni, diventa molto difficile capire dove una determinata funzione è implementata. Questo problema porta

spesso ad introdurre ridondanze. Un altro problema si riscontra in caso di guasti, infatti si perderanno tutte le funzionalità della applicazione qualora una sola di esse subisca un guasto.

- **Applicazioni basate su microservizi:** la principale caratteristica che le differenzia dalle applicazioni monolitiche è che in questo tipo di applicazioni sono presenti più componenti. Ognuno di questi componenti sviluppa una funzionalità dell'applicazione ed è indipendente dalle altre componenti. Le componenti posso comunicare tra loro tramite il protocollo REST per scambiarsi informazioni utili allo svolgimento di una funzionalità. I vantaggi di un approccio del genere sono molteplici : scalabilità dell'applicazione, resistenza in caso di guasti ad un componente. Bisogna anche pensare che la gestione della comunicazione tra le componenti dell'applicazione può non essere semplice. Per citare alcune compagnie, Netflix e Spotify adottano un approccio basato su microservizi.

La nascita di Docker come piattaforma per l'uso di container è una diretta conseguenza della vastissima diffusione che la virtualizzazione ha avuto nel corso degli ultimi anni. Come descritto nelle sezioni successive l'apice di attenzione sulla nuova tecnologia si raggiunge intorno agli anni 2000.

3.1 Virtualizzazione: dal giorno zero ad oggi

Il giorno zero per la virtualizzazione è rappresentato dall' uscita dell'IBM System/370 nel Gennaio del 1970, questo sistema fu il primo che offriva a diversi utenti la possibilità di avere una copia personale del sistema in esecuzione su un' unica macchina fisica(mainframe all'epoca). Questa innovativa funzione era permessa grazie alle componenti Virtual Machine(VM) e Conversational Monitor System(CMS) . Oltre a questo evento fanno parte della "preistoria" della virtualizzazione altri due avvenimenti, lo sviluppo del primo Virtual Machine Monitor (VMM) chiamato "Simultask" da parte della

Locus Computing Corp. (1985) e lo sviluppo del primo emulatore di sistema MS-DOS su sistema Unix da parte di Insigna Solutions con il nome di Soft-PC(1988). Intorno agli anni 2000 si concentrano la maggior parte degli eventi che hanno segnato la storia e l'ascesa della virtualizzazione,in successione, nasce VMware(1998) oggi una delle compagnie leader in campo di virtualizzazione,viene pubblicata Virtual PC (2001)per windows da parte di Connectix,successivamente acquisita da Microsoft(2003),Intel e AMD aggiungono il supporto alla virtualizzazione nei loro processori(2005) , le piattaforme vengono rilasciate gratuitamente e hypervisor presente su tutte le macchine per lanciare macchina virtuale da desktop(2008) , sviluppo di Android Studio da parte di Google e Docker da parte di Docker Inc. (2013) .Al giorno d'oggi la virtualizzazione è una tecnica usata per risolvere molti problemi, è stata proprio questa sua caratteristica general purpose a rendere la sua diffusione così vasta. Questa tecnica è utilizzata per diversi scopi , c'è chi la usa per rendere 'portabili' le proprie applicazioni come Java Virtual Machine(JVM), che può essere installata su qualsiasi sistema operativo moderno e permette l'esecuzione di applicazioni scritte in Java,o chi rende questa tecnica di virtualizzazione user-friendly , cioè permette a chiunque di creare la propria macchina virtuale, con il sistema operativo che si è scelto,bisognerà però possedere l'immagine del sistema operativo. Di quest' ultima categoria abbiamo notevoli esempi ma le piattaforme più diffuse sono senza dubbio Vmware (piattaforma leader nel settore indicata anche per tramutare una macchina fisica in una macchina virtuale) , VirtualBox (piattaforma compatibile con ogni sistema operativo e con configurazioni Vmware) .

3.1.1 Docker Inc. : la storia

Docker nasce nel 2013 da un progetto open-source della compagnia dotCloud,impegnata nel campo del "platform-as-a-service", il linguaggio scelto per il progetto fu GO (sviluppato da Google con sintassi simile a C) . Inizialmente la compagnia si concentrò sulle tecnologie cloud ma solo per pochi mesi. La svolta la diede l'avvento di Solomon Hykes,il nuovo CEO , che in

poco tempo rivoluzionò la compagnia modificando il nome della stessa in Docker Inc. e spostando l'interesse dell'intera compagnia sullo sviluppo di questa nuova tecnologia[2]. Nel giro di pochi mesi la tecnologia suscita un interesse tale da portare la compagnia, prima a un importante accordo con Red Hat nel settembre del 2013 , e poi alla collaborazione con Microsoft nell'ottobre del 2014.Parallelamente nasce anche il progetto open-source Kubernetes di Google , ormai tutto il mondo aveva messo gli occhi su Docker. Nascono presto i primi attriti tra compagnia e vendor,la volontà di rendere i container docker affiancabili ad altre soluzioni tipo Kubernetes(Red Hat) o Amazon EC2 da parte dei vendor porta la società Docker a proseguire in maniera autonoma allo sviluppo della tecnologia. Il punto di non ritorno si raggiunge nell'estate del 2016 [3] quando Docker annuncia l'implementazione di swarm, la propria soluzione di orchestrazione di container; a questo punto Red Hat presenta OCID(Open Container Initiative Daemon) considerato da molti un fork parziale di docker, la scissione tra le società non si concretizza ma di fatto avviene. Un cambiamento rilevante si ha quando Docker decide di rendere pubblico il codice sorgente di alcune componenti alla base della tecnologia, mediante donazione alla Cloud Native Computing Foundation(progetto collaborativo della Linux Foundation lanciato nel 2015) nel marzo del 2017, questa mossa garantirà alla piattaforma la compatibilità con soluzioni di terze parti. Questo cambiamento porterà ad affievolirsi l'attrito con i vendor anche se la divisione interna continuerà ad esistere.

3.2 Caratteristiche dei container software

Per capire lo scopo di docker bisogna dare uno sguardo ad alcuni dei principali problemi legati al software deployment e alla soluzione adottata da docker per risolverli o aggirarli , di seguito prenderò in considerazione alcuni problemi noti e ne mostrerò la soluzione pensata da docker per il problema stesso.

- **Inferno delle dipendenze:** ho perso il conto ormai delle volte in cui

ho provato ad installare un software e non ci sono riuscito per un problema di dipendenze,un problema molto comune e allo stesso tempo senza una soluzione standard. Il problema è che installare un'applicazione richiede,spesso, l'installazione di software aggiuntivo per funzionare;ogni applicazione richiede la creazione di un ambiente specifico per eseguire i suoi compiti e spesso non è ben chiaro come questo ambiente vada definito. **Rimedio:** non c'è nulla di più facile che usare un container nel quale tutto il software che ci serve è già stato installato,configurato e testato.

- **Documentazione imprecisa :** la documentazione riguardante le dipendenze,l'installazione e l'esecuzione del codice è spesso imprecisa , sbagliata o addirittura mancante, rendendo il codice inutilizzabile. **Rimedio:** il file di configurazione di un container docker detto DockerFile è uno script nel quale si definisce esattamente come costruire l'ambiente,passo per passo.

- **Codice obsoleto:** le dipendenze citate sopra sono dei pacchetti e come tali subiscono aggiornamenti ,aggiunta di nuove funzionalità o la scomparsa di altre ed ognuna di queste può, potenzialmente ,corrompere la nostra esecuzione[4]. **Rimedio:** anche in questo caso il dockerfile risolve il problema.

- **Portabilità:** eseguire il mio codice su un'altra macchina spesso comporta lo scontro con uno dei problemi citati sopra. **Rimedio:** il DockerFile è un file di testo di dimensioni molto ridotte, quindi se il sistema operativo ospitante supporta docker possiamo 'portare' il DockerFile su qualsiasi macchina trasportando l'intero ambiente computazionale in un file di dimensioni molto ridotte.

3.3 Elementi principali dell'architettura

Ci sono diversi aspetti interessanti da considerare quando si parla di docker ,le sue componenti giocano un ruolo fondamentale in ogni singola funzionalità di docker dall' isolamento alla documentazione passando per il versioning o per la composizione di container. Di seguito faccio un quadro sugli elementi principali che compongono l'architettura della piattaforma. In figura 3.1 è mostrata l'architettura della piattaforma docker, nelle sezioni successive analizzo i componenti principali di tale architettura.

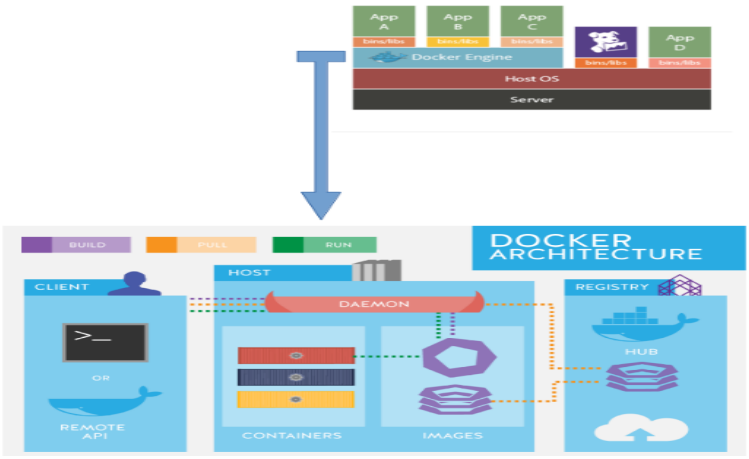

Figura 3.1: Architettura Docker

3.3.1 Docker Engine

Il cuore di Docker è docker engine, un 'architettura che comunica con il kernel Linux e accetta comandi da un client,sia da riga di comando che tramite API. Docker engine è basato su un'architettura client-server :

- **server:** è un host sul quale risiede un demone("dockerd") che si occupata della creazione e della gestione di uno o più contenitori docker, e di comunicare con il client e il sistema operativo host.

- **client:** prende comandi da utente(linea di comando) e comunica con il server.

- **registry(pubblico o privato):** contiene immagini docker, il registro pubblico ufficiale è Docker Hub.

3.3.2 Ambiente e Filesystem

Linux Container è un ambiente di virtualizzazione system-level nel quale è possibile eseguire diversi ambienti Linux virtuali (container),isolati tra loro, su una sola macchina reale avente il kernel Linux. LXC oltre che emulare un intero sistema può essere usato anche per isolare una singola applicazione, differenziando quindi i container di tipo "sistema" da quelli di tipo "applicazione"[5]. L'uso di LXC viene poi sostituito dalla libreria libcontainer che si prenderà carico di risolvere il medesimo problema. Un altro componente importante di Docker è il file system per container AuFS (Advanced Multi-Layered Unification Filesystem) , un file-system a strati che permette tra le altre cose di gestire il versioning dei container. L'immagine del container è salvata una sola volta, se un processo utilizza quell'immagine viene creato un container con l'immagine citata e successivamente viene salvata solo la differenza tra l'immagine citata e l'immagine generata dal lavoro effettuato nel container(principio del "copy-on-write").

3.3.3 DockerFile

Consiste in uno script con le precise istruzioni su come creare il container;una lista di istruzioni che rendono il processo di creazione facile e chiaro. Il dockerfile è un file di testo simile ad un Makefile, consiste in una lista di comandi per il terminale con lo scopo di settare tutte le variabili d'ambiente in modo opportuno ,scaricare ed installare tutte le dipendenze richieste. In figura 3.2 è mostrato un esempio di dockerfile. Alcune delle istruzioni che troviamo in un dockerfile sono:

- **FROM:** specifica l'immagine base dalla quale partire,presente in tutti i dockerfile.
- **ENTRYPOINT:** specifica l'eseguibile o il comando da eseguire nel contenitore creato a partire dall'immagine, di solito ultima istruzione di un dockerfile.
- **RUN:** specifica un comando da eseguire durante la costruzione dell'immagine.

```
# https://github.com/francescou/docker-compose-ui
# DOCKER-VERSION 1.12.3
FROM python:2.7-alpine
MAINTAINER Francesco Uliana <francesco@uliana.it>

RUN pip install virtualenv

RUN apk add -U --no-cache git

COPY ./requirements.txt /app/requirements.txt
RUN virtualenv /env && /env/bin/pip install --no-cache-dir -r /app/requirements.txt

COPY . /app

VOLUME ["/opt/docker-compose-projects"]

COPY demo-projects /opt/docker-compose-projects

EXPOSE 5000

CMD []
ENTRYPOINT ["/env/bin/python", "/app/main.py"]
```

Figura 3.2: Esempio di DockerFile

Quando il dockerfile viene eseguito da un terminale, viene creato il container, e avviato successivamente.

3.3.4 Docker Compose

È uno strumento (facoltativo) che permette di definire ed eseguire applicazioni docker su più contenitori, un esempio è mostrato in figura 3.3. In

compose un' applicazione è composta da uno o più servizi o micro-servizi ,dove un servizio corrisponde ad un container dedicato alla fruizione del servizio stesso. Compose non fa altro che definire automaticamente una rete dedicata all'applicazione e collegare tutti i servizi(container) a questa rete. Dove ogni blocco di istruzioni indica un container che eseguirà un servizio , per ogni container i campi :

- **image:** indica l'immagine dalla quale partire per creare il singolo container.
- **container name:** indica il nome da dare al singolo container.
- **networks:** indica la rete alla quale il container andrà collegato,di solito tutti i container in un docker compose sono collegati alla stessa rete per poter comunicare tra loro e scambiarsi informazioni e servizi.
- **command:** indica le istruzioni da eseguire una volta che il container sarà creato.

```
version: '2'

services:
  proxy:
    build: nginx/
    container_name: "portainer-proxy"
    ports:
      - "80:80"
    networks:
      - local

  templates:
    image: portainer/templates
    container_name: "portainer-templates"
    networks:
      - local

  portainer:
    image: portainer/portainer
    container_name: "portainer-app"
#Automatically choose 'Manage the Docker instance where Portainer is running' by adding <--host=unix:///var/run/docker.sock>
    command: --templates http://templates/templates.json
    networks:
      - local
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - /opt/portainer/data:/data

  watchtower:
    image: v2tec/watchtower
    container_name: "portainer-watchtower"
    command: --cleanup portainer-app portainer-watchtower portainer/templates
    networks:
      - local
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock

networks:
  local:
    driver: bridge
```

Figura 3.3: Esempio di Docker-Compose

3.4 Dalla definizione all'esecuzione

Per un corretto uso di container docker è consigliabile costruire un immagine, creare un container a partire dall'immagine creata e poi avviare quel container per lavorare. Per farlo la piattaforma docker mette a disposizione una serie di semplici comandi per il terminale che andremo ad introdurre più avanti. Nulla vieta di non seguire questi passi e, magari, partire da un'immagine creata da terzi , in questo caso è però consigliabile prendere le dovute precauzioni sulla fonte. Di seguito andrò a descrivere per passi la definizione o costruzione di un'immagine, la creazione di un container a partire dall'immagine creata e l'esecuzione di un container. A fine sezione ci sarà un breve esempio di uso dei comandi per effettuare le suddette operazioni con container docker.

1. **Costruzione immagine:** un' immagine è costituita a partire da strati(AuFS) , dove ogni strato è un insieme di file; un file viene letto nello strato più alto in cui si trova,ma può essere scritto solo nello strato più alto in assoluto. La costruzione di un'immagine consiste nell'eseguire un dockerfile (adatto) ,a questo punto : l'immagine specificata dall'istruzione FROM viene scaricata da registry e posizionata in un host. Questa immagine viene usata come strato di base della nuova immagine. Ogni istruzione successiva del dockerfile (RUN) andrà a formare un nuovo strato dell'immagine personalizzata. È però consigliabile limitare il numero di strati di un'immagine,quindi minimizzare l'uso dell'istruzione RUN.

2. **Costruzione container:** un container consiste in un file-system popolato da meta-dati, il filesystem è ottenuto dall'immagine iniziale più un nuovo strato scrivibile,solo questo strato viene realmente allocato sull' host. La stessa immagine può essere condivisa da più contenitori, sarà infatti non scrivibile.

3. **Esecuzione container:** prima di eseguire un container si dovrà allocare le risorse destinategli, successivamente si potrà avviare il conteni-

tore ,come ultima istruzione, in genere ,è eseguita ENTRYPOINT che chiuderà l'esecuzione del dockerfile . A fine esecuzione il container è avviato e pronto all'uso.

Per effettuare le operazioni appena descritte il passo zero sarà scaricare ed installare docker. Come primo passo bisogna creare un'immagine personalizzata, per farlo bisogna recarsi nella directory contenente il dockerfile e da terminale digitare il comando :

```
$ docker-build ubuntu-img
```

Una volta creata un'immagine personalizzata si potrà creare un container a partire da quell'immagine, digitando :

```
$ docker create --name=test_ubuntu ubuntu-img
```

A questo punto non resta che far partire il container chiamato test ubuntu appena creato digitando :

```
$ docker start test_ubuntu
```

3.5 Caratteristiche Docker

La vera potenzialità di docker viene fuori quando 2 o più container comunicano tra loro, attraverso una rete interna privata (può anche essere pubblica) per scambiarsi informazioni e/o servizi, questo concetto introduce la modalità swarm prevista da docker,la quale permette di creare una rete di nodi(host), con annessa gestione di essi . Un nodo è un'istanza di docker engine che partecipa ad un raggruppamento di nodi(cluster), inoltre ci sono due tipi di nodo : il nodo worker riceve la richiesta di un task e lo esegue, il nodo manager che riceve da utenti la richiesta del rilascio di un servizio e divide esso in una serie di compiti (task) da assegnare ai nodi worker,gestisce cluster e suoi nodi. Il concetto di nodo in docker introduce quello di orchestrazione, cioè di gestione dei nodi sulla rete. Una rete di host può essere gestita seguendo 2 approcci :

- **Orchestrazione:** per orchestrazione si intende un modello per la gestione del lavoro degli host sulla rete. In particolare sarà un host (direttore d'orchestra) a gestire il flusso dei dati tra gli altri host della rete per il raggiungimento di uno scopo, che può essere la realizzazione di un servizio.

- **Coreografia:** questo approccio a differenza del precedente non prevede un host speciale che dirige il lavoro degli altri , ma bensì , ogni host sulla rete sa cosa fare e con chi comunicare (come in una coreografia) , spetterà all'host che si trova al termine del flusso dei dati rendere il servizio tale.

Per quanto riguarda la gestione della rete la piattaforma Docker consente tale gestione per attuare una comunicazione in rete tra contenitori. In particolare durante l'installazione Docker crea 3 reti :bridge,host,none, ed è anche possibile aggiungerne altre. Quando un contenitore viene mandato in esecuzione docker engine gli assegna un indirizzo IP libero della rete bridge, è possibile anche collegare il container con una rete diversa(opzione : - -network=nuovarete). I contenitori possono comunicare tra loro se conoscono la posizione assoluta(indirizzo IP e porta) dei servizi presenti in rete; è anche possibile rendere questi servizi visibili all'host o al di fuori dell' host. Un container può anche essere associato a più reti. La comunicazione tra container appena descritta ha come potenzialità quella di permettere di comporre container per realizzare un unico servizio o applicazione. Per composizione si intende la possibilità di definire ed eseguire applicazioni multi-container. Tali applicazioni possono essere viste come uno stack,costituito dall'insieme dei servizi(contenitori) che la compongono. La specifica di uno stack va descritta in un file YAML, la struttura del file è quella del docker compose. Le funzionalità di base di Docker permettono di gestire la composizione solo nei casi più semplici[6],nei quali ci sono diversi container(servizi) e ognuno ha un numero prefissato di istanze. Per composizioni più complesse ci sono notevoli estensioni per docker che ne permettono la gestione. Per quanto riguarda l'orchestrazione in docker sono presenti notevoli elementi nativi che

la permettono , oltre agli strumenti nativi per docker ci sono una serie di strumenti noti per l'orchestrazione di container docker, tra i più diffusi Amazon EC2 Container Service, Google Container Engine,Kubernetes, Apache mesos. L'orchestrazione come anche la composizione sono degli strumenti fondamentali per il rilascio in produzione di applicazioni multi-servizi e multi-contenitori in un nodo o in cluster di nodi. Per mettere in atto un'orchestrazione di container docker deve essere eseguito in modalità swarm,ciò comporta che il nodo sul quale viene eseguito vada a far parte di uno swarm di nodi. In uno swarm le azioni eseguite dai nodi si distinguono in:

- **Servizio:** è una funzionalità da realizzare nello swarm, costituisce il centro del dibattito tra amministratori e lo swarm stesso.

- **Task:** un'istanza di un container ,l'esecuzione di un servizio dipende dall'esecuzione di un certo numero di task relativi a quel servizio.

Per aggiungere un nodo al cluster(o swarm) bisogna eseguire la piattaforma docker in modalità swarm,il nodo può essere distribuito su più macchine fisiche,virtuali,o su cloud . In un cluster ci posso essere più nodi worker e più nodi manager. Un ultimo aspetto fondamentale è il supporto che i sistemi operativi danno alla piattaforma docker, di seguito riporto alcune informazioni sul supporto da parte dei sistemi Windows, macOS e Linux :

- **Windows:** docker è compatibile con i sistemi windows ma funziona in modo sostanzialmente diverso dal solito;viene installata una macchina virtuale linux(virtual box) nella quale sarà eseguito il container,questa macchina virtuale è eseguita dall'Hypervisor di windows (Hyper-V),quindi una virtualizzazione ibrida senza apparenti vantaggi. Nelle ultime versioni di windows 10 e windows 10 server è supportato un container linux nativo quindi è possibile scegliere tra la modalità con macchina virtuale e hypervisor citata prima e la modalità container nativo .Se si sceglie la modalità container nativo si avranno gli stessi vantaggi di usare un sistema linux.

- **MacOS:** docker è compatibile con i sistemi MacOS ma anche in questo caso avviene una virtualizzazione ibrida, viene installato un Hyperkit,gestore virtualizzazione per macOS, al posto di virtual box ma il funzionamento è il medesimo dei sistemi windows con hypervisor.

- **Linux:** docker è concepito per Linux e quindi compatibile con qualsiasi distribuzione che abbia un kernel linux[13].

Nel prossimo capitolo ci sarà una panoramica sulla piattaforma di hosting Github ,sulle api offerte dalla piattaforma per interrogare il sistema e sulle limitazioni che ci sono su di esse. Successivamente sarà introdotto il concetto di scraping e il linguaggio xpath che saranno fondamentali nell'analisi che sarà esposta nel capitolo 5. Per concludere il capitolo ci saranno delle sezioni che descrivono la progettazione dello script che ho usato per portare effettuarel'analisi, dalla scelta delle informazioni da studiare, alla sua implementazione vera e propria per poi terminare con esempi sull'uso dello stesso script per personalizzare le proprie ricerche.

Capitolo 4

Creazione del dataset: github mining

Questo capitolo si fa carico di descrivere le varie tecnologie e gli ambienti usati per condurre la ricerca. Saranno visti nel dettaglio i singoli argomenti, iniziando con esporre un quadro generale sulla piattaforma di hosting on-line Github, dalla storia alle sue caratteristiche base. Successivamente si affronterà l'argomento delle api REST messe a disposizione degli utenti da parte di github per interrogare il sistema e delle limitazioni che vengono fatte sul loro uso. Sarà poi introdotto il concetto di scraping e il linguaggio xpath per estrarre elementi da documenti XML. Conclude il capitolo un quadro sulla progettazione e implementazione dell'algoritmo di ricerca automatica. In sintesi nelle sezioni successive andrò a spiegare nel dettaglio lo studio effettuato sullla piattaforma github e i metodi per poter interrogare la piattaforma ,personalizzando le interrogazioni. Una volta acquisito il giusto background oltre che su github anche su docker e l'uso che ne viene fatto ho iniziato a progettare un algoritmo che :

1. **Interrogasse github:** con lo scopo di ovviare alle limitazioni imposte dal sistema ed ottenere circa 200000 progetti come risultati .

2. **Analizzasse risultati:** una volta raccolti i 200000 progetti , uno ad uno analizzasse i repository e salvasse i dati per l'elaborazione.

3. **Elaborasse e stampasse i risultati:** la stampa avviene in un foglio di calcolo così da potermi permettere in modo semplice di creare dei grafici relativi a quei dati e analizzarli .

4. **Permettesse personalizzazione:** come ultimo compito ho reso personalizzabile la ricerca tramite il mio algoritmo in modo da permettere a chiunque di effettuare analisi su una qualsiasi parola chiave.

4.1 Github

Github è una piattaforma sociale per programmatori nella quale è possibile creare il proprio repositry(directory di file, in genere il termine repository è usato per indicare la cartella contenente i file sorgente di un programma o applicazione) e condividerlo con l'intera community o visionare milioni di repository condivisi in precedenza dalla community. Per entrare a fare parte della community basta iscriversi a github.com , fornendo un indirizzo email valido e confermando l'iscrizione tramite il link inviato all'email fornita, una volta confermata l'iscrizione si è a tutti gli effetti membri della piattaforma. Una volta entrati come utenti github da a disposizione 1 GB di spazio gratis per creare il proprio repository, che sarà però pubblico, a meno che non si scelgano dei piani tariffari i quali permettono,tra le altre cose, di oscurare i propri repository alla community rendondoli privati. Un utente github ha anche la possibilità di clonare un qualsiasi repository presente nella community,purchè sia un repository pubblico, nel proprio storage e farci ciò che vuole. Github è basato sul sistema di controllo delle versioni 'git' creato dal fondatore di linux Linus Torvalds, questo sistema consiste in un gestore di aggiornamenti per un progetto, cioè aggiorna il progetto non sovrascrivendo nulla di esso, semplicemente portandolo ad una versione successiva. Oggi github è nel suo campo la piattaforma più usata, gli scopi per i quali gli utenti la usano sono diversi :

- area di lavoro professionale con controllo versioning.

- condividere area di lavoro in più persone, anche dislocate tra loro.
- semplice cloud per file,fogli di lavoro o foto.
- didattico , in quanto è possibile insinuarsi in qualsiasi file di qualsiasi repository e consultarlo, scaricarlo e modificarlo, purché i file siano pubblici ovviamente.
- sociale , in quanto è possibile seguire progetti o utenti interessanti e scambiare messaggi, opinioni.

4.1.1 Breve storia

La storia di github inizia circa 5 anni prima della sua nascita quando si sente l'esigenza di un sistema come git,un sistema di controllo delle versioni. Siamo nel 2005 e nel corso dello sviluppo della versione 2.6.12 del kernel linux da parte di Linus Torvalds e i suoi collaboratori si avverte il bisogno di un gestore degli aggiornamenti, un software che sarebbe stato in grado di tornare indietro se l'installazione degli aggiornamenti non fosse andata a buon fine o se si fossero riscontrati degli errori negli aggiornamenti appena installati. In pochi mesi viene progettato e presentato git in concomitanza con la nuova versione del kernel linux che sarà la prima gestita con il nuovo sistema di versioning [7]. Qualche anno più tardi , nel 2009 , viene fondato Github da Tom Preston-Werner, Chris Wanstrath e PJ Hyett [8], una piattaforma basata sul sistema git di Linus Torvalds e sul concetto di social network. Nel giro di pochi anni github diventa la piattaforma di hosting più frequentata dagli sviluppatori di tutto il mondo e conta più di 10.000.000 di repository.

4.1.2 Api github

Github mette a disposizione degli utenti una vasta gamma di api REST per personalizzare e rendere automatiche le ricerche dei repository. In particolare è possibile interrogare la piattaforma attraverso una stringa(query) personalizzabile, la ricerca può basarsi su uno dei seguenti campi :

- **repositories:** la ricerca viene effettuata tra i repository presenti.
- **commits:** la ricerca è effettuata tra tutti gli aggiornamenti presenti nella piattaforma,un singolo aggiornamento è quello che ogni utente apporta al proprio repository.
- **code:** la ricerca si basa su file contenenti del codice.
- **users:** la ricerca viene effettuata tra gli utenti iscritti alla piattaforma.

Inoltre è possibile basare la propria ricerca su : issues, topics o text match metadata . Per ognuno di questi campi è possibile settare una serie di parametri per filtrare la propria ricerca come ad esempio : l'ordine, la data di creazione, il linguaggio utilizzato ,la licenza , il numero di stelle , effettuare la ricerca solo in un determinato file del repository, l'autore del commit ed un'altra lunga serie[9]. Qui sotto mostro due esempi di uso di api github :

```
https://api.github.com/search/commits?=repo:octocat/Spoon-Knife+css
```

Tramite l'uso di quest' api interroghiamo github sui commit relativi al CSS nello specifico repository "Spoon-Knife" dell'utente "octocat".

```
https://api.github.com/search/repositories?=docker+created%3A2014
```

Questa api permette di avere come risultato tutti i repository che usano docker creati nel 2014. La risposta che github da ad una interrogazione come questa consiste in una lista di elementi,in formato JSON, ogni elemento della lista rappresenta un repository che ha soddisfatto la richiesta,ed ha notevoli campi tra cui:nome repository, linguaggio usato , numero stelle repository , url del repository.In figura 4.1 e 4.2 un esempio di un elemento della lista JSON:

Figura 4.1: esempio lista JSON (pt.1)

```
    "private": false,
    "html_url": "https://github.com/progrium/buildstep",
    "description": "Buildstep uses Docker and Buildpacks to build applications like Heroku",
    "fork": false,
    "url": "https://api.github.com/repos/progrium/buildstep",
    "forks_url": "https://api.github.com/repos/progrium/buildstep/forks",
    "keys_url": "https://api.github.com/repos/progrium/buildstep/keys{/key_id}",
    "collaborators_url":
"https://api.github.com/repos/progrium/buildstep/collaborators{/collaborator}",
    "teams_url": "https://api.github.com/repos/progrium/buildstep/teams",
    "hooks_url": "https://api.github.com/repos/progrium/buildstep/hooks",
    "issue_events_url": "https://api.github.com/repos/progrium/buildstep/issues/events{/number}",
    "events_url": "https://api.github.com/repos/progrium/buildstep/events",
    "assignees_url": "https://api.github.com/repos/progrium/buildstep/assignees{/user}",
    "branches_url": "https://api.github.com/repos/progrium/buildstep/branches{/branch}",
    "tags_url": "https://api.github.com/repos/progrium/buildstep/tags",
    "blobs_url": "https://api.github.com/repos/progrium/buildstep/git/blobs{/sha}",
    "git_tags_url": "https://api.github.com/repos/progrium/buildstep/git/tags{/sha}",
    "git_refs_url": "https://api.github.com/repos/progrium/buildstep/git/refs{/sha}",
    "trees_url": "https://api.github.com/repos/progrium/buildstep/git/trees{/sha}",
    "statuses_url": "https://api.github.com/repos/progrium/buildstep/statuses/{sha}",
    "languages_url": "https://api.github.com/repos/progrium/buildstep/languages",
    "stargazers_url": "https://api.github.com/repos/progrium/buildstep/stargazers",
    "contributors_url": "https://api.github.com/repos/progrium/buildstep/contributors",
    "subscribers_url": "https://api.github.com/repos/progrium/buildstep/subscribers",
    "subscription_url": "https://api.github.com/repos/progrium/buildstep/subscription",
    "commits_url": "https://api.github.com/repos/progrium/buildstep/commits{/sha}",
    "git_commits_url": "https://api.github.com/repos/progrium/buildstep/git/commits{/sha}",
    "comments_url": "https://api.github.com/repos/progrium/buildstep/comments{/number}",
    "issue_comment_url": "https://api.github.com/repos/progrium/buildstep/issues/comments{/number}",
    "contents_url": "https://api.github.com/repos/progrium/buildstep/contents/{+path}",
    "compare_url": "https://api.github.com/repos/progrium/buildstep/compare/{base}...{head}",
    "merges_url": "https://api.github.com/repos/progrium/buildstep/merges",
    "archive_url": "https://api.github.com/repos/progrium/buildstep/{archive_format}{/ref}",
    "downloads_url": "https://api.github.com/repos/progrium/buildstep/downloads",
    "issues_url": "https://api.github.com/repos/progrium/buildstep/issues{/number}",
    "pulls_url": "https://api.github.com/repos/progrium/buildstep/pulls{/number}",
    "milestones_url": "https://api.github.com/repos/progrium/buildstep/milestones{/number}",
    "notifications_url": "https://api.github.com/repos/progrium/buildstep/notifications{?
since,all,participating}",
    "labels_url": "https://api.github.com/repos/progrium/buildstep/labels{/name}",
    "releases_url": "https://api.github.com/repos/progrium/buildstep/releases{/id}",
    "deployments_url": "https://api.github.com/repos/progrium/buildstep/deployments",
    "created_at": "2013-01-02T22:11:42Z",
    "updated_at": "2018-01-16T08:08:24Z",
    "pushed_at": "2017-04-20T16:25:55Z",
    "git_url": "git://github.com/progrium/buildstep.git",
    "ssh_url": "git@github.com:progrium/buildstep.git",
    "clone_url": "https://github.com/progrium/buildstep.git",
    "svn_url": "https://github.com/progrium/buildstep",
    "homepage": "",
    "size": 2895,
    "stargazers_count": 904,
    "watchers_count": 904,
    "language": "Groovy",
    "has_issues": true,
    "has_projects": true,
    "has_downloads": true,
    "has_wiki": true,
    "has_pages": false,
    "forks_count": 291,
    "mirror_url": null,
    "archived": false,
    "open_issues_count": 13,
    "license": {
      "key": "mit",
      "name": "MIT License",
      "spdx_id": "MIT",
      "url": "https://api.github.com/licenses/mit"
    },
    "forks": 291,
    "open_issues": 13,
    "watchers": 904,
    "default_branch": "master",
    "permissions": {
      "admin": false,
      "push": false,
      "pull": true
    },
    "score": 11.174894
  }
 ]
}
```


Figura 4.2: esempio lista JSON (pt.2)

4.1.3 Limitazioni

Il servizio di risposta alle api da parte di github ha delle limitazioni sia nel tempo che nel numero di risultati .Potenzialmente potrei inserire un' api in un programma che la modella e la esegue migliaia di volte ,github per non permette di sovraccaricare i propri server con milioni di richieste al secondo,quindi, impone due tipi di limitazioni :

- **rate-limit:** un utente non iscritto a github sarà identificato con il proprio indirizzo IP e può effettuare 10 richieste al secondo con un massimo di 60 richieste all'ora. Il discorso invece cambia con un utente iscritto a github, infatti l'utente sarà identificato con il suo id utente e ,soprattutto, avrà a disposizione 30 richieste al minuto per un massimo di ben 5000 richieste all'ora.

- **paginazione:** per limitare le risorse impiegate github ha impostato per i risultati delle richieste delle limitazioni in termini di numero, infatti effettuando una richiesta la risposta del sistema sarà solo parziale,per default solo 100 saranno i risultati visualizzati, inoltre una richiesta non può avere più di 1000 risultati visualizzati[10]. Se io volessi visualizzare tutti i 2900 repository di una qualche tipologia dovrei impostare la paginazione a 1000 risultati per pagina ed inoltre effettuare tre richieste al sistema github , nella prima richiederò la pagina uno dei miei risultati(primi 1000), nella seconda chiederei la pagina due di risultati(secondi 1000) e nella terza chiederei la terza ed ultima pagina di risultati(ultimi 900).Dove per risultati si intendono gli elementi della lista citati a fine sezione 4.1.2 .

4.2 Tecnologie ausiliarie

Per arrivare ad un'analisi consistente sorge il bisogno di analizzare ogni singolo elemento della lista dei risultati e estrapolare le informazioni interessanti per poi elaborare statistiche grafici e conclusioni. L'analisi che voglio

effettuare è sui container docker , sul loro uso e sulla loro diffusione ,per una analisi preliminare non ho avuto bisogno di particolari tecnologie ,è bastato un sistema operativo linux, Ubuntu 17.10 per la precisione, con installati i pacchetti python e git. Dopo la prima analisi mi sono reso conto che le vere potenzialità di questo algoritmo sarebbero venute fuori se avessi analizzato oltre che il semplice campo dell'elemento della lista(nome,linguaggio,numero stelle,ecc), anche il link al quale esso porta, in particolare per quanto riguarda il campo "html url". Il passo successivo è stato analizzare la pagina html relativa ad ogni elemento della lista dei risultati tramite l'ausilio di alcune tecnologie che andrò ad introdurre di seguito.

4.2.1 Scraping e XPATH

Lo scraping (o web scraping) è una tecnica software per estrarre informazioni da una pagina internet e trasformare o rielaborare quelle informazioni per creare nuovi contenuti[11].Lo scraping si concentra principalmente sul prendere dei dati non strutturati, in genere in formato HTML, e dare loro una struttura così che possano essere salvati , analizzati. Punto focale dello scraping è ovviamente rendere automatico tutto il processo di prendere dati ,strutturarli ed elaborarli a piacimento. Un tipico strumento per attuare scraping e il linguaggio xpath, usato in questa analisi da me. Il linguaggio xpath è tipicamente usato per indicare parti di un documento XML. Xpath è un tipico linguaggio usato per fare scraping,opera su una visione logica del documento XML il quale viene costruito come un albero [12]spetta al linguaggio accedere ai nodi dell'albero. Oltre alla restituzione dell'elemento nel nodo xpath è usato per la manipolazione dello stesso. Un esempio di un'istruzione xpath per estrapolare l'informazione indicata dal rettangolo rosso (data ultimo commit) di figura 4.3:

```
tree.xpath('/html/body/div[4]/div/div/div[2]/div[1]/div[6]/span[1]/span/relative-time')
```

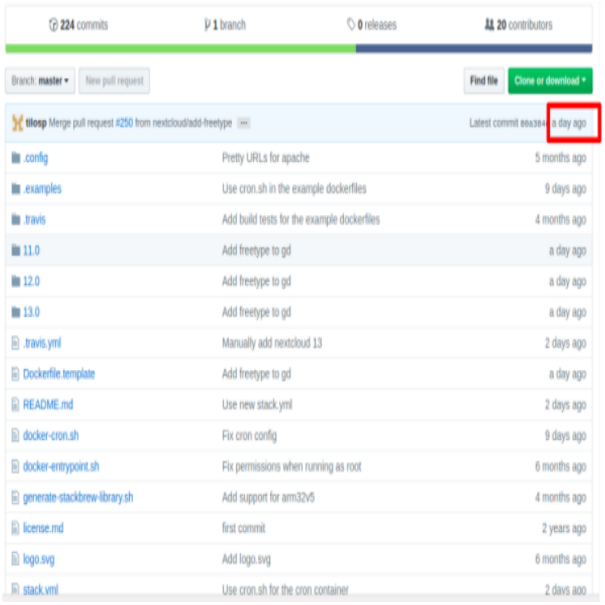

Figura 4.3: Esempio pagina github di un repository

Questa istruzione xpath (libreria lxml per linguaggio python), applicata alla pagina mostrata in figura 4.3 , restituisce la stringa “a day ago”.

```
tree.xpath('/html/body/.../div[1]/div[6]/span[1]/span/relative-time/@datatime')
```

Questa istruzione invece, sempre applicata alla pagina di figura 4.3, restituisce il contenuto del parametro datatime dello span relative-time “ 2018-02-05 17:12:22 ”.

4.3 Implementazione script

Una volta fatte le dovute considerazioni ho quindi progettato un algoritmo che in un primo momento, per ogni giorno dell’anno , legge quante pagine di risultati ci sono da visionare ,sfruttando l’uso della query citata poco prima, totalizzando un totale di minimo 365 query per ogni anno. Dal momento in cui questa prima parte di esecuzione termina , conoscendo per ogni singolo giorno quante pagine di risultati chiedere , si passa alla composizione vera e propria delle query , quelle che mi restituiranno la lista di risultati, nel nostro

caso si parla di circa 200.000 elementi appartenenti alla lista. Nell'eseguire le query mi attengo al rate-limit quindi rallento l'esecuzione di circa 2 secondi per query, la quale ci impiega circa 1 secondo per avere una risposta dal sistema, per un totale di 3 secondi per ogni query che in questa prima fase dell'analisi saranno circa 5000 . I risultati delle query vengono reindirizzati tutto nello stesso file di testo che a fine esecuzione conterrà i circa 200.000 elementi di cui parlavo prima ed avrà una dimensione di poco più di 1 GB. Terminata questa prima fase mi trovo con un file di testo da esaminare , quindi lo scorro riga per riga e per ogni elemento salverò solo i dati che mi interessano , quando ho finito di scorrere il file elaboro tutti i dati raccolti e li stampo i un foglio di calcolo per poi poterli trasformare semplicemente nei grafici che mostrerò nelle sezioni successive. Successivamente o contemporaneamente a questa fase vado ad analizzare ogni singola pagina html relativa ad un elemento della lista, essendo questa analisi molto più lunga in termini di tempo della fase precedente ho preferito dividerle. Anche in questa fase, come quella citata poco fa, lo scopo e salvare alcune informazioni che mi interessano , questa volta però devo effettuare una query per avere la pagina di mio interesse,una volta effettuata la query posso esaminare la pagina e salvare le informazioni che mi interessano per poi passare alla prossima query. Come detto questa fase impiega un tempo ingente in quanto deve effettuare più di 200.000 query , e c'è da considerare il rate-limit da rispettare. Al termine di questa fase elaboro i dati raccolti dai 200.000 repository che usano docker e li trascrivo in un foglio di calcolo da affiancare a quello citato nella fase precedente come mostro nella sezione 4.3.5. La progettazione non è stata priva di intoppi ed errori ,tra le difficoltà maggiori che ho riscontrato ci sono : rispettare i limiti imposti da github che ha portato alla conseguente gestione di 5000 query cioè alla gestione dei giorni dell'anno , manipolare file di testo da 1 GB e inoltre l'ingente tempo tra una computazione e l'altra. In figura 4.4 mostro l'architettura del software che ho prodotto , nelle sezioni successive andrò a mostrare nel dettaglio il compito di ogni componente.

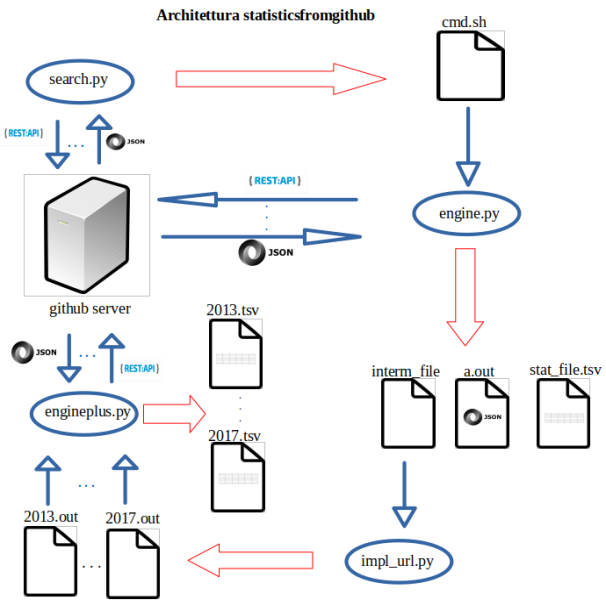

Figura 4.4: Architettura del software prodotto

4.3.1 Il repository 'statisticsfromgithub'

Il codice che ho usato per effettuare la ricerca di cui ho parlato nella sezione precedente è a questo indirizzo : 'https://github.com/cirwillgio/statisticsfromgithub'. Ho prodotto 4 file per un totale di 404 linee di codice sorgente (SLOC). Di seguito descrivo i file presenti prima e dopo l'esecuzione :

- **search.py :** è lo script principale , in questo file è presente la fase di estrapolazione delle informazione relative alle pagine di risultati e la fase di composizione delle query vere e proprie .Questo script genera un file intermedio cmd.sh,il quale contiene tutte le query che devono essere effettuate,generato il file chiama engine.py il quale si fa carico di eseguire le query.

- **engine.py :** in questo file è presente la fase di esecuzione delle query, con annesso rallentamento del tempo per il rispetto del ratelimit, e l'elaborazione con relativa stampa, in un foglio di calcolo chiamato stat_file.tsv ,dei dati relativi alla prima fase di stampa descritta nel-

la sezione precedente. Oltre al file di statistiche appena citato questo script genera un file a.out(1GB) in cui è presente l'intera lista dei risultati analizzati e un file_interm file il quale contiene tutte le informazioni interessanti trovate nel file a.out, tra le altre anche gli indirizzi delle pagine html di ogni singolo elemento.

- **impl_url.py :** questo script si occupa di leggere il file interm_file generato da engine.py e di estrapolare tutti i link alle pagine html, successivamente genera un file per ogni anno del tipo ‘ anno.out ‘ nel quale inserisce tutti i link relativi alle pagine html dei repository di quell'anno . Ho diviso i file per poi poter dividere le esecuzioni ed eseguirle in parallelo tramite l'ausilio di qualche utente github.

- **engineplus.py :** questo script contiene la fase di analisi pagina html ed elaborazione e stampa dei risultati, ovviamente riferendosi ad un solo anno la tabella di risultati generata , nominata ‘statANNO.tsv ’ si riferirà solo a quell'anno , se si ha intenzione di effettuare la ricerca per più anni bisognerà eseguire più volte questo script, una per ogni file generato da impl url ,poi affiancare i fogli di calcolo risultanti dalle varie esecuzioni.

- **aut.py :** questo file è vuoto e ci sarà il bisogno di andare a inserire la seguente stringa affinché lo script funzioni : ‘ aut='nomeutente:password' ’ dove per nome utente e password si intendono le credenziali di accesso a github.

4.3.2 Uso del tool per personalizzare le ricerche

Per personalizzare la ricerca bisognerà recarsi nell'indirizzo citato a inizio sezione 4.3.4 e scaricare il repository, oppure tramite terminale usando il comando git clone seguito dall'indirizzo. Scaricato il repository bisognerà modificare il file aut.py , aggiungendo la stringa oppurtuna come spiego nella sezione precedente, una volta modificato il file aut.py si potrà procedere con

l'esecuzione. Come detto prima l'esecuzione è divisa in fasi quindi ci sono più file da eseguire, il primo è search.py , il programma prende in input 4 parametri :

- **directory di lavoro:** utilizzando il parametro -i si inserisce il percorso della cartella nella quale si vuole eseguire la computazione (consigliata cartella del repository stesso) .
- **anno inizio ricerca:** tramite il parametro -s si specifica l'anno dal quale si vuole partire ad effettuare la ricerca.
- **anno fine ricerca:** con il parametro -e si indica l'anno in cui far terminare la ricerca.
- **parola chiave:** utilizzando il parametro -k si va ad inserire la parola chiave,intorno alla quale sarà effettuata la ricerca.

Qui mostro un esempio di uso :

```
$ python search.py -i /home/test/statisticsfromgithub -s 2015 -e 2017 -k docker
```

Come risultato di questa esecuzione avremo un file stat file.tsv contenente , organizzate per mese , le statistiche relative a : numero repository analizzati, media stelle , media partecipanti e linguaggi di programmazione utilizzati . Un esempio è mostrato nella figura 4.5:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	MONTH	YEAR	TOTAL COUNT	MONTHLY STAR AVERAGE	CONTRIBUTORS AVERAGE	Shell	Python	Javascript	PHP	Ruby	Makefile	Java	Go	Null	Others
2	January	2015	2619	9	1	945	149	94	43	101	76	29	57	975	150
3	February	2015	2711	6	1	903	117	127	48	100	103	39	48	955	271
4	March	2015	3210	12	1	1091	161	144	57	115	84	55	68	1139	296
5	April	2015	3147	7	1	1045	147	146	53	135	93	48	79	1143	258
6	May	2015	3369	7	1	1110	182	151	63	110	109	44	66	1242	292
7	June	2015	3273	8	1	1057	153	164	55	102	101	53	76	1233	279
8	July	2015	3531	7	1	1115	169	141	66	127	92	54	75	1379	313
9	August	2015	3623	6	1	1174	184	129	67	115	127	52	104	1390	281
10	September	2015	3654	9	1	1117	166	182	76	137	94	75	111	1384	312
11	October	2015	3831	3	1	1192	268	179	99	117	106	73	75	1416	306
12	November	2015	4479	5	1	1396	235	197	81	145	136	100	90	1711	388
13	December	2015	4214	4	1	1369	206	191	91	120	142	79	76	1540	400
14															
15		2015	41661	7.33333333333	0.000288039173328	###	2137	1845	799	1424	1263	701	925	15507	3546
16															
17	January	2016	4813	4	1	1462	221	282	114	122	137	91	102	1893	389
18	February	2016	5011	6	1	1592	208	233	103	136	108	100	99	1953	479
19	March	2016	5561	3	1	1792	290	271	124	136	168	137	99	2081	463
20	April	2016	5039	4	1	1560	234	279	110	123	161	114	88	1904	466
21	May	2016	5480	4	1	1662	277	298	116	129	153	93	115	2161	476
22	June	2016	5364	3	1	1575	256	296	114	142	126	122	101	2148	484
23	July	2016	5586	4	1	1599	293	316	142	143	156	120	97	2208	512
24	August	2016	6107	2	1	1788	301	321	155	170	145	129	110	2442	546
25	September	2016	6017	2	1	1754	301	300	146	119	137	140	122	2370	628
26	October	2016	6437	2	1	1791	347	363	168	157	172	145	122	2549	623
27	November	2016	6542	1	1	1773	341	351	167	143	146	179	134	2641	667
28	December	2016	5888	4	1	1705	314	306	157	124	157	131	120	2331	543
29															
30		2016	67845	3.75	0.000176873756356	###	3383	3616	1616	1644	1766	1501	1309	26681	6276
31															
32	January	2017	7064	2	1	1956	405	401	205	142	159	140	141	2782	733
33	February	2017	7114	1	1	1919	416	425	215	159	204	187	131	2750	708

Figura 4.5: Esempio di risultato per l'esecuzione del tool

Se si vuole effettuare una ricerca dettagliata e quindi analizzare ogni singola pagina html relativa ad un elemento della lista dei risultati bisogna eseguire il file impl_url. L'esecuzione comporta la generazione di un file contenente i link alle pagine html,per ogni anno. Questi file generati ci serviranno poi per l'estrazione di statistiche quali media commit per giorno , media commit per numero repository , repository attivi/non attivi(riconosciuti in base alla data dell'ultimo commit , per essere attivo un repository ha ricevuto un commit dopo il 01-01-2017) tramite l'esecuzione di un ultimo script. Prima di eseguire il file impl_url va modificata una stringa in questo file, in particolare se si è effettuata una ricerca dal 2013 al 2017 il file impl_url deve presentarsi come in figura 4.6:

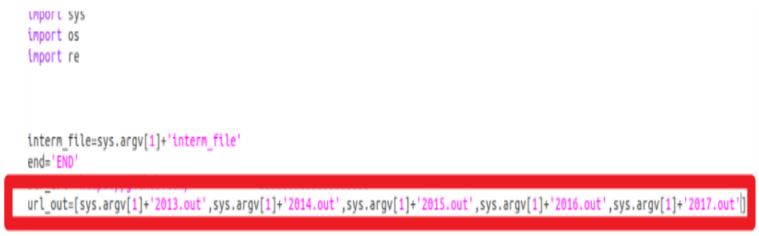

Figura 4.6: intestazione del file 'impl_url'

Se la ricerca presenta più anni basta aggiungere o rimuovere un elemento come mostrato di seguito:

```
AGGIUNGERE ELEMENTO:
url_out=[sys.argv[1]+'2012.out',sys.argv[1]+'2013.out', ..., sys.argv[1]+'2018.out' ]

RIMUOVERE ELEMENTO:
url_out=[sys.argv[1]+'2013.out',sys.argv[1]+'2014.out']
```

Quindi eseguiamo il file impl url che prende come parametro il percorso della cartella nella quale è presente il file interm file generato dall'esecuzione citata precedentemente, la presenza di questa file sarà indispensabile per

l'esecuzione dello script. Terminata l'esecuzione saranno stati generati un numero di file pari al numero di elementi della lista url out citata in precedenza. Per ognuno di questi file generati ci sarà bisogno di eseguire engineplus.py che si occuperà di raccogliere tutti i dati relativi a quell'anno e raggrupparli in un foglio di calcolo da affiancare al foglio di calcolo ottenuto dal' esecuzione di search.py.

4.4 Progettazione script

La mia analisi è partita da uno studio manuale di alcuni repository che fanno uso di container docker, dopo aver visionato le prime pagine di risultati che github mi ha proposto, e aver fatto il dovuto backgroud sui container docker e sul loro impiego da parte degli sviluppatori,mi sono concentrato su un repository, in cerca di informazioni interessanti da analizzare per la mia ricerca. In genere un repository che usa docker ha almeno un dockerfile nel quale è indicato dove reperire l'immagine alla quale si riferisce e altri comandi da eseguire una volta reperita l'immagine. In alcuni repository oltre ad esserci più dockerfile è presente anche un docker-compose che indica come comporre l'architettura dell'applicazione, basata su uno stack di container docker,assegnando ad ognuno una specifica parte(micro-servizio) per il raggiungimento di un servizio finale. Oltre a questi file non ho molto altro su cui basare le mie ricerche , quindi ho effettuato una ricerca con parola chiave "docker" ottenendo circa 200000 repository da analizzare come risultato. Successivamente mi sono preoccupato di capire se effettivamente la ricerca era sensata, nel senso che mi avrebbe dato solo repository che fanno uso in qualche modo di docker, così sempre con parola chiave docker sono andato a cercare solo nei campi nome , descrizione e readme di github, ottenendo circa 500000 repository come risultato, più del doppio dei risultati rispetto alla ricerca precedente, ma andandoli ad analizzare manualmente mi sono reso conto che molti di questi repository non facevano uso di docker né diretto né indiretto ,In figura 4.7 riporto la lista dei 20 repository analizzati

manualmente. Di questi 20 ben 10 repository non facevano uso di docker , la maggior parte dei repository di questa categoria erano progetti contenenti link,con una breve descrizione sul contenuto del link.Tali link erano riferiti ad articoli scientifici o pagine internet o altri progetti sulla piattaforma github. Avendo appurato che circala metà dei repository riguardanti questa ricerca potesse essere simile ai repository appena citati ho fatto un passo indietro. A questo punto ho eliminato dalla ricerca il campo readme e effettuato nuovamente la ricerca, i risultati ottenuti da questa ricerca sono circa 200000 come la prima ricerca, differiscono solo per 500 elementi circa . La ricerca che ho effettuato equivale quindi a cercare nei campi nome e descrizione, di tutti i repository presenti su github, la parola chiave docker ed avere come risultati circa 200000 repository da analizzare.

```
https://github.com/sindresorhus/awesome
https://github.com/jwasham/coding-interview-university
https://github.com/moby/moby
https://github.com/vinta/awesome-python
https://github.com/axios/axios
https://github.com/nvbn/thefuck
https://github.com/kubernetes/kubernetes
https://github.com/getlantern/lantern
https://github.com/josephmisiti/awesome-machine-https://github.com/learning
https://github.com/typicode/json-server
https://github.com/vuejs/awesome-vue
https://github.com/justjavac/free-programming-books-zh_CN
https://github.com/avelino/awesome-go
https://github.com/firehol/netdata
https://github.com/creationix/nvm
https://github.com/Unitech/pm2
https://github.com/gogits/gogs
https://github.com/sahat/hackathon-starter
https://github.com/lord/slate
https://github.com/serverless/serverless
```

Figura 4.7: lista repository analizzati

4.4.1 Scelta informazioni da analizzare

Una volta indirizzata la ricerca mi sono concentrato sulle tante informazioni che un repository che usa container docker mostra, in cerca di quelle più interessanti per me da andare a studiare. In primo luogo ho analizzato solo le statistiche presenti nella lista di elementi tipo quella mostrata nella sezione 4.1.2 , quindi estraendo come statistiche: linguaggio utilizzato nel repository e numero stelle del repository . Successivamente aggiungo alle statistiche da estrapolare dagli elementi della lista anche url della pagina html del repository,dal quale,analizzando la pagina html con una qualche tecnica, avrei potuto estrapolare altre statistiche interessanti quali : numero commit , data ultimo commit e numero di partecipanti al repository. Le stesse statistiche avrei potuto raccoglierle tramite una serie di query, in particolare con la api riguardante i commit e quella riguardante i partecipanti ,una volta ricevute le risposte avrei dovuto elaborarle per estrarre i dati di mio interesse , totalizzando un totale di 2 query per ogni elemento più eventuale elaborazione della risposta da implementare. Ho invece deciso di non intraprendere questa strada per risparmiare un po' di tempo che in termini di 200000 elementi equivarrebbe a giorni. Per questo motivo ho scelto di usare xpath ed un approccio scraping per poter estrarre con l'uso di una sola query ad elemento tutti i dati interessanti , con la consapevolezza che un cambio di template da parte di github potrebbe compromettere la seconda parte della raccolta delle statistiche. Una volta effettuata la mia analisi mi sono reso conto che il dato sul numero dei partecipanti ad ogni repository era poco significativo in quanto quasi sempre 1 , portando la media per ogni anno ad 1, quindi ho deciso di oscurare questo dato nella mia analisi , ma viene comunque registrato dall'algoritmo nella tabella dei risultati mostrata nelle sezioni seguenti. In figura 4.8 mostro un esempio di pagina html dalla quale sono andato successivamente ad estrapolare le informazioni di mio interesse in modo automatico.

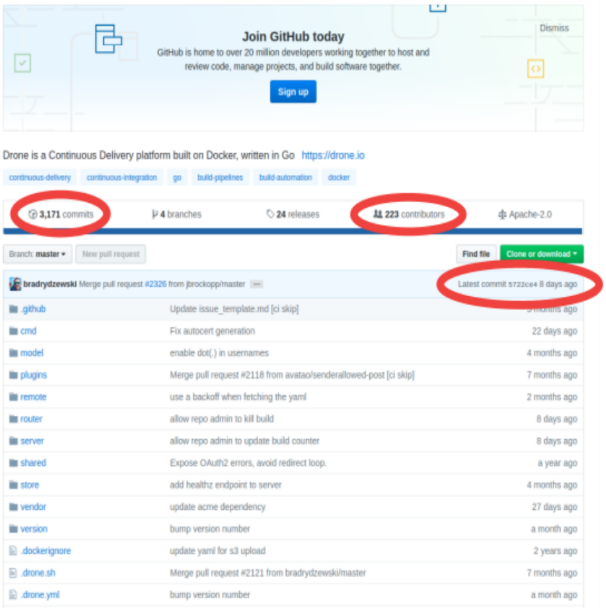

Figura 4.8: Esempio di informazioni interessanti di un progetto

4.4.2 Studio per una ricerca automatica

Per iniziare il lavoro di programmazione ho per prima cosa creato un repository vuoto su bitbucket, una piattaforma gemella di github che permette di rendere il proprio repository privato senza l'obbligo di sottoscrivere un abbonamento, successivamente ho portato il repository anche su github . Il passo successivo è stato progettare l'algoritmo che mi avrebbe permesso di automatizzare la ricerca, riguardiamo l' api usata in 4.1.2 e la risposta che github fornisce; come compito quindi devo raccogliere per ogni repository che usa docker 3 informazioni: url della pagina html, linguaggio usato e numero stelle; per far ciò però ho bisogno di una api che mi dia una lista di elementi. Come spiegato nella sezione 4.1.3 le limitazioni che github impone sui risultati sono molte, in particolare github può fornire massimo 1000 risultati per ogni query/api , quindi la stessa query se dovesse chiedere : "tutti i repository che usano docker creati nel 2014" , il sistema di risposta github risponderebbe : "sono 50.000 ecco i primi 1000" , da qui sorge il primo problema : c'è bisogno di tante query e quindi di un compositore automatico,

possibilmente personalizzabile , di query . In particolare la mia ricerca vuole tutti i risultati, nessuno escluso, quindi un approccio naìve : comporre una query per ogni giorno dell'anno, da un anno d'inizio ad uno di fine , non ha successo dal momento in cui ci sono giorni in cui sono state create diverse migliaia di repository che fanno uso di docker, e il massimo di risultati per una query è 1000 come detto in precedenza. Ogni giorno dell'anno può avere diverse pagine di risultati ,questa informazione la si può estrapolare grazie ad una query leggermente diversa da quella mostrata in 4.1.2, la mostro di seguito :

```
$ curl -I
>_
>'https://api.github.com/search/repositories?=docker+created%3A2015-01-02'
```

Con l'opzione -I si richiede al sistema solo l'informazione relativa all'intero blocco di risultati, quindi con la query sopra si richiede l'informazione relativa al blocco di risultati della api tra apici ,risultati relativi ai repository che usano docker creati in data '2015-01-02', in figura 4.9 mostro la risposta che si ottiene da questa query.

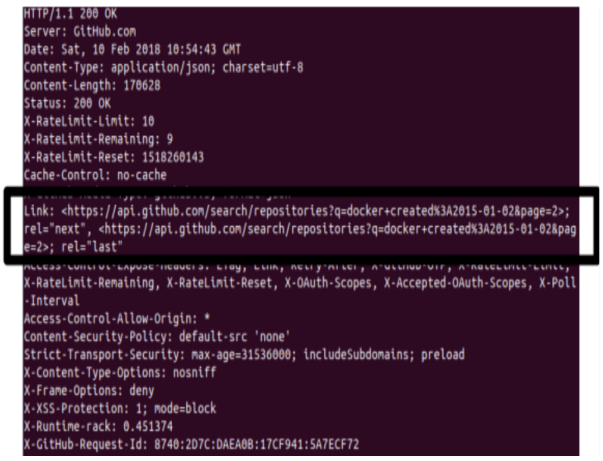

Figura 4.9: Esempio di risposta ad api con campo -I

La risposta oltre a informazioni relative alla dimensione della lista dei risultati, al rate-limit citato nella sezione 4.1.3, al tempo relativo affinché quest'ultimo si resetti e varie informazioni sullo stato del server , contiene l'informazione che ho usato per rendere più dettagliata possibile la mia ricerca. Nel riquadro evidenziato della figura 4.9 è contenuto il campo link il quale indica ,nella prima parte , il link alla pagina successiva alla prima,in genere la seconda pagina di risultati, seguita dalla stringa ‘ rel=”next” ’ che sta ad indicare la relazione che c'è tra la prima pagina è questa,in questo caso next quindi questa pagina è la successiva rispetto alla prima. Nella seconda parte del campo link è presente il link relativo all'ultima pagina dei risultati, seguito dalla stringa ‘ rel=”last” ’ che come spora indica la relazione tra la prima pagina di risultati e questa pagina, in questo caso la pagina 2 è sia la successiva alla prima che l'ultima pagina di risultati; ciò significa che in data ‘2015-01-02’ sono stati creati dai 1001 ai 2000 repository. Nel caso in cui ci sia una sola pagina di risultati relativi a quel giorno, il campo link sarà formato da una sola parte , la quale conterrà il link alla prima pagina, seguito dalla stringa ‘ rel=”last” ’.

Nel capitolo successivo vado a mostrare i risultati raccolti nella mia ricerca sui container docker, divisi per anno e poi dei risultati totali per avere un quadro più ampio riguardo l'uso e la diffusione di questa tecnologia. In particolare mi soffermo sui linguaggi utilizzati, le stelle e i commit. Termino il capitolo con la presentazione di alcuni repository che fanno un uso interessante di docker che andrò successivamente ad evidenziare e categorizzare.

Capitolo 5

Analisi

Nel seguente capitolo andrò a descrivere l'analisi vera e propria che ho effettuato. L'analisi consisterà nell'esporre alcuni grafici riguardanti le statistiche raccolte in modo automatico ed alcune tabelle contenenti i dati che popolano i grafici stessi. Nella prima sezione vado ad esporre i dati relativi ai linguaggi utilizzati nei progetti che fanno uso di container docker, in questa sezione, come anche nelle successive, esporrò prima un quadro generale dei dati , per poi andarli ad analizzare più nel dettaglio. I dati saranno organizzati sempre secondo una distribuzione temporale, o in mesi o in anni. Nella seconda sezione mostro i grafici e le tabelle relative ai feedback ricevuti da parte di ogni progetto dalla comunità, per far ciò andrò a studiare le stelle di ogni progetto. Per quanto riguarda la sezione 5.3 ,come le precedenti, mostra l'andamento temporale delle attività legate ad un repository, studiando i commit di ogni progetto. Per chiudere il capitolo, e l'analisi , vado ad analizzare più nel dettaglio un certo numero di progetti che usano docker, per capire il motivo che spinge gli utenti ad usare questa tecnologia ed in quale la tecnologia aiuta l'utente a raggiungere un obbiettivo . Questa analisi dettagliata è consistita in una visione e studio di circa 50 progetti che fanno uso di docker, con lo scopo di capire l'ausilio che docker da al progetto in analisi . Con l'ausilio di questa analisi manuale vado poi a categorizzare l'uso che in genere gli utenti fanno della tecnologia docker , ogni categoriz-

zazione sarà anticipata dalla presentazione di un repository tipo di quella categoria. Prima di entrare nel dettaglio dell'analisi vado ad introdurre un po' di nozioni riguardanti l'analisi stessa . L'analisi si è basata sull'analisi di un file JSON contenente circa 160000 risultati, che sarà il mio dataset. Per generare questo dataset mi servo del repository citato nella sezione 4.3.4 realizzato da me , il quale impiega per popolare il dataset circa 3sec a query, quindi in particolare nell'analisi che sto presentando la computazione è durata circa 5 giorni . Nelle sezioni successive andrò a mostrare i dati ,raccolti dall'algoritmo appena citato in un foglio di calcolo, organizzati in grafici e tabelle .

5.1 Uso linguaggi

In questa sezione vado a mostrare alcuni grafici relativi ai risultati dell'analisi effettuata, in particolare mi concentro sui linguaggi di programmazione utilizzati e sulla loro distribuzione nel tempo. In tutti i grafici ho deciso di oscurare alcuni linguaggi per rendere lo stesso grafico più leggibile. Tra questi Shell, in quanto presente in tutti i repository analizzati , anche i campi others,riguardante tutti i linguaggi non analizzati, e null , riguardante i repository con il campo riguardante il linguaggio uguale a null. Sempre per rendere più leggibili i grafici in alcuni casi ho preferito non riportare tutti i linguaggi analizzati in quanto molti hanno andamento quasi uguale tra loro come python e javascript o ruby ,makefile e go . Come primo grafico riporto un quadro generale.Questo grafico è riassuntivo dell'intera situazione riguardante i linguaggi di programmazione utilizzati, dal gennaio 2013 al maggio 2017, da questa figura si evince il fatto che la crescita sia netta, come detto prima ci sono linguaggi che vanno a braccetto come python e javascript, che sono anche i linguaggi più usati, probabilmente per la loro versatilità e la loro fama.

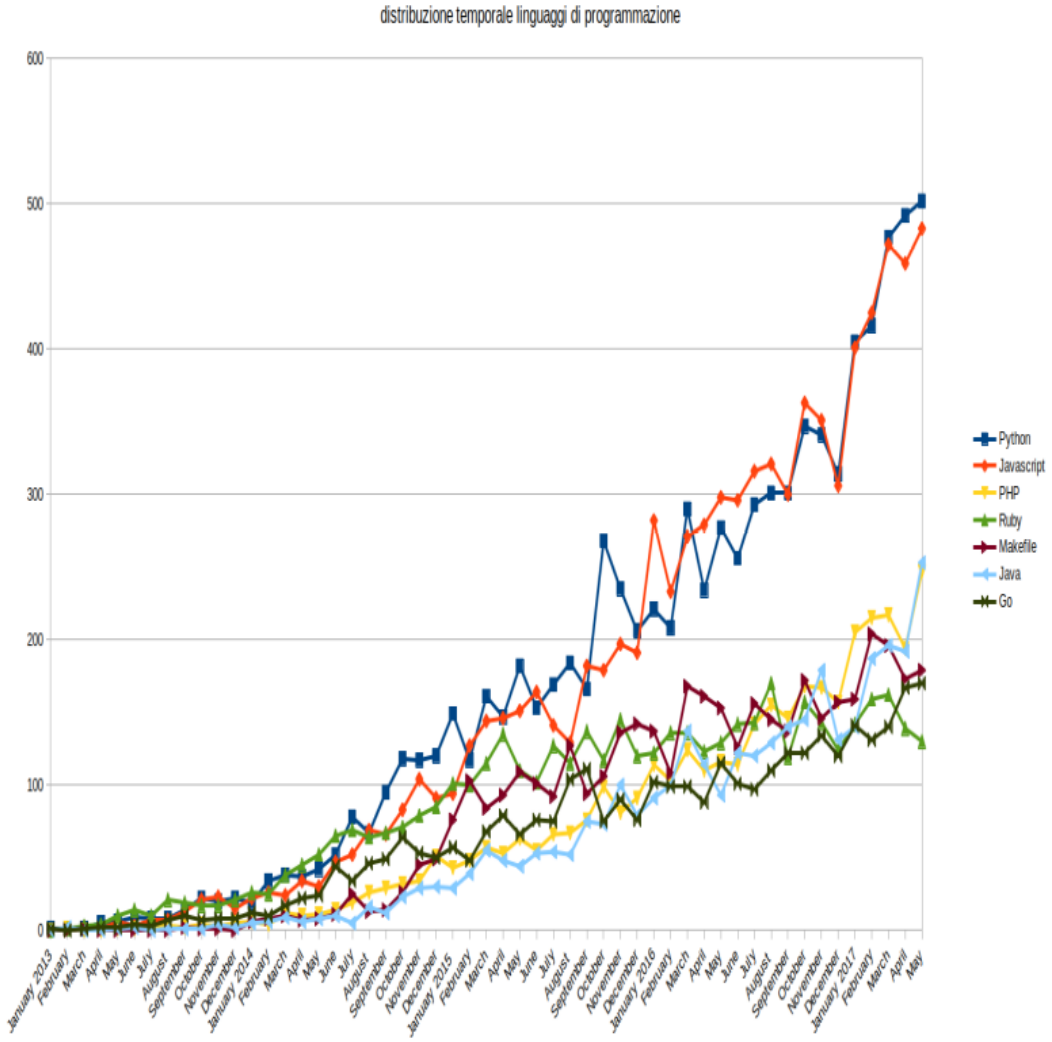

Figura 5.1: diffusione uso linguaggi

Stando alla figura 5.1 riportata sopra notiamo che si passa da gennaio 2013 nel quale i repository creati che usano docker sono appena 6 , al gennaio 2017 nel quale, contando i 2000 repository creati in linguaggio shell, i repository creati che usano docker sono circa 7000 . In figura 5.2 riporto la tabella contenente i numeri totali risultanti dal grafico :

LINGUAGGIO	NUMERO REPOSITORY	USO
Shell	53462	32,75%
Python	8580	5,25%
Javascript	8266	5,06%
PHP	3688	2,25%
Ruby	4486	2,74%
Makefile	4073	2,49%
Java	3268	2,00%
Go	3350	2,05%
Others	14320	8,77%
Null	59731	36,59%
Totale	163224	

Figura 5.2: dati che popolano grafico di figura 5.1

Nella figura 5.2 è mostrato in termini percentuali l'uso totale che nel tempo è stato fatto dei vari linguaggi analizzati, come detto il linguaggio Shell è quello più usato, dal momento in cui il Dockerfile è un file contenente comandi Shell ciò era previsto, più interessante è osservare i due blocchi formatisi intorno al 5 percento , Python e JavaScript , e intorno al 2 percento , PHP , Ruby,Makefile,Java , Go . Così mi sono focalizzato sui linguaggi più interessanti e ho diviso i dati per anni, così da ottenere un grafico per ogni anno e avere un quadro più dettagliato dell'andamento temporale riguardante lo sviluppo di ogni linguaggio , per avere un quadro ancora più chiaro userò

un altro tipo di grafico rispetto a quello mostrato sopra. In figura 5.3 mostro i grafici risultanti dai filtri appena elencati:

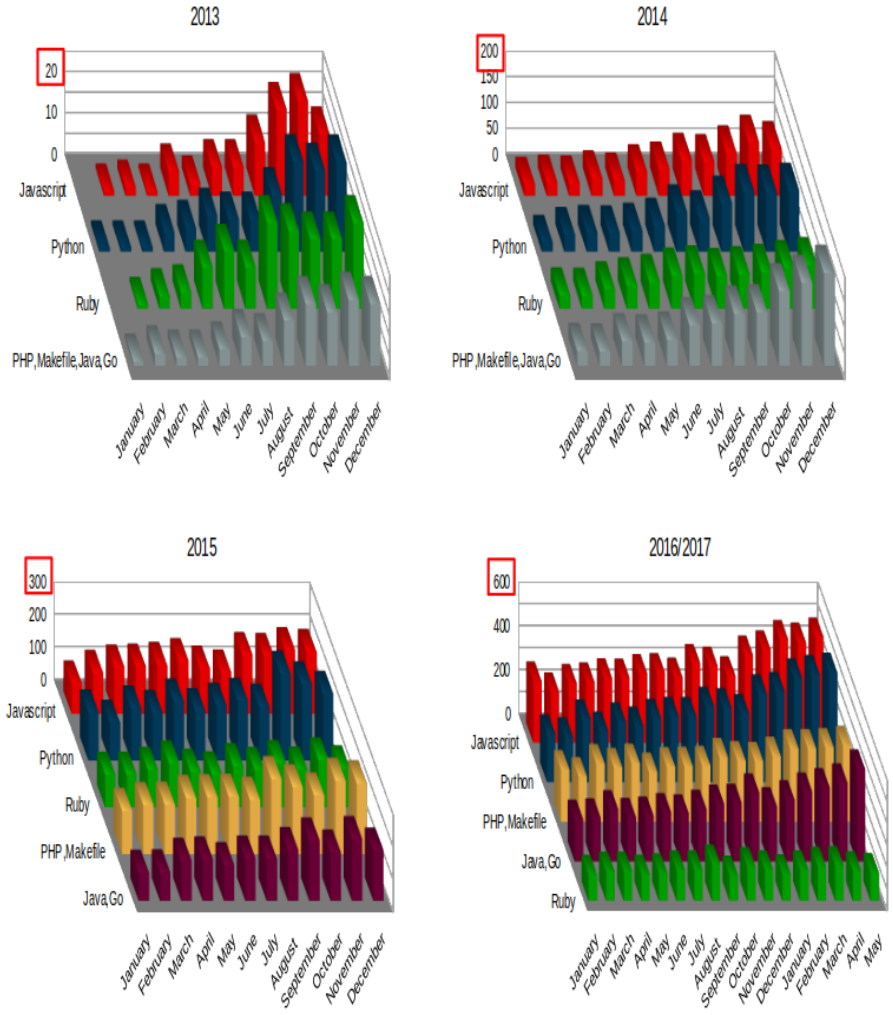

Figura 5.3: uso linguaggi dal 2013 al 2017

In figura 5.3 ho voluto riassumere gli andamenti riguardanti l'uso dei linguaggi dal 2013 al 2017, come premessa bisogna rendersi conto che per ogni anno la scala del grafico, che ho evidenziato con un riquadro rosso , cambia

abbastanza vertiginosamente, se nel 2013 abbiamo una limite superiore di 25 elementi , l'anno dopo questo limite è aumentato di ben 5 volte (120) , negli anni successivi gli aumenti ci sono stati ma non così vertiginosi. Notiamo che dal 2014 al 2015 il limite è circa raddoppiato (300) , negli anni successivi le scale aumentano in maniera più lineare andando da 300 a 400, dal 2015 al 2016, e da 400 a 600 ,dal 2016 al maggio 2017 in particolare nel 2017 probabilmente si riscontrerà , nelle analisi future , un altro aumento vertiginoso in quanto in metà anno il limite superiore ha già superato la previsione rispetto al precedente anno. Per quanto riguarda l'uso dei linguaggi , notiamo che nel 2013 , oltre ai previsti python e javascript con picchi di rispettivamente 20 progetti e 15 progetti al mese abbiamo un alto uso di ruby,picchi di circa 20 progetti al mese , Restano ai margini go,java,makefile e php con picchi inferiori a 5 progetti mensili. Nel 2014 si riscontra un aumento sostanziale dei picchi, si passa da picchi di 20 progetti al mese dell'anno precedente a picchi di circa 120 progetti , ad esempio per quanto riguarda python,ruby e javascript invece sembrano viaggiare di pari passo ma una spanna sotto python arrivando a picchi di circa 80 repository mensili, per quanto riguarda php e makefile java e go restano abbastanza ai margini, con una piccola eccezione per go che sembra essere usato più degli altri. Per quanto riguarda gli anni 2015 e 2016 sembrano essersi stabilizzati gli usi dei linguaggi nonché la diffusione degli stessi in progetti che fanno uso di docker. Se nel 2015 python è nettamente il linguaggio più usato con picchi di 250 progetti mensili e a seguirlo ci sono javascript e ruby. Anche php e makefile insieme a go e java aumentano la loro diffusione arrivando a picchi di 200 progetti al mese, nel 2016 l'uso di python e javascript si allinea quasi con picchi di 400 repository mensili e l'uso di tutti gli altri linguaggi si stabilizza intorno a picchi di 200 repository mensili.

5.2 Feedback per repository

In questa sezione vado a mostrare altri interessanti dati che ho raccolto nel corso della mia analisi riguardo alle stelle che un progetto può avere, le stelle stanno a rappresentare il feedback che quel progetto ha avuto dalla comunità di github. Anche in questa sezione come la precedente la maggior parte dei dati sarà mostrata sotto forma di grafico per avere un visione più chiara sulle statistiche generate dall'elaborazione dei dati raccolti. Come nella sezione precedente anche in questa ho preferito collassare alcuni dati per rendere i grafici più leggibili, tali dati però sono comunque stati presi in considerazione nelle conclusioni. Di seguito riporto un quadro generale riguardo alla situazione della media stelle repository mensile :

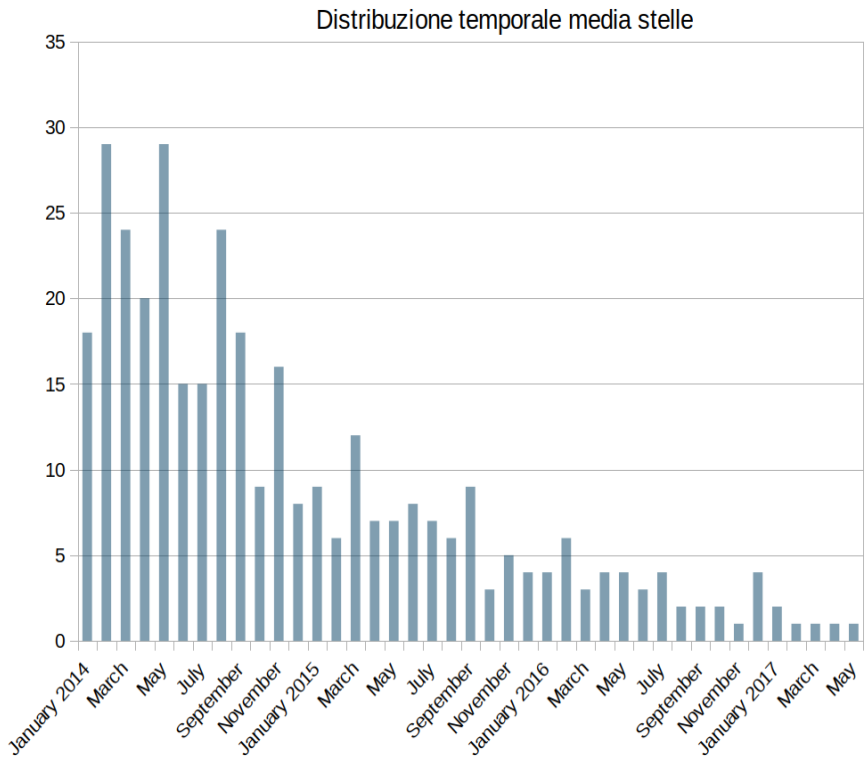

Figura 5.4: calo feedback community

Nella figura 5.4 è rappresentato un grafico riassuntivo,ho voluto oscurare ,per il momento ,il 2013 in quanto è più opportuno fare un discorso a parte per quell'anno. In questo grafico si nota un' inaspettata decrescita dell'andamento della media stelle per repository. In un primo momento potrebbe sembrare una decrescita inspiegata dato quanto detto nella sezione precedente in quanto a diffusione. Facendo un' analisi più dettagliata si può però arrivare alla conclusione che la decrescita del feedback della comunità è prevedibile, in quanto un sempre maggiore uso e sviluppo della tecnologia ha portato a un aumento vertiginoso del numero di repositry che usano docker, come visto nella sezione precedente. Questo vertiginoso aumento del numero di repository che usano docker ha , per forza di cose , portato ad una ripartizione dei feedback della comunità, nel senso che se inizialmente i repository che usavano docker erano pochi e l'attenzione della comunità si concentrava su quei pochi , portando a questi pochi progetti grandissimi feedback, con l'aumento dei progetti che fanno uso della tecnologia non è più alla portata della comunità riuscire a dare un feedback su ogni progetto così quello che si nota è una vera e propria ripartizione dei feedback . Di seguito riporto una tabella che riassume ancora i dati riportati in figura 5.4 per anno :

ANNO	NUMERO REPOSITORY	MEDIA STELLE
2013	1838	739,33
2014	17864	19,25
2015	41661	7,33
2016/2017	105515	3,75

Figura 5.5: dati che popolano grafico di figura 5.4

Come si nota nella figura 5.5 ad un aumentare vertiginoso del numero dei progetti che usano docker è seguito un altrettanto vertiginoso diminuire della media di stelle per progetto. Come detto prima è una conseguenza abbastanza prevedibile , è però interessante osservare come nel 2013 la media stelle sia davvero alta circa 700 per un numero relativamente piccolo di progetti circa 2000 e appena l'anno dopo il numero di progetti si sia moltiplicato per dieci, circa 18000 progetti , e la media stelle si sia divisa addirittura di 30 volte . Negli anni seguenti invece si ha una crescita del numero di progetti e una decrescita della media stelle costante , in particolare se nel 2014 i progetti che utilizzano docker sono circa 20000, l'anno dopo sono il doppio circa 40000 e nel 2016 sono circa 100000, ancora il doppio .Stesso discorso va fatto per la decrescita della media di stelle per progetto che se nel 2014 è a 19,25 nell'anno seguente e si è poco più che dimezzata arrivando a circa 7 e nel 2016 sarà circa 3,50 , ancora la metà dell'anno prima . Di seguito vado a mostrare nello specifico le statistiche ottenute dall'analisi dei vari anni organizzate in grafici tipo quello di inizio sezione :

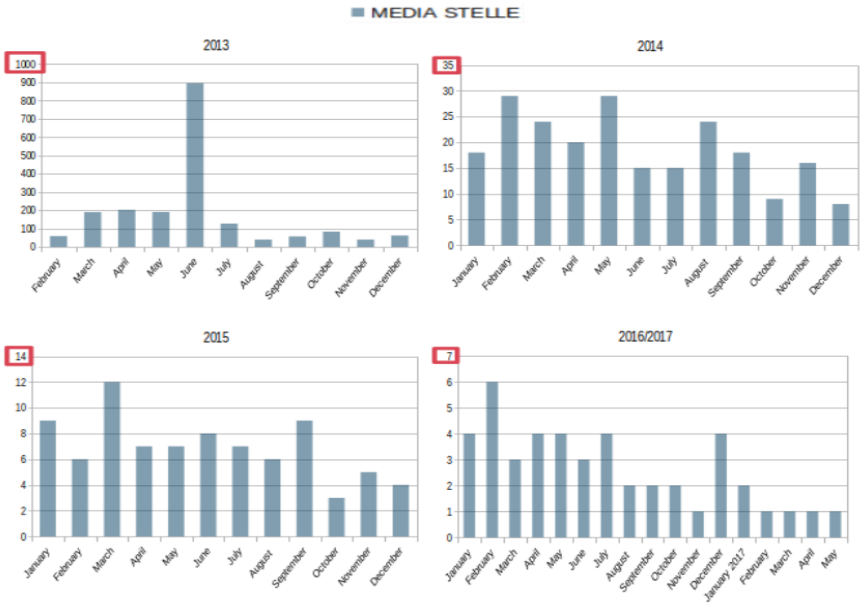

Figura 5.6: andamento media stelle dal 2013 al 2017

In figura 5.6 mostro i grafici che affrontano più nel dettaglio i dati raccolti nel corso dell'analisi, partendo dal 2013, l'anno in cui nasce la tecnologia, sembra che la media stelle per repository sia destinata a essere molto alta. Se si osserva bene nel primo grafico è stato oscurato il mese di gennaio, in quanto rendeva poco leggibile il grafico con la sua media stelle a circa 7000, di contro un numero di repository pari appena a 6, tra cui abbiamo il repository " moby/moby ", molto diffuso con ben 48000 stelle e il repository " progrium/buildstep " anch'esso molto diffuso con circa 1000 stelle. Come detto prima la decrescita della media di stelle per repository è sostanziale, ciò è sottolineato dai riquadri rossi nella figura sopra , i quali evidenziano un cambiamento vertiginoso di scala passando da un anno all'altro, un po' come nella sezione precedente 5.1. Se nel 2013 il limite superiore della media di stelle per repository è circa 1000 mentre nell'anno seguente è circa 30 volte inferiore, quindi pari a 35, nel 2015 è la metà dell'anno precedente, circa 15, e nel 2016 è ancora dimezzato a circa 7. Un altro dato interessante è il restringimento della media di stelle nell'inizio del 2017 che sembra avere un andamento costante, un terzo dei mesi dell'anno hanno come media una stella per repository, ciò fa presagire che con il passare del tempo questo media si avvicini sempre di più a 0 o comunque rimanga costante su una sola stella per repository.

5.3 Attività legate ai repository

Nella sezione seguente vado a mostrare i dati relativi ai commit che i progetti analizzati hanno subito, un commit è inteso come modifica a un qualche file del progetto da parte di uno dei partecipanti al progetto stesso. I dati relativi ai commit permettono di osservare molte statistiche interessanti, io ad esempio ho voluto usare il numero di commit di ogni progetto per calcolare una media del numero di commit, sia per repository che per giorno. Inoltre ho trovato interessante il dato relativo alla data dell'ultimo commit,

che ho usato per determinare se un repository fosse attivo o non attivo, fissando come data di attività il 1-1-2017, se un progetto non ha un commit più recente di questa data è stato considerato non attivo. Non ho tenuto conto nella mia statistica del fatto che un progetto può essere creato con il primo commit e poi non più usato. In figura 5.7 mostro un grafico riassuntivo di tutta la situazione dal 2013 al 2017 riguardante la media commit per repository e media commit per giorno:

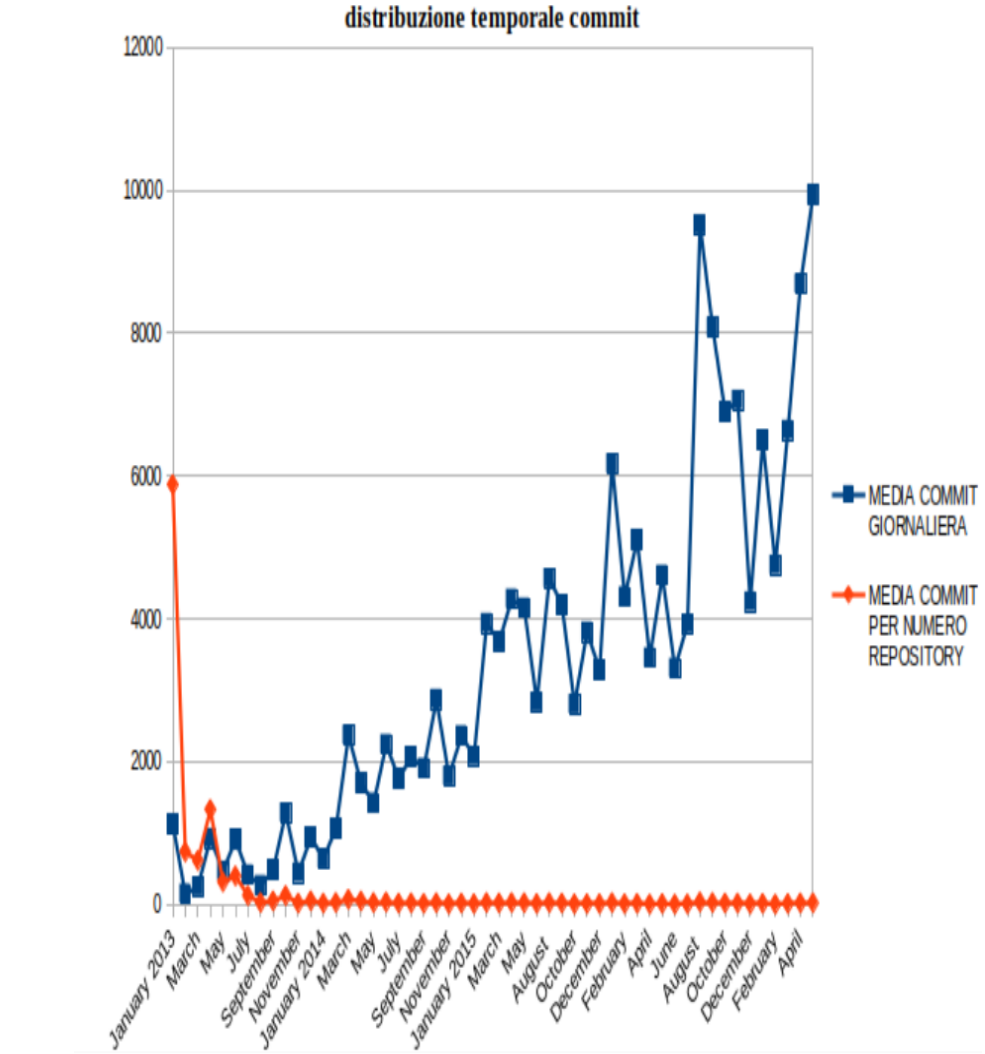

Figura 5.7: andamento attività per repository

Dal grafico riassuntivo si nota subito l'andamento opposto che anno le due medie calcolate, se da un lato la media commit per giorno cresce a vista d'occhio dal giorno 0 ad oggi , per quanto riguarda la media dei commit confrontato con il numero dei repository creati, come nella sezione precedente , l'andamento è costante su 1 , discorso a parte va fatto per l'anno 2013 che come già detto ha portato a degli squilibri in questa analisi in quanto a differenza degli altri anni successivi a questo il numero dei repository creati è rilevante. Come già spiegato la enorme diffusione della tecnologia ha portato un aumento vertiginoso del numero dei repository che ne fanno uso, rendendo le medie riferite a questo dato poco interessanti . Di seguito mostro i dati relativi alle medie annuali :

ANNO	NUMERO REPOSITORY	MEDIA COMMIT GIORNALIERA	MEDIA COMMIT PER NUMERO REPOSITORY
2013	1838	609,25	5,34
2014	17864	1861,75	0,02
2015	41661	4997	0,01
2016/2017	105515	6074,29	0,004

Figura 5.8: dati che popolano grafico di figura 5.7

Dalla tabella di figura 5.8 si nota quanto già detto, cioè all'aumentare del numero dei progetti che usano docker il numero di commit per progetto scende, ciò probabilmente è dovuto dal fatto che molti hanno solo provato ad usare la tecnologia, ma pochi hanno poi continuato a sviluppare usando docker . Dall'altro dato emerge però che con il passare degli anni il numero di commit giornaliero sembra essere aumentato quindi chi ha iniziato a sviluppare con la tecnologia docker ha poi continuato su questa strada . Come

nella sezione precedente il 2013 è un anno particolare per docker in quanto coincide con la sua nascita , questo è l'unico anno nel quale possiamo dire che ogni progetto che usa docker ha in media più di 5 commit , per gli anni successivi ci ritroviamo con una decrescita costante che tende a zero. Di seguito vado a mostrare nel dettaglio i dati relativi all'andamento dei vari anni , nei grafici seguenti ho preferito dividere le due medie in due grafici così da riuscire a visualizzare meglio il quadro, tenendo sempre un occhio di riguardo per la scala dei vari grafici :

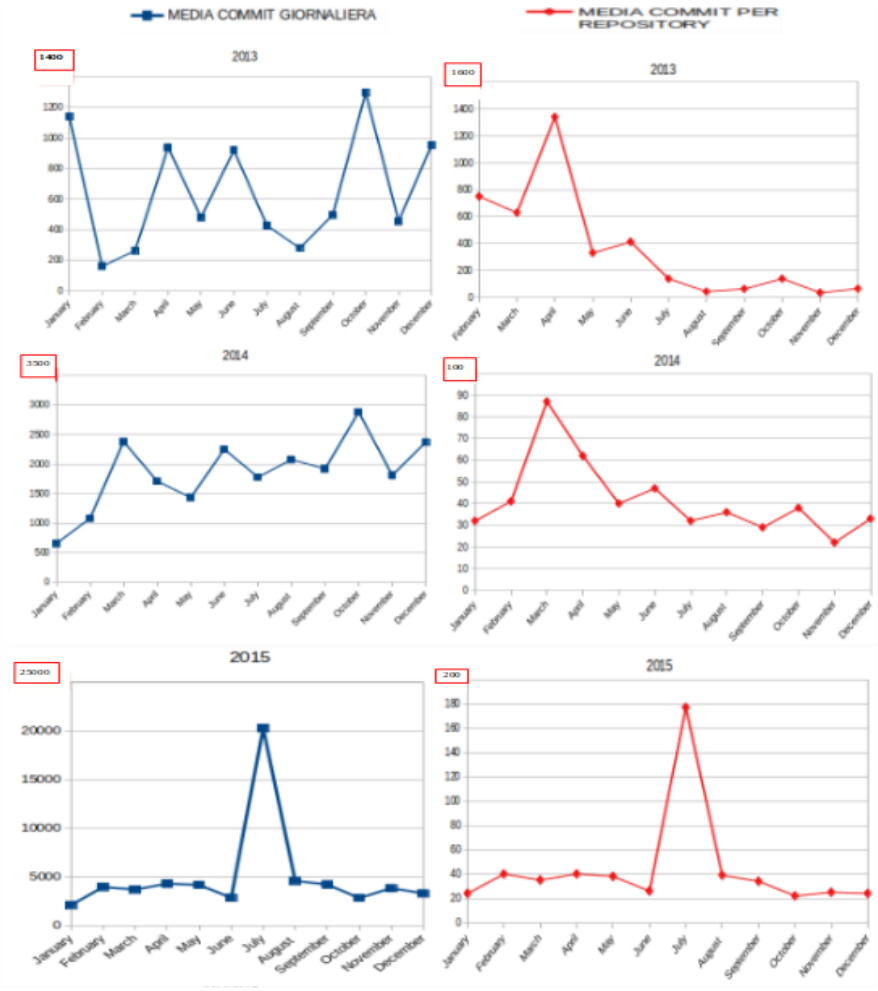

Figura 5.9: media commit dal 2013 al 2015

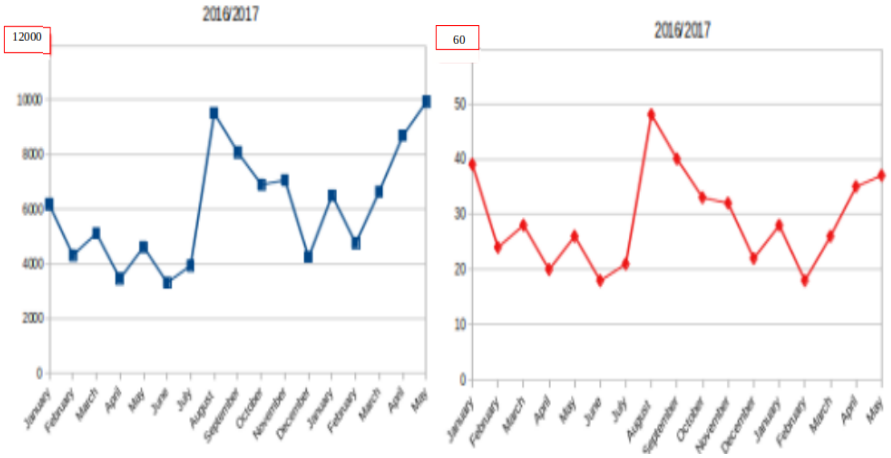

Figura 5.10: media commit 2016/2017

Prima di discutere riguardo il contenuto dei grafici di figura 5.9 e 5.10 voglio far notare che nel primo grafico riguardante la media commit per repository ho dovuto oscurare il mese di gennaio per rendere il grafico leggibile, in quanto il suddetto mese ha una media commit per repository pari a circa 6000 , molto sbilanciata rispetto al resto dei dati. Per quanto riguarda i grafici sopra andrei subito a discutere il fatto che nel 2013 le due medie crescono di pari passo con un limite superiore per entrambe di circa 1500 , anche se questo discorso non è già più praticabile nella seconda metà del 2013 nel quale si riscontra un forte calo. Nel 2014 la situazione sembra cambiare nettamente con una media commit giornaliera in crescita con picchi di 3000 commit giornalieri e una media commit per repository in netto calo con media comprese tra 20 e 90 commit per repository . Negli anni successivi si riscontra una particolarità, cioè le duè medie anche se limitate da estremi molto diversi hanno un andamento quasi uguale, ciò si riscontra particolarmente nel 2015 dove i due grafici rappresentano praticamente lo stesso andamento contro una diversità di estremi netta, si parla di 20000 per i commit giornalieri e di appena 200 per i commit di ciascun repository. Va però aperta una parentesi per quanto riguarda la media commit giornalieri riguardante il 2015

in quanto abbiamo picchi massimi sotto i 5000 commit giornalieri , ma nel mese di luglio si riscontra un picco di 20000 commit, stessa cosa dicasi per i commit per repository riguardanti il 2015; nel mese di luglio si riscontra un picco di circa 180 commit per repository contro un andamento annuale che non supera i 40 commit per repository.

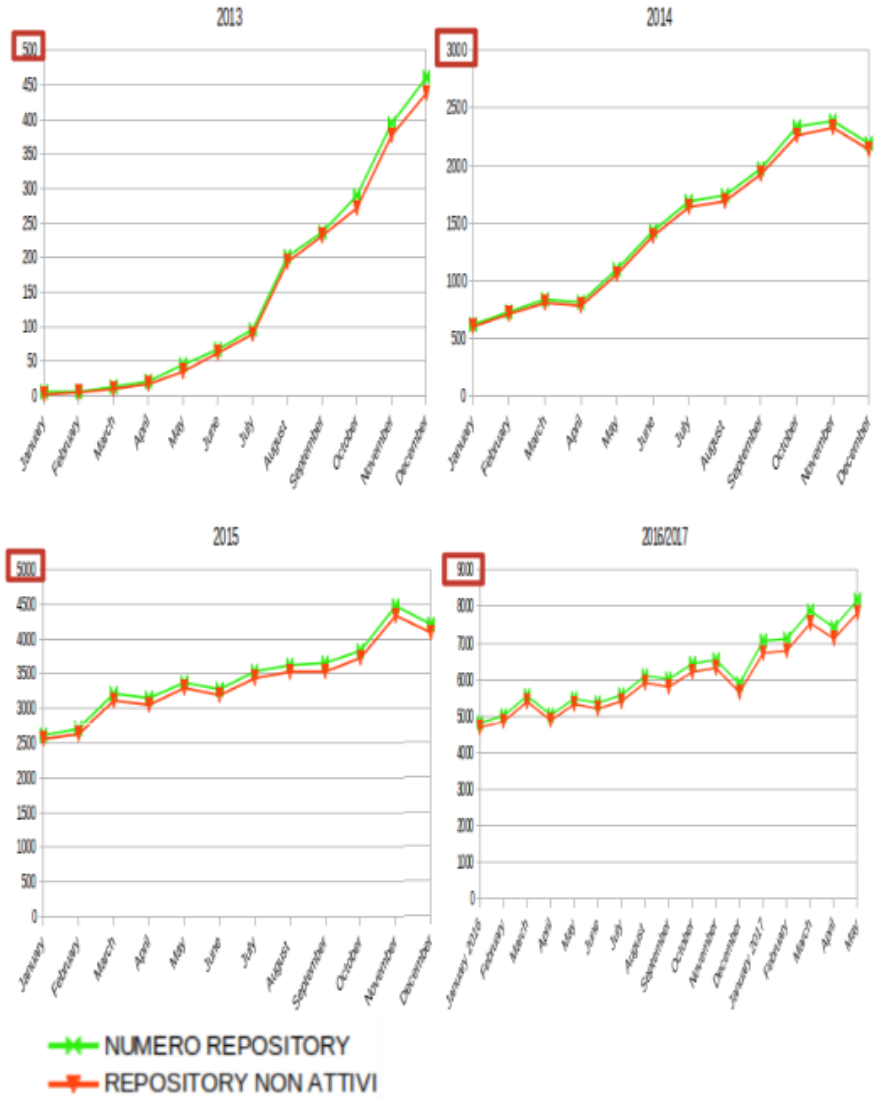

Figura 5.11: repository non più attivi dal 2013 al 2015

Anche nel 2016 e inizio 2017 la situazione sembra analoga all'anno precedente , dando uno sguaro ai graficidi figura 5.9 e 5.10 si nota un andamento molto simile per quanto riguarda i grafici, di contro i limiti delle due rappresentazioni sono ben diversi, da un lato abbiamo la media di commit giornalieri con picchi di 10000 commit e dall'altro lato abbiamo la media commit per repository con picchi di 50 commit. Oltre alla media sui commit ho cercato di analizzare anche la data dell'ultimo commit di ogni repository così da poter capire quanto dei repository creati continuano a funzionare ad oggi e ad essere supportati e quanti invece vengono abbandonati dopo il primo o i primi usi. I dati che sono emersi sono abbastanza previsti e rispecchiano un andamento che è solito su github , in figura 5.11 mostro dei grafici riassuntivi. Nei grafici in figura 5.11 ho raffigurato i dati relativi al numero di repository creati e al numero di repository non più attivi, dando uno sguardo si capisce perché io non abbia raffigurato i dati relativi ai repository attivi, infatti dalla figura si nota che il numero di repository creati e il numero di repository non più attivi coincide quasi per tutto l'andamento temporale. Questo dato resta abbastanza previsto in quanto su github sono molti i progetti iniziati solo per prova, e sono molte le prove prima di iniziare un progetto vero e proprio da portare avanti con commit regolari. Il dato che accomuna tutte le rilevazione fatte è quello relativo al limite superiore in aumento vertiginoso, qui ancor di più dato il fatto che è presente l'andamento dei repository creati, che come detto è in forte ascesa dal giorno zero. Si nota infatti come nel 2013 il limite superiore sia 500, nell' anno successivo sia 6 volte più grande, circa 3000. Negli anni successivi l'andamento è similare andando a toccare nel 2015 un limite di 5000 e nel 2016 un limite di ben 9000. In figura 5.12 ho comunque voluto riportare i dati relativi ai repository attivi e non attivi in una tabella per avere più chiara la situazione. I dati contenuti nella tabella sopra evidenziano quanto detto più del 90 percento dei progetti che usano docker non hanno avuto un commit recente quindi possono essere considerati non attivi. È interessante notare come nel 2013, come anche negli altri dati analizzati, ci siano dei dati poco equilibrati con il resto degli andamenti annuali. La soglia

di repitory attivi è molta alta rispetto agli anni successivi, infatti arriva a circa il 6 percento.

ANNO	NUMERO REPOSITORY	REPOSITORY ATTIVI	TASSO ATTIVI	REPOSITORY NON ATTIVI
2013	1838	105	5,71%	1733
2014	17864	520	2,91%	17334
2015	41661	1182	2,83%	40479
2016/2017	105515	3824	3,62%	101691

Figura 5.12: dati che popolano grafico di figura 5.11

Dal 2014 con l'aumento della diffusione della tecnologia si registra un drastico calo dei progetti che sono restati attivi ad oggi, intorno al 3 percento. Nel 2015 si registra un calo poco significativo del tasso di progetti rimasti attivi rispetto all'anno precedente, tenendo conto del fatto che il tasso non è quasi cambiato ma il numero di repository creati è più che raddoppiato si arriva alla conclusione che il numero di progetti attivi aumenta con l'aumentare dei progetti totali ma in maniera poco significativa rispetto al totale di progetti analizzati. Nel 2016 /2017 si registra un aumento sostanziale del tasso di progetti attivi, intorno al 3,6 percento, portando il numeroi di repository attivi a ben 4000 circa.

5.4 Categorizzazione dell'uso di container docker

Nella seguente sezione entreremo nel vivo dell'analisi, prima presento dei repository che usano docker che ho trovato interessanti per diversi aspetti che poi andrò ad enunciare. Insieme alla presentazione del repository presenterò anche la categorizzazione che ho fatto di quel repository. Per poter andare

a categorizzare l'uso che viene fatto della tecnologia docker mi sono servito di un'analisi sempre fatta da me manualmente su un ristretto campione di progetti che fanno uso di docker, come descritto nella sezione successiva, una volta svolta un'analisi dettagliata ho potuto distinguere diverse categorie dell'uso che viene fatto di docker e dei suoi container. Oltre alla categorizzazione riporto dei dati interessanti riguardo quest'ulteriore analisi svolta. Tali dati saranno organizzati in tabelle e riguarderanno la concentrazione di una o dell'altra categoria nel campione preso in considerazione di 58 elementi. Dopo aver terminato la fase di raccolta dati e stesura statistiche mi sono voluto concentrare sull'uso che gli utenti fanno dei container docker, che come visto nei capitoli precedenti può essere vario, ciò è dovuto alle grandi potenzialità di docker. Il passo successivo quindi è stato analizzare manualmente alcuni repository che usano i container docker per capire a che scopo lo fanno. I repository analizzati manualmente sono stati circa 50 e mi hanno permesso di trarre interessanti conclusioni, l'analisi si è basata su una lettura, spesso difficoltosa, dei file readme per capire lo scopo del repository in generale, e dei file tipicamente legati ai container docker tipo dockerfile o docker-compose per capire in che modo docker aiuta a raggiungere lo scopo. Di seguito riporto la lista dei repository che hanno rappresentato il mio campione:

`https://github.com/deis/deis`
`https://github.com/eclipse/che`
`https://github.com/sqshq/PiggyMetrics`
`https://github.com/tomav/docker-mailserver`
`https://github.com/remind101/empire`
`https://github.com/bcicen/ctop`
`https://github.com/NVIDIA/nvidia-docker`
`https://github.com/spotify/docker-gc`
`https://github.com/docker/distribution`
`https://github.com/gliderlabs/registrator`
`https://github.com/gliderlabs/logspout`
`https://github.com/docker/docker-py`
`https://github.com/containous/traefik`
`https://github.com/tootsuite/mastodon`
`https://github.com/reactioncommerce/reaction`

https://github.com/kanaka/mal
https://github.com/ory/hydra
https://github.com/PokemonGoF/PokemonGo-Bot
https://github.com/firehol/netdata
https://github.com/dutchcoders/transfer.sh
https://github.com/GoogleChrome/rendertron
https://github.com/jwilder/docker-gen
https://github.com/taskrabbit/elasticsearch-dump
https://github.com/pachyderm/pachyderm
https://github.com/JrCs/docker-letsencrypt-nginx-proxy-companion
https://github.com/kylemanna/docker-openvpn
https://github.com/apache/incubator-openwhisk
https://github.com/PipelineAI/pipeline
https://github.com/containerd/containerd
https://github.com/verdaccio/verdaccio
https://github.com/nosqlclient/nosqlclient
https://github.com/Chatie/wechaty
https://github.com/michaelgrosner/tribeca
https://github.com/kubernetes/kompose
https://github.com/moby/moby
https://github.com/laradock/laradock
https://github.com/boot2docker/boot2docker
https://github.com/mesosphere/marathon
https://github.com/docker/docker-bench-security
https://github.com/coreos/clair
https://raw.githubusercontent.com/githubsaturn/captainduckduck
https://github.com/tsuru/tsuru
https://github.com/floydhub/dl-docker
https://github.com/docker-library/official-images
https://github.com/jupyter/docker-stacks
https://github.com/goreleaser/goreleaser
https://github.com/api-platform/api-platform
https://github.com/nodejs/docker-node
https://github.com/phusion/passenger-docker
https://github.com/widuu/chinese_docker
https://github.com/docker/swarm
https://github.com/docker/compose
https://github.com/docker/kitematic

```
https://github.com/portainer/portainer
https://github.com/weaveworks/weave
https://github.com/ClusterHQ/flocker
https://github.com/coreos/flannel
https://github.com/weaveworks/scope
```

Per andare a categorizzare l'uso che viene fatto dei container docker ho provato a capire il motivo per il quale un progetto usa docker e quali vantaggi ne trae. Se da un lato ci sono moltissimi utenti che usano docker semplicemente per ovviare all' inferno delle dipendenze dall'altro c'è chi ne fa un uso più interessante come controllare altri container o ancor più interessante chi aggiunge nuove funzionalità alla tecnologia. Molti combinano i vari usi di docker entrando a far parte di più categorie contemporaneamente. Inizialmente ho distinto 2 categorie principali : quelli che usano docker così com'è (c1) e quelli che estendo le funzionalità della piattaforma(c2). Di seguito mostro quanti repository ho analizzato relativi a ciascuna categoria :

CATEGORIA	NUMERO REPOSITORY	PERCENTUALE
c1	51	88%
c2	7	12%
totale	58	

Figura 5.13: percentuale categorie in base al numero repository

Successivamente a questa prima distinzione mostrata dalla figura 5.13 ,mi sono reso conto che analizzando più dettagliatamente i progetti che usavano docker (c1) avrei potuto individuare un'ulteriore categorizzazione dell'uso dei docker, che andrò a descrivere dettagliatamente nelle sezioni successive. Ho quindi individuato le seguenti categorie: docker come soluzione alle dipendenze (c1a), docker come ambiente per lo sviluppo (c1b), gestore di docker (c1c), composizione di docker (c1d).In figura riporto questa ulteriore distinzione .

CATEGORIA	NUMERO REPOSITORY	PERCENTUALE
c1a	22	37,9%
c1b	16	27,5%
c1c	8	13,8%
c1d	5	8,6%
c2	7	12%
totale	58	

Figura 5.14: percentuale categorie dettagliate in base al numero repository

Come previsto la percentuale maggiore è quella relativo allo scopo più semplice per il quale docker è nato , o se vogliamo lo scopo nativo di docker, quello di essere un agile rimedio all'inferno delle dipendenze ,quasi allo stesso andamento troviamo l'uso di docker come ambiente di test, che poco si differenzia dalla prima categoria se non fosse per lo scopo diverso che le due categorie aiutano a raggiungere. Una percentuale più di nicchia se la aggiudicano le categorie gestore docker e composizione di docker , intorno al 10 percento dei progetti totali fanno parte di ciascuna categoria. Di seguito mostro dei repository analizzati per poi andare a categorizzarli nel dettaglio.

- **Soluzione alle dipendenze:** Il repository mastodon dell'utente tootsuite è un social-network server open-source , ha lo scopo di restituire il controllo della distribuzione dei contenuti all'utente[15] .In mastodon docker ha il compito di creare l'ambiente ideale. Questo repository è molto diffuso infatti possiede 12.108 stelle e 2250 fork, circa 5000 commit, di cui l'ultimo il 09-02-2018, e 451 partecipanti al progetto.Il repository è disponibile per la visione e lo studio al seguente url: 'https://github.com/tootsuite/mastodon' . Il repository mastodon appena descritto costituisce un repository tipo della prima categoria,e non solo, cioè docker come soluzione alle dipendenze. Viene creato un container contenente tutti i pacchetti necessari affinché l'applicazione funzioni ,questo è l'uso più comune che si fa dei container docker . Il vantaggio di usare docker per creare ambiente ideale c'è sia per l'utente

che vuole scaricare ed usare l'applicazione,in quanto non dovrà avere a che fare con nessun errore di incompatibilità, e sia per il produttore dell'applicazione che non si vedrà costretto a rispondere alle richieste di soluzione di problemi,spesso legati a incompatibilità.

- **Ambiente di test:** Il repository laradock dell'utente laradock è un ambiente di test per sviluppatori di applicazioni PHP[14] , come si capisce dal nome usano container docker , con lo scopo di creare l'ambiente ideale al test . Il repository è abbastanza diffuso date le sue 4800 stelle e 1500 fork , inoltre ha avuto 1500 commit ,di cui l'ultimo il 09-01-2018, e 166 utenti che hanno contribuito al progetto.Il repository laradock è disponibile al seguente url:' https://github.com/laradock/laradock'.La seconda categoria è docker come ambiente di test, riguardando il repository appena presentato si capisce che l'uso che si fa di docker sembra semplice, cioè viene usato per configurare un ambiente predefinito come descritto nella categoria precedente, la cosa interessante è il diverso scopo che ricade sul container docker, infatti in questo caso l'ambiente è preconfigurato per effettuare test , con i vantaggi ulteriori di avere un ambiente isolato e privo di overhead .

- **Gestore container:** Il repository Kitematic dell'utente docker è un'applicazione per gestione docker su qualsiasi sistema operativo, dal design pulito [16]. Di questa tipologia di repository ne sono presenti davvero molti in giro per github e sono una delle tipologie più interessanti nonostante non usino direttamente docker. Kitematic vanta 8400 stelle e ben 900 fork, per quanto riguarda i commit è a quota 2200, di cui l'ultimo il 25-01-2018 , e 70 partecipanti al progetto. Il repository e disponibile al seguente url : 'https://github.com/docker/kitematic' . Per la terza categoria riguardiamo il repository che è un gestore di container, una applicazione che in genere risiede in un container legandosi alla prima categoria, che comunica con i container presenti nel sistema. La comunicazione tra applicazione(o container se l'applicazione è

eseguita in un container) e gli altri container avviene tramite la rete , l'applicazione conosce gli indirizzi IP dei container, ciò le permette di comunicare con loro ed effettuare operazioni di gestione su di essi tipo :creazione container, cancellazione container, rinomina container, e altre.

- **Estensione docker:** Il repository nvidia docker dell' utente NVIDIA è un' utility per schede grafiche nvidia che permette di costruire e gestire uno o più docker sulla GPU [17]. Ho trovato molto interessante questo repository e altri di questa tipologia dal momento in cui aggiungono ai container una funzionalità , in questo caso quella di divenire compatibili con gpu nvidia. Anche questo repository vanta ben 500 stelle e 600 fork, inoltre ha avuto 270 aggiornamenti,di cui l'ultimo il 07-02-2018, da parte di 9 partecipanti al progetto.Il repository è disponibile all'indirizzo : 'https://github.com/NVIDIA/nvidia-docker' . Questa categoria si riferisce a repository come quello appena presentato,il quale permette di usare docker in modo diverso dall'originale, estende la tecnologia docker, in questo caso rendendola eseguibile su un hardware specifico,propietario, sul quale inizialmente docker non poteva essere eseguito. L'aggiunta di funzionalità alla tecnologia non può che portare ad uso e una diffusione sempre maggiore della stessa e in ambiti sempre più differenti tra loro. Come descritto nella tabella di inizio sezione la percentuale di progetti di questo tipo sembra essere piuttosto bassa rispetto al numero totale di progetti ma comunque una percentuale significativa .

- **Composizione docker:** Il repository che di eclipse è un IDE eclipse per sviluppatori, la particolarità di questo IDE e che ogni servizio offerto dall'applicazione risiede su container isolato dal resto dei componenti, i vari container comunicano tra loro attraverso una tecnica di orchestrazione per arrivare a realizzare il servizio offerto [18] . Il progetto presentato è molto diffuso, tant'è che vanta circa 4500

stelle e ben 800 fork , oltre ad essere diffuso e anche in continuo aggiornamento, ciò è testimoniato dai circa 5800 commit ,di cui l'ultimo il 07-02-2018 . Il repository è pubblico e disponibile all'indirizzo: 'https://github.com/eclipse/che'. Questa categoria è l'ultima affrontata ed è se vogliamo la più interessante e quella dal potenziale maggiore. Non si fa fatica a immaginare le enormi potenzialità di questa tecnologia usata in questo modo, cioè un 'applicazione composta da 1 o più servizi, ognuno di loro isolato e autonomo rispetto agli altri , con la possibilità di essere riusato in un altro contesto, senza nessuna limitazione sul numero dei servizi . Il modo in cui gli utenti realizzano questo tipo di applicazioni è molto semplice, in linea di massima si sviluppano i vari componenti separatamente ,se ne crea le immagini separatamente e come ultimo passo si costruisce il docker-compose, file di cui ho parlato nella sezione 3.3.5 , il quale si occuperà di orchestrare i vari container eseguiti .

Conclusioni

In questo capitolo andrò a tirare le somme sull'intera analisi svolta e evidenzierò le osservazioni più importanti che sono emerse durante la ricerca. Dall'analisi effettuata nel capitolo 5 emergono notevoli caratteristiche interessanti riguardo i docker e soprattutto riguardo l'uso che si fa di questa tecnologia. In primo luogo l'obbiettivo posto nel capitolo 1, di effettuare un' analisi completa sull'uso e la diffusione della tecnologia, è stato raggiunto. Il mio contributo in sintesi consiste nella realizzazione di un programma configurabile per eseguire un'analisi su repository github. Oltre all'analisi automatica ho effettuato un'analisi manuale su alcuni di questi repository che mi ha permesso di categorizzare l'uso che viene fatto di docker. Il programma da me realizzato non gestisce gli output, che vanno organizzati dall'utente in un unico foglio di calcolo. Inoltre non ho previsto un'entità che gestisca il flusso dei dati tra le componenti della mia architettura. Di seguito vado ad elencare le osservazioni più interessanti emerse dall'analisi effettuata:

- In sintesi la piattaforma docker ha una diffusione in continuo aumento, dovuta probabilmente alla versatilità della tecnologia alla base di docker e alla semplicità con cui la piattaforma rende utilizzabile tale tecnologia.

- Docker è affiancata a qualsiasi altra tecnologia, oltre al linguaggio shell, che sarà sempre presente in un repository che usa docker, possiamo trovare qualsiasi linguaggio in concomitanza con un container docker, da python e javascript, i più diffusi, a go o ruby; inoltre si sta molto diffondendo l'uso di linguaggi sempre diversi da affiancare a docker.

- La media stelle che ha un repository che usa docker è in calo dalla nascita della piattaforma, ciò è probabilmente dovuto alla grande diffusione della tecnologia che ha spostato la centralità dei riferimenti che la community aveva in alcuni repository verso decine se non centinaia di repository che svolgono lo stesso compito.

- Di pari passo alla grandissima diffusione della tecnologia viaggia la media commit giornaliera per repository che fanno uso di docker, in sintesi essa va da un minimo di 100 commit al giorno nel 2013 ad un massimo di 10000 commit al giorno, ciò evidenzia ancor di più la diffusione che sta avendo questa tecnologia.

- Quasi la totalità dei repository creati per usare docker vengono poi abbandonati e non diventano veri e propri progetti, ciò è evidenziato dalla statistica relativa ai repository attivi o non attivi, nonostante questo però il numero di progetti che restano attivi nel tempo è in evidente crescita.

- Per quanto riguarda l'usare docker come soluzione alle dipendenze ha i vantaggi di evitare all'utente l'operazione, spesso difficoltosa, di installare le dipendenze di cui in genere un'applicazione ha bisogno per funzionare e allo stesso tempo permette a più gente possibile di usare l'applicazione stessa.

- Per quanto riguarda l'usare docker come ambiente di test ha notevoli vantaggi tra i quali avere un ambiente isolato dal resto del sistema evitando che eventuali guasti nei test compromettano l'uso del sistema stesso.

- La categoria riguardante l'uso di docker come gestore di container può essere molto interessante da studiare in quanto, in sintesi, definisce una sorta di meta-container, il quale gestisce gli altri container in modo elementare cioè permette operazioni tipo creazione, cancellazione e rinominazione di container.

- La categoria riguardante le estensioni di docker è forse la più importante in merito alla diffusione della tecnologia su una così vasta scala di progetti, in quanto estende la compatibilità della tecnologia, ma non solo, infatti in questa categoria risiedendo anche quei progetti che estendono le funzionalità o le prestazioni di docker.

- Per quanto riguarda l'usare la composizione di container è senza dubbio la categoria più interessante da un punto di vista di studio e di potenzialità, in quanto consente di definire vari sevizi, isolati tra loro, ed un orchestratore (docker-compose) che si occuperà di orchestrare il lavoro dei vari servizi, i quali risiederanno ognuno in un container ad-hoc. Con il vantaggio in caso di guasti ad un container di non propagare il guasto in altri container. Con la potenzialità di definire applicazioni che forniscono un numero sempre maggiore di servizi, e inoltre la possibilità di riusare quelli stessi servizi in altre applicazioni.

Sarebbe opportuno aggiungere al programma una funzionalità che organizzi meglio le statistiche raccolte e che realizzi i grafici in modo automatico. Spero che il mio contributo venga valorizzato ed usato da chiunque sia interessato ad effettuare delle ricerche su github. Per il futuro potrebbe essere interessante approfondire l'analisi di alcune categorie, come le composizioni di docker o le estensioni di docker.

Bibliografia

[1] Chamberlain, R., & Schommer, J. (2014). *Using Docker to support reproducible research.* DOI: https://doi. org/10.6084/m9. figshare, 1101910, 44.

[2] Merkel, D. (2014). *Docker: lightweight linux containers for consistent development and deployment.* Internet post, Linux Journal, 2014(239), 2.

[3] Internet Post, *L'ecosistema docker dal 2013 ad oggi,* 2017, http://www.internetpost.it/docker-ecosistema-2013-oggi/, 11 aprile 2017.

[4] Boettiger, C. (2015). *An introduction to Docker for reproducible research.* ACM SIGOPS Operating Systems Review,49(1), 71-79.

[5] Wikipedia, *LXC,* 2015, https://it.wikipedia.org/wiki/LXC, 8 dicembre 2017.

[6] Uniroma, *Docker,* 2017, http://cabibbo.dia.uniroma3.it/asw/pdf/asw650-docker.pdf, 18 maggio 2017.

[7] Wikipedia, *Git,* 2016, https://it.wikipedia.org/wiki/Git_(software), 6 dicembre 2017.

[8] Fastweb, *Che cos'è, e come funziona github,* 2016, http://www.fastweb.it/web-e-digital/che-cos-e-e-come-funziona-github/, 20 gennaio 2017.

[9] Github, *Search,* 2015, https://developer.github.com/v3/search/, 25 gennaio 2017.

[10] Github, *Rate-limit,* 2015, https://developer.github.com/v3/#rate-limiting, 25 gennaio 2017.

[11] Vargiu, E., & Urru, M. (2012). *Exploiting web scraping in a collaborative filtering-based approach to web advertising.* Artificial Intelligence Research, 2(1), 44.

[12] W3cschools, *Intro xpath,* 2016, https://www.w3schools.com/xml/xpath_intro.asp, 2 gennaio 2017.

[13] Docker, *Docker,* 2014, https://www.docker.com, 5 gennaio 2017.

[14] Github, *Laradock,* 2016, https://github.com/laradock/laradock, 2 febbraio 2017.

[15] Github, *Mastodon,* 2015, https://github.com/tootsuite/mastodon, 5 febbraio 2017.

[16] Github, *Kitematic,* 2016, https://github.com/docker/kitematic, 2 febbraio 2017.

[17] Github, *Nvidia-docker,* 2016, https://github.com/NVIDIA/nvidia-docker, 3 febbraio 2017.

[18] Github, *Che,* 2015, https://github.com/eclipse/che, 5 febbraio 2017.

Ringraziamenti

Voglio rivolgere i miei rigraziamenti al dott. Poggi e al professore Ciancarini che mi hanno aiutato a percorrere l'ultimo passo prima del traguardo. Voglio ringraziare la mia famiglia che mi ha reso quello che sono oggi e mi ha sempre invogliato e motivato lungo il cammino. Ringrazio i miei amici di sempre , che non mi hanno mai abbandonato nonostante la lontananza . Ringrazio inoltre tutte le persone che ho conosciuto lungo questo cammino, le quali si sono rivelate amicizie sincere e mi hanno aiutato a maturare.