POLITECNICO DI TORINO

Magistrale's Degree in Ingegneria Informatica

Magistrale's Degree Thesis

Unbreakable Delivery Pipeline

Supervisors

Prof. Guido MARCHETTO

Dott. Guido VICINO

Dott. Olga MORGA

Candidate

Domenico LA ROCCA

Luglio 2020

Sommario

Accelerare i tempi di rilascio del software è ormai un'esigenza di qualsiasi azienda. Il DevOps, tramite la collaborazione tra i team e l'utilizzo di nuove teconologie, permette di automatizzare tutti quei processi affidati agli sviluppatori diminuendo gli errori e rallentamenti nei rilasci.

La fase di rilascio di un software avviene in almeno due fasi: Test e Produzione.

Senza l'approccio DevOps, durante la **fase di Test**, lo sviluppatore ha il compito di decidere le risorse da assegnare all'applicazione, verificarne le funzionalità e fare degli stress test, infine, prende la decisione di convalidare il prodotto. Una volta che il software è stato rilasciato nell'ambiente di **Produzione**, si verifica a runtime il comportamento e in caso di problemi, sarà lo sviluppatore ad effettuare dei meccanismi di rollback manualmente.

L'elaborato di tesi ha come intento di migliorare le fasi descritte precedentemente con l'adozione della metodologia DevOps e inserire ulteriori automatismi che ad oggi non sono ancora adottati dalle aziende. Le tecnologie utilizzate per farlo sono state Jenkins, Istio e Spinnaker.

- **Jenkins**: attraverso le sue Pipeline è possibile programmare l'automatizzazione di varie parti del processo.
- **Istio**: per la gestione del traffico più preciso rispetto a quello della versione base di Kubernetes.
- **Spinnaker**: per effettuare il rilascio al pubblico della nuova versione, sfruttando la sua interfaccia funzionale e la sua grande compatibilità con Kubernetes.

Nella **Fase di Test** si è automatizzato il processo utilizzando i test automatici di Selenium e JMeter i cui risultati vengono analizzati sfruttando dei plugin di Jenkins. Mentre per la parte della stima delle risorse si è creato un algoritmo eseguito su una Pipeline di Jenkins.

Per distribuire al pubblico o meno la nuova versione si è deciso di classificare le performance di quest'ultima. I classificatori proposti sono due, in base alle esigenze del team DevOps si può scegliere di utilizzarne solo una o entrambe in modo complementare.

1. **Analisi matematica delle performance**, per classificare la nuova versione.

2. **Classificatore di machine learning**, sfruttando un dataset contenente le performance che deve avere una versione dell'applicazione per poter essere distribuita. È stata utilizzata la libreria python XGBOOST per l'algoritmo di parallel gradient boosting tree.

In questo modo l'intera fase iniziale di test sulla nuova versione è totalmente automatizzata se non fosse per una fase di preparazione dei dati da utilizzare.

A questo punto l'ultima fase da migliorare è la **Fase di Produzione**. Per fare ciò, sfruttando le funzionalità di Istio e Spinnaker, si è implementato un canary deploy. La nuova versione non viene più rilasciata a tutto il pubblico immediatamente ma viene resa disponibile per gradi. Ogni volta che il traffico verso la nuova versione deve aumentare viene eseguito un test automatico che va a giudicarne le performance. Il test utilizzato viene fornito da Spinnaker che sfrutta il test di Wilcoxon-Mann-Whitney. In questo modo si assicura che la nuova versione andrà a ricevere più utenti se e solo se il suo comportamento procede come programmato. Se invece il test darà un risultato negativo verrà eseguita una rollback automatica alla versione precedente.

Sfruttando tecnologie nuove, come Istio e Spinnaker, e introducendo sia il machine learning che il canary deploy gestiti in modo automatico, il progetto di tesi riesce ad automatizzare il processo. Il progetto è aperto a possibili migliorie da svolgere come lavori futuri che riguardano varie parti dell'elaborato. Si può studiare una soluzione con un classificatore in machine learning per quanto riguarda la stima delle risorse. Esso, ricevendo una serie di parametri che vanno a caratterizzare il codice, dovrebbe classificarlo come appartenente a una determinata categoria di applicazioni che necessitano di determinate risorse. Uno sviluppo interessante potrebbe essere quello di rifinire la complementarità delle due soluzioni per la classificazione della nuova versione dove il passaggio dall'analisi matematica a quella con machine learning sia automatica. Per quanto riguarda invece la fase di Produzione potrebbe essere interessante implementare una nuovo test da effettuare ad ogni step del canary deploy invece di quello offerto da Spinnaker.

Ringraziamenti

Grazie, GRAZIE, **GRAZIE**.

Prima di procedere con la trattatazione dell'elaborato penso sia necessario dedicare due parole alle persone che mi hanno sostenuto.

Questa tesi rappresenta la fine di un percorso che rappresenta un pezzo della mia vita. Non sarebbe stato possibile seguirlo se non con il sostegno di molte persone alle quali devo molto.

Per prima cosa devo ringraziare la mia famiglia, in particolare, mio padre, mia madre, mio fratello e mia sorella, ed anche nel senso più allargato del termine con zii, nonni e cugini. Senza il loro sostegno non sarebbe stato possibile neanche pensare di iniziare questa avventura al Politecnico di Torino, aggiungo anche la buona dose di pazienza che hanno dovuto avere nel s**o**pportarmi nel suo svolgimento.

Parlando di avventura, è impossibile non ringraziare chi poi mi ha accompagnato materialmente in questi anni di corsi, esami e tesi, cioè i miei colleghi, diventati amici nel corso del tempo, Luca, Enrico, Emanuele e Stefano. Avrei probabilmente abbandonato il percorso di studi senza il loro aiuto pratico e per molti versi anche psicologico nelle varie fasi degli studi.

Concentrandomi sullo sviluppo dell'elaborato di tesi è doveroso ringraziare chi mi ha dato l'opportunità di realizzarlo seguendomi nel processo. Parlo del mio relatore, il Prof. Guido Marchetto, e dell'azienda che mi ha accolto, Blue Reply, in particolare il Dott. Guido Vicino e la Dott. Morga Olga. Senza i loro preziosi consigli e suggerimenti non mi sarebbe stato possibile farcela.

Per ultimi ma non per importanza, sono da ringraziare i miei amici di 'giù', che sono comunque rimasti presenti nel corso del tempo, dove ogni volta che finalmente ci si poteva rincontrare era grande festa ed era come se non fosse passato un giorno dal mio 'abbadono' della Sicilia.

Ancora, Grazie, GRAZIE, **GRAZIE**.

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 Ambiente di sviluppo

Per introdurre l'obiettivo della tesi è necessario in primo luogo specificare l'ambiente in cui è stata sviluppata, cioè quella DevOps.

DevOps "Secondo il modello DevOps, i team dedicati a sviluppo e produzione non agiscono più separatamente. In alcuni casi, al contrario, i due team vengono fusi in un'unità in cui i tecnici sono attivi lungo tutto il ciclo di vita dell'applicazione, da sviluppo e testing a distribuzione e produzione, e acquisiscono una serie di competenze non limitate da una singola funzione." [1]

Il modello DevOps porta con sè l'utilizzo di alcune best practice [1]:

- **Continuous Integration**: è un metodo di sviluppo software dove gli sviluppatori aggiungono le modifiche al codice vegono compilate, testate e successivamente messe insieme in un repository centralizzato (Es. Git) tutto questo sfruttando dei metodi automatici, ciò risolve il problema dei conflitti tra le varie ramificazioni che può avere il codice in fase di sviluppo. È anche indicato come CI.
- **Continuous Delivery**: è un metodo di sviluppo che punta sempre all'automazione come CI ma che ne estende il campo di interesse. Le modifiche al codice vengono applicate ad una build, testate e *preparate al rilascio* in un *ambiente di produzione* in modo automatico, in questo modo è possibile avere una build temporanea sempre pronta per la distribuzione che ha già passato la fase di test. È anche indicato come CD.
- **Microservizi**: sono un tipo di architettura che può avere un'applicazione che viene divisa nelle sue funzionalità di base, ognuna della quale viene denominata *microservizio*. Ogni microservizio è indipendete dall'altro rendendone più facile

l'aggiornamento o sostituzione in quanto non è più neccesario andare a ritoccare l'intero progetto, per lo stesso motivo rende anche più semplice l'aggiunta di funzionalità. I microservizi essendo divisi gli uni dagli altri non hanno la necessità di essere creati allo stesso modo permettendo agli sviluppatori di utilizzare gli strumenti più oppurtuni per la funzione che deve svolgere. Per poter comunicare tra loro infatti utilizzano una API che di per sè è indipendente dal linguaggio utilizzato.

- **Infrastruttura come codice**: invece di gestire la configurazione fisica dell'hardware l'infrastruttura dovrebbe essere programmabile come un codice eseguibile potendogli applicare controllo di versione e CI. In questo modo l'infrastruttura ed i server possono essere distribuiti velocemente, aggiornati e duplicati.

- **Monitoraggio e accessi**: monitorare l'applicazione e l'infrastruttura è fondamentale per capire come il cambiamento delle prestazioni possa andare ad influire sull'utente finale. Quindi bisogna acquisire dati e log, suddividerli in categorie ed analizzarli in modo da poter comprendere le cause di problemi o modifiche impreviste.

- **Comunicazione e collaborazione**: sono i due aspetti fondamentali di un ambiente DevOps, migliore collaborazione e unione dei flussi di lavoro e responsabilità tra sviluppatori ed ambiente di produzione.

1.1.1 Stato dell'arte

Attualmente quando un team DevOps deve rilasciare una nuova versione di un'applicazione al pubblico segue una certa procedura che prevede molte interazioni con un operatore umano.

Build e Deploy Per prima cosa viene eseguito il build del codice per creare l'eseguibile dell'applicazione, ma a questo punto è necessario determinare le *risorse* da allocargli.

Questa stima non è possibile farla precedentemente dal team di sviluppo in quanto non si può avere un'idea precisa delle risorse occupate dall'applicazione sottoposta a carico e rilasciata in un certo ambiente che può essere sconosciuto agli sviluppatori. Quindi dopo aver rilasciato l'applicazione in un ambiente protetto vengono eseguiti una serie di test dall'operatore e dai loro risultati stimerà le risorse da dare all'applicazione.

Assegnate le risorse all'applicazione, essa viene rilasciata in un ambiente di test in cui si andranno a verificare le sue performance.

Test Automatici Per verificare il corretto funzionamento dell'applicazione vengono eseguiti dei *test automatici*. Dai risultati di questi test l'operatore dovrà scegliere se far proseguire l'applicazione nel processo di rilascio oppure farla ritornare agli sviluppatori per correggerne gli errori.

Analisi delle performance Per poter essere rilasciata ,oltre a funzionare, un'applicazione ha bisogno comunque di rispettare un certo profilo di *performance*. Per fare questo l'operatore va a guardare le metriche di vari parametri dell'applicazione mentre viene sottoposta ad un traffico creato artificialmente.

Queste metriche vengono raccolte ed analizzate, se l'applicazione rispetta certi parametri allora è pronta per essere rilasciata al pubblico. In caso contrario deve essere fatta ritornare agli sviluppatori per apportagli delle modifiche.

Rilascio al Pubblico La fase di rilascio al pubblico oggi non è gestita con una strategia particolare dalle aziende. Si va semplicemente a rilasciare l'applicazione per tutto il pubblico andando ad intervenire nel caso in cui si presentino problemi per gli utenti.

1.1.2 Obiettivo della tesi

L'obbiettivo del progetto di tesi è quello di migliorare il processo descritto precedentemente adottando la metodologia DevOps e inserire ulteriori automatismi che ad oggi non sono ancora adottati dalle aziende. Il processo di rilascio viene fatto principalmente in due fasi, la *Fase di Test* e la *Fase di Produzione*.

Fase di Test La fase di Test va ad indicare quella parte del processo in cui l'applicazione non è ancora disponibile al pubblico. Dopo il build del codice il primo obbiettivo è quello di automatizzare la stima delle risorse da assegnare all'applicazione eliminando una fase che per l'operatore può risultare particolarmente lunga e faticosa.

A questo punto l'applicazione viene rilasciata nell'ambiente di test e vengono eseguiti i test automatici. L'analisi dei risultati deve essere automatizzato con l'utilizzo di tool che ne permettano una corretta analisi.

Per quanto riguarda la parte di analisi delle performance bisogna integrare l'analisi esistente con un'alternativa che può portare a una classificazione più veloce e precisa. In particolare l'obbiettivo principale è lo sviluppo di una soluzione che sfrutti le possibilità offerte dal machine learning.

Fase di Produzione In questa fase l'applicazione viene rilasciata in un ambiente disponibile al pubblico. L'approccio utilizzato fino ad ora comporta diverse

problematiche. Intanto si espone tutta la fascia di utenti ad una applicazione che potenzialmente può ancora contenere errori non rilevati dai test automatici effettuati. Inoltre, ogni cambiamento effettuato per qualsivoglia motivo a questa nuova versione va ad impattare l'esperienza di tutti gli utenti.

Tenendo conto di queste problematiche non è possibile nemmeno svolgere ulteriori test durante questa fase in quanto essendo l'applicazione disponibile al pubblico si andrebbero ad inficiare le sue performance peggiorando l'esperienza utente. L'unico modo per poter migliorare questa parte è adottare una strategia di rilascio che permetta di ottenere una transizione facilitata dalla versione attualmente live alla nuova.

1.2 Docker

Le tecnologie base di un ambiente DevOps sono Docker e Kubernetes che verranno spiegati in questa sezione e nella successiva. Docker è una piattaforma per sviluppare, inviare, ed eseguire applicazioni [2]. Permette di separare le applicazioni dalla infrastuttura su cui è eseguita, rendendo più veloce il processo di distribuzione del software. Ma non solo le applicazioni, l'infrastruttura stessa è possibile implementarla utilizzando Docker. Sfruttando quindi le sue metodologie è possibile ridurre il ritardo tra la fase di scrittura del codice e la messa in produzione dell'applicazione, riducendo quindi i tempi di sviluppo.

Docker offre la possibilità di impacchettare ed eseguire un'applicazione in un ambiente più o meno isolato, chiamato *Container*. Dato che permettono questo isolamente rendono possibile l'esecuzione di più container nello stesso host contemporaneamente. Inoltre, risultano avere un minor consumo di risorse rispetto ad esempio ad una macchina virtuale in quanto non hanno bisogno del carico extra necessario di supervisore, dato che vengono eseguiti direttamente nel kernel dell'host.

Un componente fondamente per il funzionamento di Docker è il Docker Enrgine che è un'applicazione con un'architettura di tipo client server i cui componenti principali sono:

- Un server dove viene eseguito il ***Docker daemon*** process, crea e utilizza i Docker object, che possono essere immagini, container, network e volumi. È responsabile per il lavoro di building, running e distribution dei Docker Container.

- Una ***API REST*** che specifica l'interfaccia che i programmi possono usare per parlare con il daemon, per la connessione vengono utilizzati i socker UNIX o un'interfaccia di rete.

- Un client che fa da ***command line interface*** (CLI), utilizza l'API REST per interagire ed anche controllare il Docker daemon. Può farlo utilizzando degli script o direttamente sfruttando i comandi forniti dal CLI.

Il client e il daemon possono essere eseguiti nello stesso sistema, oppure si può connettere il client con un daemon in remoto.

Per conservare le immagini create e/o da eseguire Docker utilizza il ***Docker Registry***. Molto famoso è Docker Hub, un registry pubblico che è anche quello configurato come default. È ovviamente possibile utilizzare/creare un proprio registry privato.

Immagine Un'***immagine*** è un template in sola lettura con istruzioni per creare un container Docker. Spesso una immagine si basa su un'altra immagine con delle personalizzazioni. Per creare la propria, bisogna scrivere un *Dockerfile* dove si definiscono gli step necessari per creare l'immagine ed eseguirla. Ogni istruzione nel Dockerfile crea un layer nell'imagine. Se il Dockerfile viene modificato e si prova a fare il rebuild dell'immagine non viene ricreata tutta da 0, per ottimizzare i tempi solo ai nuovi step (layer) viene eseguito il rebuild, gli altri layer vengono recuperati dalla build precedente.

Container Un Container è un'istanza di una immagine eseguibile. Si può creare, far partire, bloccare, muovere o eliminare un container usando la Docker API o CLI. Un container può essere connesso a una o più reti, collegargli uno storage, o creare un'immagine basata sul suo stato corrente.

Di default, un container è relativamente ben isolato dagli altri container e dalla macchina che fa da host. Si può controllare il grado di isolamento della rete del container, lo storage, o altri sottosistemi sempre del container o dell'host.

Un container è definito dalla sua immagine così come le altre opzioni di configurazione che vengono date quando al momento della sua creazione ed esecuzione. Quando un container è rimosso ogni cambiamento sul suo stato che non è stato salvato su uno spazio di archiviazione persistente viene eliminato.

1.3 Kubernetes

In un ambiente di produzione c'è bisogno di gestire i container che contengono le applicazioni e assicurarsi che non ci siano tempi di inattività che possono portare a disservizi. A tutto questo cerca di porre rimedio Kubernetes, offrendo un framework per eseguire un sistema distribuito resiliente [3]. Kubernetes è una piattaforma portabile, estensibile e open-source per gestire workload containerizzati e servizi, questo facilita sia la configurazione dichiarativa sia l'automazione.

Kubernetes offre:

- **Service discovery and load balancing**: Kubernetes può esporre i container utilizzando il loro nome DNS o utilizzando il loro indirizzo IP. Se il traffico del container è alto, Kubernetes è capace di bilanciare il carico e distribuirlo nella rete così che il traffico rimanga stabile.

- **Storage orchestration**: Kubernetes permette di montare automaticamente un sistema di storage a propria scelta.

- **Automated rollouts and rollbacks**: Usando Kubernetes si possono descrivere gli stati desiderati per ogni container rilasciato, si può cambiare il suo stato, da quello attuale a uno desiderato, ad un ritmo controllato.

- **Automatic bin packing**: Kubernetes può avere un cluster di nodi che viene utilizzato per eseguire i container. Si può specificare quanta CPU e memoria deve essere assegnata ad ogni container ,ed in base alla richiesta, Kubernetes li sistema nei nodi in modo tale da utilizzare le risorse in modo efficiente.

- **Self-healing**: Kubernetes fa ripartire i container che falliscono, può sostituire o eliminare quelli che non rispondono ai controlli definiti dall'utente, senza esporli ai client prima che risultino pronti all'utilizzo.

- **Secret and configuration management**: Kubernetes permette di conservare e gestitere informazioni sensibili come password, OAuth token e chiavi SSH. Possono essere rilasciati e aggiornati segreti e configurazioni per le applicazioni senza che ne venga fatto il rebuild della loro immagine e senza esporre il segreto nello stack della configurazione.

Per lavorare con Kubernetes bisogna utilizzare gli oggetti della Kubernetes API in modo da descrivere lo stato del cluster desiderato, cioè quali applicazioni o carichi di lavoro si vogliono in esecuzione, le immagini che i container devono utilizzare, il numero di repliche ecc. Per utilizzare la Kubernetes API si va a utilizzare una command-line interface (***kubectl***).

Una volta impostato lo stato desiderato il piano di controllo di Kubernetes cambia lo stato corrente del cluster. Uno stato consiste in una collezione di processi che vengono eseguiti nel cluster:

- **Kubernetes Master**, che contiene in un solo nodo i tre processi fondamentali per controllare il cluster: kube-apiserver, kube-controller-manager e kube-scheduler.

- ogni altro nodo che non è il master deve contenere due processi. **kubelet** per poter comunicare con il master e **kube-proxy** che riflette il servizio di networking di Kubernetes in ogni nodo.

1.3.1 Namespace

I Namespace offrono uno scope per i nomi. I nomi delle risorse devono essere univoci dentro un namespace ma non tra loro. I namespace non possono essere annidati uno dentro l'altro e ogni risorsa Kubernetes può appartenere ad un solo namespace.

Kubernetes parte con tre namespace iniziali:

- **default**: il namespace per gli oggetti senza namespace specificato.
- **kube-system**: il namespace per gli oggetti creati dal sistema Kubernetes.
- **kube-public**: è creato automaticamente ed è leggibile da tutti gli utenti (anche quelli non autenticati). È utilizzato nel caso in cui qualche risorsa debba essere pubblicamente visibile e leggebile da tutto il cluster.

Quando si crea un Service, viene creato la corrispondente voce DNS, di solito nella forma <service-name>.<namespace-name>.svc.cluster.local . Se un container utilizza solo il nome del servizio esso verrà risolto in automatico all'interno dello stesso namespace.

Quasi tutte le risorse Kubernetes sono in un namespace, tranne quelle di più basso livello come i nodi e i persistent volume.

1.3.2 Pod

Un Pod è l'unità base di esecuzione di un'applicazione Kubernetes, la più piccola e semplice unità nel modello di Kubernetes che si può creare e rilasciare, rappresenta un processo in esecuzione nel cluster.

Docker è il più comune container runtime utilizzato nei Pod di Kubernetes, anche se è possibile utilizzarne di altri. I Pod in un cluster Kubernetes possono essere usati in due modi:

- **Pod che eseguono un singolo container**, questo modello è il più comune in Kubernetes, si può pensare al Pod come ad il contenitore di un singolo container e Kubernetes gestisce il Pod invece di gestire direttamente il container.
- **Pod che eseguono più container che hanno bisongno di lavorare insieme**. Un Pod può incapsulare un'applicazione composta da più container che sono strettamente legati ed hanno bisogno di condividere delle risorse. Questi containter possono formare una singola unità di servizio, un container mette a disposizione i file da un volume condiviso al pubblico, mentre un "sidecar" container separato li aggiorna. Il Pod incapsula questi container e conserva le risorse insieme come se fosse un'unica entità.

Ogni Pod è pensato per eseguire una singola istanza di una data applicazione. Se si vuole scalare l'applicazione in orizzontale bisogna utilizzare più Pod, uno per ogni istanza. In Kubernetes questo è generalmente chiamato *replication*. Essi sono creati e gestiti con un'altra astrazione di Kubernetes chiamata Controller.

Networking Ad ogni Pod è assegnato un indirizzo IP univoco. Ogni container nel Pod condivide il network namespace, incluso l'IP address e le porte di rete. I container all'interno del Pod, invece, possono comunicare l'un l'altro utilizzando localhost. Nel caso di più container all'interno di un Pod, se vogliono comunicare con entità all'esterno devono essere coordinati su come usare la rete dato che è una risorsa tra loro condivisa.

Storage Un Pod può specificare un set di Volumi condivisi. Ogni container in un Pod vi può accedere, permettendo in questo modo di condividere dati. I Volumi, inoltre permettono ai dati di sopravvivere nel caso in cui uno di questi container debba ripartire.

I Pod raramente vengono creati direttamente in Kubernetes in quanto essi sono entità effimere e considerate monouso. Quando un Pod viene creato viene stabilito in quale nodo del cluster deve essere eseguito. Il Pod ci rimane fino a quando: il Pod termina, il Pod viene eliminato, il Pod viene sfrattato per mancanza di risorse o il nodo fallisce.

I Pod non possono rigenerarsi da soli. Se un Pod in un nodo fallisce viene direttamente eliminato. Kubernetes utilizza un'astrazione di alto livello, chiamato Controller, per gestire le istanze dei Pod. Esso può creare o gestire più Pod, gestirne il replication e rollout e provvedere a rigenerarlo. Per poter istanziare i Pod i Controller utilizzano un Pod Template, fornito dall'utente, per creare i Pod per i quali sono responsabili.

Controller I Pod sono pensati per essere effimeri, quando vengono creati sono schedulati per essere eseguiti su un nodo del Cluster. Il Pod rimane nel nodo fino a quando il processo non è terminato, il Pod viene eliminato o il Pod per mancanza di risorse passa allo stato *evicted*. Da soli non hanno la possibilità di ritornare nello stato *running*, hanno bisogno di un'entità di astrazione superiore chimato *Controller*.

Un Controller può creare e controllare più Pod, gestendo la creazione di repliche, il rilascio e provvedento a capacità di rigenerazione. I Controller utilizzano un Pod Template che permette di creare Pod dei quali ne è responsabile.

ReplicaSet I ReplicaSet servono per mantenere stabile il numero di repliche dei Pod in esecuzione in ogni momento. È quindi utilizzato per garantire la disponibilità di almeno un certo numero di Pod identici.

Per identificarli al momento della definizione del ReplicaSet viene specificato quale selector utilizzare per identificare i Pod che gli appartengono in modo tale da poterne tenere traccia. C'è anche da quantificare il numero di Pod da mantenere ed il relativo Pod Template che indica i dati e le impostazioni da utilizzare.

Deployment Offre la possibilità di fare degli update dichiarativi per Pod e ReplicaSet. In un Deployment si descrive lo stato desiderato, l'effettivo cambio di stato viene fatto dal Deployment Controller. I casi in cui è principalmente utilizzato sono:

- Creare un Deployment per espletare un ReplicaSet. Successivamente andrà a creare in background i Pod necessari.
- Dichiarare un nuovo stato dei Pod, aggiornando i PodTemplateSpec del Deployment. Un nuovo ReplicaSet è creato e il Deployment gestisce lo spostamento dei Pod dal vecchio ReplicaSet al nuovo.
- Rollback verso uno stato precedente se il Deployment non è stabile.
- Scalare il Deployment per potersi far carico di più lavoro.
- Bloccare il Deployment nell'applicare le correzioni al PodTemplateSpec per poi farlo ripartire dall'inizio in un nuovo rollout.
- Utilizzare lo status di un Deployment come un indicatore che il rollout è rimasto bloccato.
- Pulire vecchi ReplicaSet che adesso non sono più necessari.

1.3.3 Service

Essendo i Pod un'entità effimera non è possibile utilizzare il loro indirizzo IP per collegarli alla rete esterna in quanto, per definizione, può variare. È qui che entrano in gioco i Service.

Un Service è un'astrazione che permette di esporre un'applicazione in esecuzione su un set di Pod come un servizio di rete. Il set di Pod puntati da un servizio sono selezionati attraverso un *selector*.

Un Service in Kubernetes è un oggetto REST, simile a un Pod. Come tutti gli oggetti REST, si può fare la POST della definizione di un Service alla API del server per crearne una nuova instanza. Il nome di un Service deve essere un DNS label name valido. Un Service viene collegato a tutti i Pod che hanno lo stesso selector contenuto nella sua definizione, una volta creato gli viene dato un indirizzo IP (chiamato anche "clusterIP"), che viene utilizzato dai Service proxy.

Il controller per il selector del Service scansiona continuamente per i Pod che ne hanno uno uguale ed effettua una POST per fare l'update su un Endpoint chiamato come il Service.

Nella sua definizione viene specificata la porta su cui contattare il servizio (definita come *port*) e viene specificata la porta (dei container) su cui reindirizzare il traffico verso i Pod (definita come *targetPort*), viene inoltre specificato il tipo di protocollo da utilizzare. Se nei Pod la porta è specificata attraverso un nome è possibile scrivere quello nel campo targetPort. È possibile specificare nella creazione del Service più porte se è neccesario esporne di più.

Ogni nodo in un cluster kubernetes esegue un kube-proxy, esso è responsabile dell'implementazione di una sorta di virtual IP per i Service di tipo diverso da ExternalName (il significato verrà spiegato più avanti).

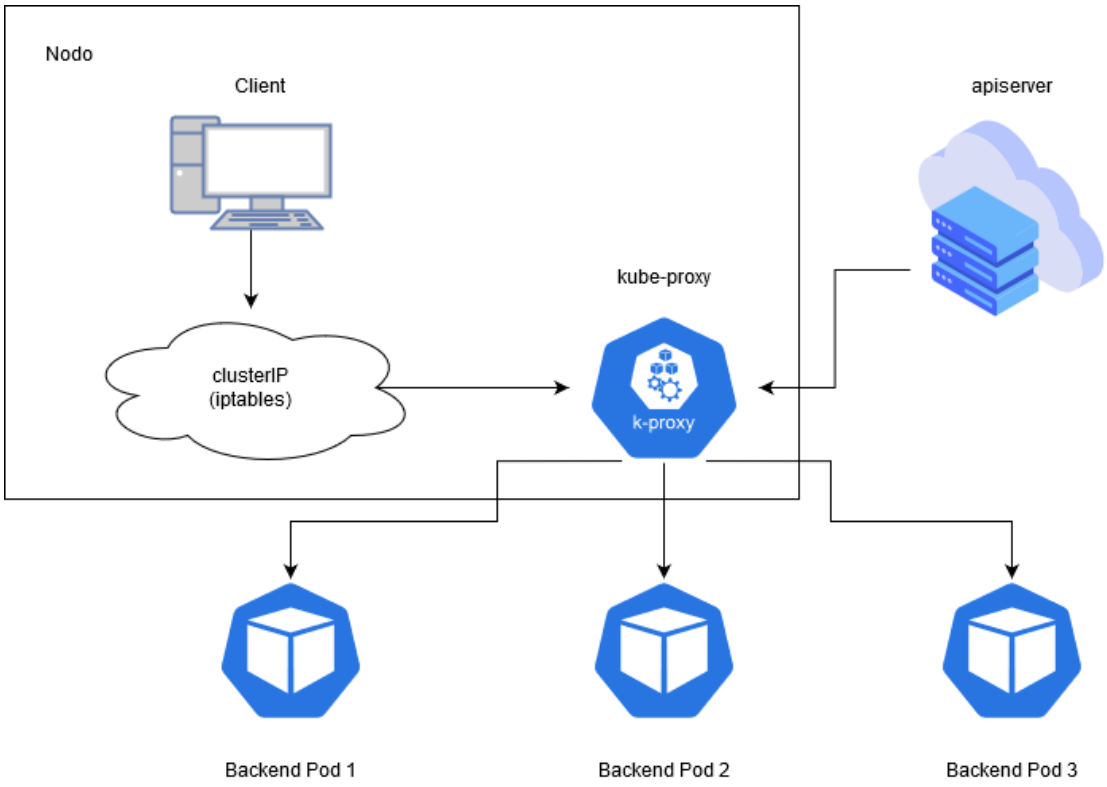


Figura 1.1: Proxy mode user space

User space proxy mode I kube-proxy osservano il Master Kubernetes per l'aggiunta o rimozione di Service e oggetti Endpoint. Per ogni Service apre una porta (selezionata casualmente) sul nodo in cui si trova. Ogni connesione a questa porta è inoltrata a uno dei Pod collegati. Infine, lo user-space proxy aggiunge regole

che catturano il traffico del clusterIP del Service e la sua relativa porta e, attraverso la loro definizione, reindirizza il traffico alla porta del proxy che sta davanti ai Pod. Di default il kube-proxy seleziona un Pod con un algoritmo round-robin.

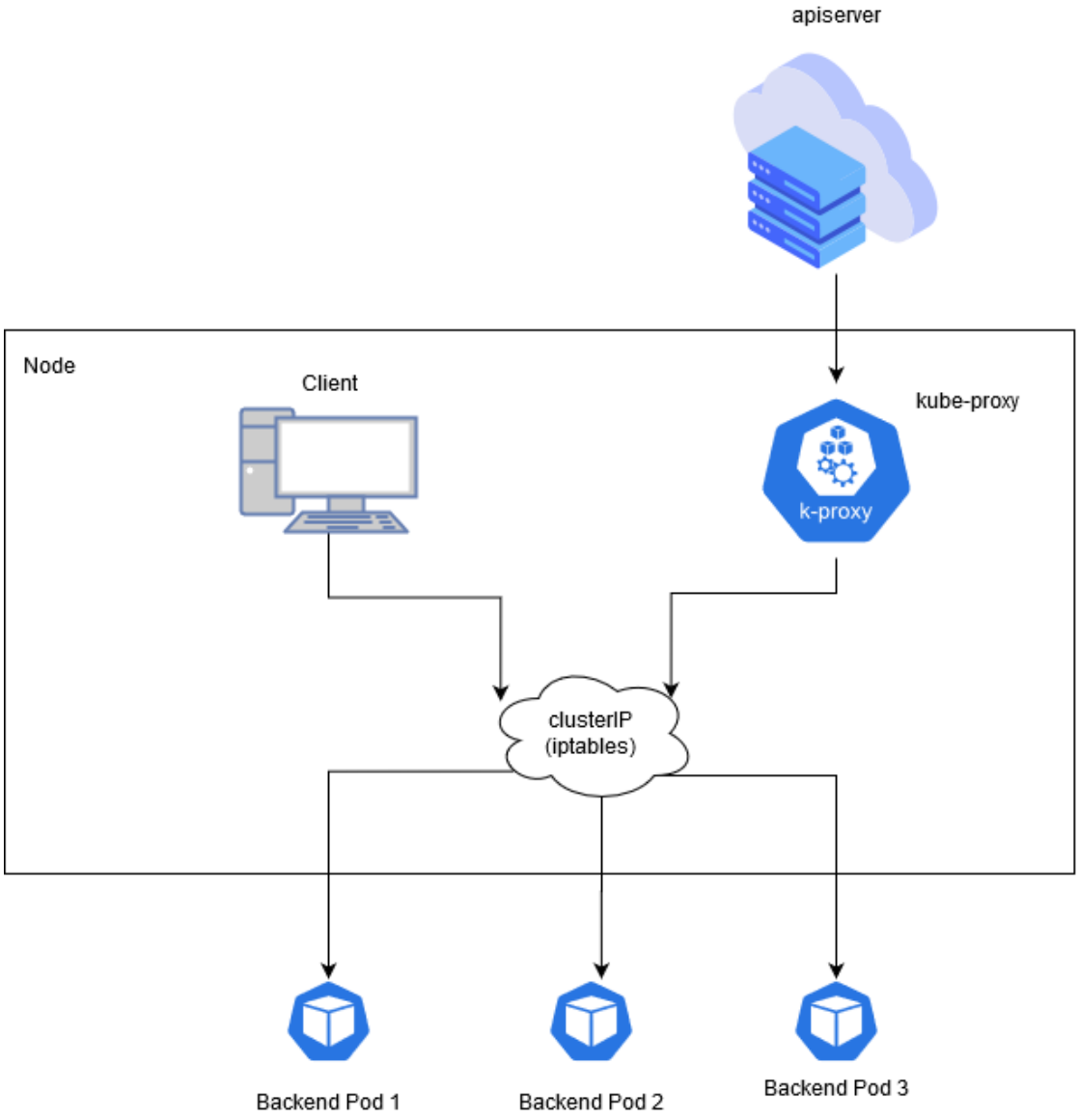


Figura 1.2: Proxy mode iptables

Iptables proxy mode In questa modalità i kube-proxy osservano il piano di controllo di Kubernetes per l'aggiunta o rimozione di Service e Endpoint. Per ogni Service viene inserita una regola nelle iptable che cattura il traffico del clusterIP

e porta del Service e la reindirizza in uno dei suoi backend. Per ogni Endpoint inserisce una regola iptable che seleziona un Pod di backend.

Di default, un kube-proxy in questa modalità di selezione del backend è casuale. Se in questa modalità il primo Pod selezionato non risponde, la connessione fallisce, differentemente dalla userspace mode che rileverebbe il fallimento del Pod e riproverebbe con un altro Pod scelto in modo casuale.

IPVS proxy mode In modalità IPVS, il kube-proxy osserva i Service Kubernetes e gli Endpoint, chiama l'interfaccia *netlink* per creare regole IPVS e le sincronizza periodicamente. Questo loop di controllo assicura che la stato di IPVS sia sempre quello desiderato. Quando si accede a un Service, IPVS reindirizza il traffico a uno dei Pod backend.

Questa modalità è basata su delle funzioni di netfilter hook in modo simile alla modalità iptable, ma utilizza delle tabelle di hash come struttura dati e lavora a livello kernel. Questo comporta che rispetto alla modalità iptable la latenza è minore, con performance migliori al momento della sincronizzazione delle regole. Rispetto alle altre modalità, inoltre, IPVS supporta un throughput maggiore. Esistono diverse modalità per bilanciare il traffico tra i Pod in backend che sono:

- **rr**: round-robin
- **lc**: least connection (minor numero di connessioni aperte)
- **sh**: source hashing
- **sed**: shortest expected delay
- **nq**: never queue

DNS È possibile creare un servizio DNS all'interno di un cluster Kubernetes utilizzando un add-on.

Un server DNS cluster-aware, come ad esempio CoreDNS (utilizzato nel corso della tesi), osserva l'API di Kubernetes per nuovi Service e crea un set di record DNS per ognuno di loro. Se attivato, tutti i Pod possono risolvere l'indirizzo dei Service con il loro nome DNS. Il nome creato segue la struttura <nome-service>.<nome-namespace>. Per questo motivo i Service devono avere un nome valido per poter essere inserito all'interno di un servizio DNS.

ExternalIPs Se esistono indirizzi IP esterni che instradano il traffico in uno o più nodi del cluster, i Service possono essere esposti su di essi. Il traffico che è in ingresso al cluster con questi indirizzi IP sulla porta del Service viene reindirizzato ai suoi Endpoint. Gli externalIPs non sono gestiti da Kubernetes ma sono una

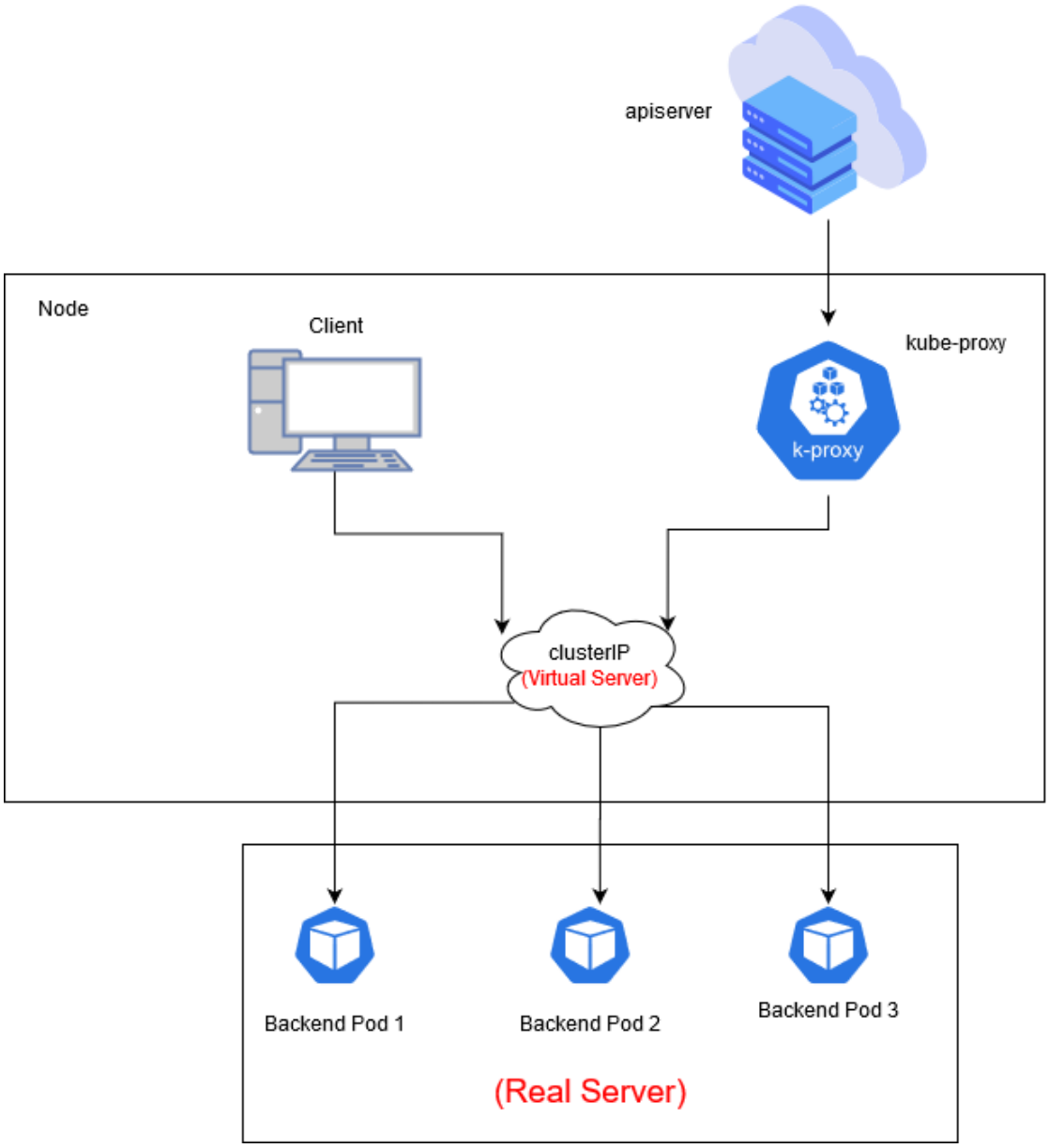


Figura 1.3: Proxy mode IPVS

responsabilità dell'amministratore del cluster. Il campo externlIPs può essere specificato, come anche il tipo del Service, nel suo file yaml.

Esporre i Service

Alcune applicazioni o parti di esse devono essere esposte su un indirizzo IP esterno, fuori dal cluster. Il campo ServiceTypes di Kubernetes permette di specificare che tipo di Service si vuole, quello di default è ClusterIP. Ne esistono di differenti tipi:

- **ClusterIP**: espongono il Service su un indirizzo IP interno al cluster, rendendolo raggiungibile solo dal suo interno.
- **NodePort**: espone il Service su ogni indirizzo IP dei nodi. Viene creato un Service di tipo ClusterIP a cui il Service di tipo NodePort fa capo. Sarà possibile contattare la *nodePort* del Service da ogni indirizzo IP dei nodi.
- **LoadBalancer**: espone il servizio esternamente utilizzando un load balancer del cloud provider. Il load balancer instrada il trafficao verso dei Service NodePort e ClusterIP che sono automaticamente creati.
- **ExternalName**: mappa il Service con il contenuto del campo externalName, ritornando un record CNAME con questo valore.

NodePort Quando viene creato un Service di tipo NodePort il piano di controllo di Kubernetes alloca una porta in un range specificato nel campo –service-node-port-range (di default è tra 30000 e 32767). Ogni nodo fa da proxy per il Service sulla porta selezionata. La porta assegnata al Service è specificata nel campo spec.ports[*].nodePort.

È possibile impostare un particolare indirizzo IP che fa da proxy per i NodePort attraverso un campo del kube-proxy. È anche possibile specificare una determinata porta come nodePort e non per forza passare dal piano di controllo di Kubernetes.

I NodePort permetteno di impostare una soluzione proprietaria per il load balancing, configurare ambienti che non supportano completamente Kubernete, o solamente esporre uno o più indirizzi IP dei nodi.

LoadBalancer Se si ha un cloud providers che supporta load balancer esterni, creando un Service di tipo LoadBalancer, viene dato automaticamente un load balancer per il Service creato. La creazione del load balancer avviene in modo asincrono e le informazioni sul balancer dato è pubblicato nel campo .status.loadBalancer del Service. Il cloud provider decide che tipo di load balancer mettere a disposizione.

Il traffico del load balancer esterno vieni reindirizzato ai Pod backend.

Alcuni cloud provider permettono di specificare l'indirizzo IP da utilizzare per il load balancer. Se l'indirizzo IP scelto non è disponibile, il load-balancer è comunque creato, ma gli viene assegnato un indirizzo IP effimero. Se invece è il cloud provider stesso a non prevedere questa funzionalità l'eventuale scelta dell'indirizzo IP dell'utente viene ignorata.

1.3.4 Volumi

I file su disco nei container sono effimeri, questo fa sorgere dei problemi in alcune applicazioni se eseguite al loro interno. Quando un container si blocca, kubelet lo riavvia, ma i file che sono stati creati fino a quel momento vengono persi poiché il container riparte sempre reinizializzando l'immagine di base selezionata. Un'altra problematica è quella della condivisione dei file tra container che si trovano in esecuzione nello stesso Pod. I *Volumi* Kubernetes sono un'astrazione che risolve entrambi i problemi.

Docker possiede il concetto di Volume, ma non possiede le caratteristiche necessarie per essere la soluzione ideale in un ambiente Kubernetes. Su Docker i Volumi sono delle semplici cartelle nel disco o nel caso in un altro container. Il suo ciclo di vita **non** è gestito.

Invece un Volume Kubernetes ha un ciclo di vita esplicito, cioè lo stesso del Pod in cui è contenuto. Conseguentemente i Volumi soppravvivono ai container eseguiti dentro il Pod e i dati continuano ad essere presenti anche nel caso in cui il container necessita di essere riavvitato. Quando un Pod cessa di esistere il Volume viene a sua volta eliminato, è possibile utilizzare più tipi di Volumi e i Pod possono utilizzarne di vario tipo e in vario numero simultaneamente.

Un Volume viene visto come una cartella, possibilmente con dei dati al su interno, che è accessibile dai container nei Pod. Come questa cartella viene resa disponibile, la struttura che ci sta dietro ed il suo contenuto sono determinati dal particolare tipo di Volume utilizzato. Ne esistono quindi di vario tipo, in seguito viene fatta una panoramica su quelli che sono stati particolarmente utilizzati nel corso della tesi.

ConfigMap Le risorse di tipo *configMap* offrono un modo per inserire dei dati di configurazione all'interno dei Pod. I dati conservati in un oggetto ConfigMap possono essere refereziati da un Volume di tipo configMap per poi essere consumati da un'applicazione all'interno di un container in esecuzione in un Pod.

Local Un Volume di tipo *local* rappresenta uno spazio di archiviazione montato come disco, partizione o cartella. Possono essere usati solo come PersistentVolume staticamente creati.

Sono soggetti alla disponibilità del nodo sottostante e non sono ottimali per tutte le applicazioni. Se un nodo passa ad uno stato *unhealthy*, il local Volume diventerà inaccessibile ed il Pod che lo sta utilizzando rimarrà bloccato. Le applicazioni che usano un local Volume devono essere capaci di sopravvivere a questa eventualità, come anche la possibilità di perdere dati, dipendentemente dalle caratteristiche del disco del nodo.

NFS Un Volume *nfs* permette ad un esistente NFS (Network File System) di essere montato in un Pod.

Quando un Pod viene rimosso, il contenuto di un volume nfs è preservato ed il Volume praticamente non viene rimosso. Questo significa che un Volume NFS può essere popolato con i dati prima della creazione del Pod. Un'altra caratteristica che hanno i Volumi nfs è che i container possono utilizzarli per scambiarsi i dati tra loro in quanto un Volume nfs può essere montato da più writer simultaneamente.

PersistentVolumeClaim Un Volume *persistentVolumeClaim* è utilizzato per montare un ***PersistentVolume*** in un Pod, come funzionano effettivamente i PersistentVolumeClaim e i PersistentVolume verrano spiegati nella sezione successiva.

Persistent Volume

Il sottosistema dei PersistentVolume offre un'API per gli utenti ed amministratori che astrae i dettagli di come uno spazio di archiviazione è offerto e come esso venga consumato. Per questo vengono introdotte due nuove risorse, ***PersistentVolume e PersistentVolumeClaim***

Un PersistentVolume (Indicato anche come PV) è uno spazio di archiviazione nel cluster che è stato dato da un amministratore, o dato dinamicamente utilizzando uno *Storage Class*. I PV sono aggiunti come i Volumi, ma hanno un ciclo di vita indipendente dai Pod che li utilizzano.

Un PersistentVolumeClaim (Indicato anche come PVC) rappresenta una richiesta di spazio di archiviazione da un utente. I PVC "consumano" i PV e ne possono richiedere uno con una specifica quantità di spazio come anche discriminare in base al modo di accesso.

Per dare la possibilità agli amministratori dei cluster di avere una varietà di PV che differiscono non solo per dimensioni e modalità di accesso, senza però esporre agli utenti i dettagli di come questi Volumi sono stati implementati, si utilizzano gli StorageClass.

Ciclo di vita di un volume e claim I PV rappresentano una risorsa nel cluster. I PVC, invece, sono una richiesta per queste risorse. Le interazioni tra PV e PVC seguono un determinato ciclo di vita.

Provisioning Esistono due modi in cui un PV può essere creato, statico o dinamico.

- **Statico** l'amministratore del cluster crea uno o più PV. Contengono già al loro interno i dettagli sul reale storage che è disponibile all'utilizzo da parte degli utenti del cluster. I PV creati sono disponibili dall'API di Kubernetes per essere consumati.
- **Dinamico** quando nessun PV creato staticamente rispetta il PVC di un utente, il cluster deve provare a crearlo dinamicamente. Questo sistema è basato sullo StorageClass, il PVC deve richiederne uno fra quelli messi a disposizione dall'amministratore.

Binding Un utente crea un PVC che richiede una specifica quantità di storage e con certe modalità di accesso. Quando viene creato un PVC viene notato dal ciclo di controllo di Kubernetes che cerca un PV in grado di soddisfarne le richieste. Se il PV è stato creato dinamicamente apposta per soddisfare le richieste del PVC l'accoppiamento avviene in automatico.

Un PV può essere collegato ad un solo PVC e viceversa, se non viene trovato un PV adatto al PVC allora rimarrà in attesa fino a quando non se ne presenterà uno.

Utilizzo I Pod utilizzano i PVC come Volumi. Il cluster ispeziona il PVC per trovare il PV a cui è collegato e ne fa il mount per il Pod.

Una volta che l'utente ha un PVC, ed è accoppiato al PV, gli apparterrà fintanto che ne avrà bisogno.

Reclaim Quando un utente ha finito di utilizzare il PVC può eliminarne l'oggetto dall'API, a quel punto in base alla *reclaim policy* adottata si aprono diversi scenari riguardo al cosa fare delle risorse del PV, sono possibili tre strade:

- **Retain**: permette di risanare le risorse manualmente. Quando un PVC è eliminato il PV esiste e viene considerato "released", ma non è disponibile per essere preso da un altro PVC. Per liberarlo si può o eliminare il PV (pulire i dati manualmente) o, sempre manualmente, ricreare un altro PV per quello storage.
- **Delete**: viene rimosso sia l'oggetto PV, sia lo storage nella infrastruttura esterna.
- **Recycle**: quello che viene eseguito è essenzialmente un "rm -rf /volume/*" e lo rende disponibile di nuovo per un PVC. È tuttavia possibile impostare un comando qualunque al momento del recycle.

Storage Class

Uno *StorageClass* offre agli amministratori un modo per descrivere una "classe" di storage da offrire. Le classi possono differenziarsi per: livelli di qualità del servizio, policies di backup o in base a politiche arbitrarie scelte dall'amministratore del cluster. Ogni StorageCass contiene i campi *provisioner*, *parameters* e *reclaimPolicy* che sono utilizzati quando un PV appartenente ad una classe deve essere dinamicamente offerto.

Provisioner Ogni StorageClass ha bisogno di specificare che tipo di Volume plugin usare per creare il PV. Ciò viene specificato nel campo provisioner. Sono possibili diversi plugin come visto nella parte in cui si descrivevano i Volumi, quelli che sono stati usati nella tesi sono NFS e Local.

Reclaim Policy Nel campo reclaimPolicy dello StorageClass si va a specificare la Reclaim Policy che verrà utilizzata per il PV da creare. È possibile utilizzare solamente *delete* e *retain* (spiegati in precedenza). Se la reclaim policy non è stata specificata quella utilizzata di default viene impostata la delete.

Volume Binding Mode Il campo volumeBindingMode controlla quando deve avvenire la creazione ed il binding del Volume.

La modalità di default è quella ***Immediate***. In questo modo la creazione ed il binding del PV viene fatta al momento della creazione del PVC. Se l'architettura di storage non permette una modalità del genere il PV verrà creato e collegato senza andare a controllare le richieste per poter schedulare il Pod. Ciò potrebbe portare a Pod che non possono passare allo stato di esecuzione.

Per evitare questo problema è possibile utilizzare un'altra modalità, cioè la modalità ***WaitForFirstConsumer*** che ritarda la creazione e l'accoppiamento del PV fino a quando il Pod ed il PVC sono creati. In questo modo il PV viene selezionato o creato a seconda dei vincoli specificati al momento della schedulazione del Pod.

Capitolo 2

Tecnologie utilizzate

2.1 Jenkins

Jenkins è un server di automazione, autosufficiente e open-source utilizzato per automatizzare qualunque tipo di processo di compilazione, testing, delivery e deploy [4]. Permette di fare CICD (Continuous Integration Continuous deploy) definendo una serie di step in una Jenkins Pipeline. L'avvio della Pipeline può essere manuale o con un trigger impostato per attivarsi in caso di determinati eventi (es. push su repository, scadenza di un timer). Un altro valido motivo per utilizzare Jenkins è la sua capacità di riportare i report dei test effettuati e l'invio di notifiche sullo stato dell'esecuzione (es. e-mail).

Grazie alla presenza di numerosi plugin Jenkins è uno strumento estremamente versatile capace di adattarsi a tutte le situazioni.

L'automatizzazione dei processi avviene attraverso la scrittura di Jenkins Pipeline (da ora in poi chiamate semplicemente Pipeline).

Jenkins Pipeline Le definizione di una Jenkins Pipeline è scritta in un file di testo chiamato Jenkins File.

Le caratteristiche principali di una Jenkins Pipeline sono:

- **Code**: le Pipeline sono implementate con la scrittura di codice, con la possibilità di avere il controllo di versione, la possibilità di modificare, revisionare e iterare sulla Delivery Pipeline.
- **Durabilità**: la Pipeline sopravvive ai restart che siano volontari o meno del Jenkins master.
- **Pausa**: una Pipeline può essere messa in pausa ed aspettare per un intervento umano per farla ripartire.

- **Versabilità**: le Pipeline rispettano i requisiti di CD, con la possibilità di fare fork/join, iterare ed eseguire del lavoro in parallelo.

- **Estensibile**: il plugin delle Pipeline supporta le estensioni custom al suo **DSL** e ha più opzioni di integrazione con altri plugin.

Esistono due tipi di Pipeline, le *Declarative* e le *Scripted* Pipeline. Le Declarative Pipeline sono un'aggiunta recente a Jenkins e sono loro che hanno la caratteristica di poter essere scritte in un Jenkinsfile che può essere messo in un sistema di gestione di controllo come ad esempio Git. Le Scripted Pipeline non hanno questa funzionalità, il codice deve essere sempre scritto nella UI di Jenkins.

Entrambe le Pipeline si basano sul groovy DSL (Domain-Specific Language). Le Scripted Pipeline usano praticamente solo il groovy DSL, per questo motivo offrono un grosso controllo sullo script e può essere ampiamente manipolato. Le Declarative Pipeline, invece, impongono all'utente una struttura predefinita.

2.2 Istio

Nel contesto di un team DevOps che lavora sulla portabilità di microservizi e alla gestione di deploy variegati Istio aiuta a ridurne la complessità e ne agevola il lavoro [5]. L'insieme di microservizi che compongono un'applicazione e l'interazione fra essi viene chiamata *Service Mesh*, Istio permette di operare ad alto livello trattando una service mesh nel suo insieme. Le sue funzionalità possono essere divise in delle macro aree, alcune elle quali sono state molte utilizzate nel corso dello sviluppo della tesi mentre altre meno. Per quest'ultime verrà fatta una introduzione giusto per far vedere le potenzialità di Istio mentre per le altre verrà effettuata una trattazione più approfondita in seguito.

Security I microservizi hanno bisogno di particolari attenzioni riguardo la sicurezza:

- Devono difedersi da attacchi di tipo man-in-in-the-middle, quindi hanno bisogno di crittografare il traffico.

- Offrire un accesso ai servizi flessibile, hanno bisogno di mutual TLS e politiche di accesso particolareggiate.

- Controllo su chi ha fatto accesso, cosa ha fatto e in quale momento.

Istio offre delle soluzioni a questi problemi fornendo: forte autenticazione, autorizzazione, transparent TLS, crittografia e strumenti di audit per proteggere i servizi e i dati. Gli obiettivi principali sono:

- **Security by default**: non c'è bisogno di fare cambiamenti al codice dell'applicazione e alla infrastruttura.
- **Defense in depth**: integrarsi con i sistemi di sicurezza esistenti per dare più strati di difesa.
- **Zero-trust network**: costruire soluzioni di sicurezza senza avere come requisito la sicurezza intrinseca della rete.

Policies Istio permette di personalizzare le policy delle applicazioni per forzare delle regole a runtime come:

- Limitare staticamente o dinamicamente il traffico verso un servizio.
- Divieti, whitelist e blacklist per limitare l'accesso a un servizio.
- Riscrivere header e reindirizzazione del traffico.

Oltre a questo, permette anche di creare i propri policy adapters.

2.2.1 Traffic Managment

Le Traffic Rule di Istio permettono di controllare il traffico e le chiamate alle API dei servizi.

Istio semplifica la configurazione delle proprietà a livello di servizi come ad esempio: circuit breaker, timeout, e retries. Semplifica la procedura per creare A/B testing, canary deploy e staged rollout grazie alla possibilità di dividere il traffico tra più servizi impostando la percentuale da dare ad ognuno.

Il Traffic Managment di Istio si affida agli Envoy proxy che vengono rilasciati insieme ai servizi. Tutto il traffico della Service Mesh passa attraverso questi proxy rendendo più facile controllarlo senza dover apportare cambiamenti ai servizi.

Per poter fare ciò è necessario conoscere dove si trovano tutti gli endpoint e a quali servizi appartengono, queste informazioni vengono contenute nel *Service Registry*, per popolarlo Istio si connette a un sistema di ricerca dei servizi (se ad esempio Istio è stato installato su un cluster Kubernetes riesce a farlo in automatico grazie alle informazioni che Kubernetes offre di suo). È grazie al Service Registry se gli Envoy proxy possono mandare il traffico ai servizi.

La maggior parte delle applicazioni a microservizi hanno più istanze di ogni servizio per ricevere il traffico. La distribuzione del traffico verso queste molteplici istanze viene fatta dagli Envoy proxy, di deafult viene utilizzato come algoritmo di schedulazione il Round Robin.

Utilizzando alcune astrazioni, che verrano introdotti nei paragrafi successivi, Istio permette di controllare più finemente il traffico.

Virtual Service I Virtual Service sono lo strumento fondamentale che rendono il Traffic Managment di Istio flessibile e potente, grazie a loro è possibile fare il decoupling tra dove il client invia la sua richiesta e dove effettivamente viene presa in carico.

Con i Virtual Service si può specificare il comportamento del traffico in base a uno o più hostname. Si possono specificare delle rounting rule nei Virtual Service che specificano agli Envoy proxy come inviare il traffico alle destinazioni. Le destinazioni possono essere differenti versioni dello stesso servizio o completamente diversi.

Il caso più frequente è quello di inviare il traffico a diverse versioni dello stesso servizio, specificati come subset del servizio. I client inviano il traffico verso il servizio, vedendolo come una entità unica, saranno gli Envoy proxy a reindirizzarlo tra le diverse versioni a seconda delle regole specificate nel Virtual Service. Le traffic rule sono completamente separate da come effettivamente sono stati rilasciati i servizi, questo implica che il numero di istanze può scalare in base al carico della rete senza andare a modificare le traffic rule precedentemente create. Ciò non è possibile da fare con il solo Kubernetes in quanto esso offre la possibilità di fare distribuzione del traffico basandosi sulla variazione del numero delle istanze, il che diventa complicato se il servizio diventa più complesso.

I Virtual Service permettono di indirizzare più microservizi definendo un unico Virtual Service, facilitando la transizione da una applicazione monolitica a una ad una a microservizi. Un'altra possibilità offerta è quella di combinare le traffic rule ai *Gateway* per controllare il traffico in entrata e in uscita.

Spesso è però necessario utilizzare le *Destination Rule* per utilizzare correttamente i Virtual Service dato che permettono di dividere le varie istanze dei servizi in subset, in questo modo si può riutilizzare lo stesso Virtual Service per più implementazioni.

Destination Rule Le Destination Rule sono un'altra parte importante del Traffic Managment di Istio, se i Virtual Service vanno a specificare a chi e come deve essere destinato il traffico, le Destination Rule specificano cosa succede al traffico una volta raggiunta la sua destinatione. Le Destination Rule vengono applicate dopo che vengono valutate le traffic rule specificate nei Virtual Service, quindi agiscono sul traffico 'reale'. In particolare, si utilizzano per specificare dei sottoinsiemi sui named service, come ad esempio raggruppare tutte le istanze di un servizio in base alla versione, successivamente, si possono utilizzare questi sottoinsiemi per creare routing rule per regolare il traffico tra le differenti versioni.

Un'altro utilizzo delle Destination Rule è quello di modificare le traffic policy degli Envoy proxy quando chiamano per intero un servizio o un suo particolare sottoinsieme, ad esempio selezionare uno specifico modello di load balance, TLS

security mode o le impostazioni di circuit breaker. In particolare sono possibili diversi modelli di round-robin load balancing che sono:

- **Random**: le richieste sono inoltrate in modo casuale tra le istanze disponibili.
- **Weighted**: le richieste sono inoltrate alle istanze, per ogni subset viene decisa una percentuale di traffico che gli deve arrivare.
- **Least requests**: le richieste sono inoltrate alle istanze che ne hanno ricevuto un numero minore.

Gateway Il Gateway viene utilizzato per gestire il traffico in entrata e in uscita, dando la possibilità di specificare quale traffico può entrare o uscire. La configurazione dei Gateway viene applicata agli Envoy proxy che sono ai confini della mesh invece di farlo per ogni Envoy proxy (*sidecar*) che si trova in esecuzione per ogni servizio. I Gateway di Istio permettono di configurare le proprietà di bilanciamento del carico dal livello 4 al 6 come porte da esporre, configurazione di TLS, ecc. Invece per aggiungere traffic rounting al livello applicativo si associano i Virtual Service di Istio ai Gateway. Questo permette di gestire il traffico dei Gateway come ogni altro traffico nella mesh.

Il principale utilizzo è per il traffico in ingresso ma si possono configurare anche egress Gateway. Un egress Gateway permette di configurare uno specifico nodo di uscita per il traffico della mesh, permettendo di limitare quale service può accedere alla rete esterna o per effetture controlli di sicurezza sul traffico.

Istio mette a disposizione alcuni deploy di Gateway preconfigurati. Si può applicare la proprio configurazione a questi deploy o farli da zero.

Sidecars Di base Istio configura ogni Envoy proxy per accettare il traffico per ogni porta associata al proprio carico di lavoro e di raggiungere ogni service nella mesh quando inoltra il traffico. Si possono utilizzare delle configurazioni dei Sidecar per:

- impostare le porte e i protocolli che un Envoy proxy accetta.
- limitare l'insieme di servizi che l'Envoy proxy può raggiungere.

Limitare la raggiungibilità dei Sidecar diventa particolarmente interessante nel caso di applicazioni molto grandi dove avere tutti i servizi raggiungibili da ognuno può potenzialmente peggiorare le prestazioni della mesh per via dell'elevato utilizzo di memoria.

Si può specificare se si vuole che la configurazione del Sidecar sia applicabile a tutti i service in uno specifico namespace, oppure applicarla per alcuni service selezionati singolarmente.

2.2.2 Observability

Istio produce telemetrie dettagliate per ogni servizio in comunicazione con la mesh. Con le telemetrie si fornisce l'osservabilità sul comportamento dei servizi, facilitando gli operatori durante le fasi di troubleshooting, manutenzione e ottimizzazione delle loro applicazioni. Istio genera i seguenti tipi di telemetrie per offrire osservabilità:

- **Metrics**: metriche sui servizi basate su latenza, traffico, errori e saturazione, oltre a fornire metriche dettagliate sul piano di controllo della mesh. È fornita, inoltre, una dashboard su queste metriche.
- **Distributed traces**: Istio genera distributed trace spans per ogni servizio, dando agli operatori informazioni dettagliate sulla serie di chiamate effettuate e le dipendenze tra i servizi dentro la mesh.
- **Access Logs**: Istio genera dei record per ogni richiesta, includendo il mittente ed il destinatario.

Metrics

Per monitorare il comportamento di un service, Istio genera delle metriche per tutto il traffico in ingresso, in uscita e tra i servizi della mesh. Oltre a monitorare i service, è importante monitorare la mesh in sè, i componenti di Istio esportano metriche del loro comportamento interno fornendo visione sullo stato del piano di controllo della mesh. La collezione delle metriche di istio è guidata dalla configurazione degli operator. Si può configurare come e quando collezionare le metriche e quanto dettagliate esse devono essere.

Proxy-level metrics Le metriche di Istio iniziano ad essere prese con i sidecar proxy. Ognuno genera una ricca serie di metriche sul traffico che gli passa attraverso. Inoltre, offrono statistiche dettagliate sulle funzioni amministrative dello stesso proxy, includendo configurazione e salute delle informazioni.

È possibile, attraverso gli operator, selezionare da quali Envoy proxy vengono generate le metriche a seconda del carico di lavoro. Di default solo una minima parte delle metriche è generata, per evitare il sovraffollamento di metriche e per ridurre il CPU overhead associato, anche se è possibile abilitarne di altri.

Service-level metrics Istio offre un set di metriche service-oriented per monitorare la comunicazione dei servizi. Quelle offerte coprono la latenza, il traffico, gli errori e la saturazione.

Di default, sono definite attraverso un set di artefatti di configurazione che vengono esportati a Prometheus di default. Gli operatori sono liberi di modificare la forma ed il contenuto di queste metriche, come anche il modo con cui le collezionano.

Prometheus Prometheus è un sistema di monitoraggio open-source oltre ad essere anche un alerting toolkit [6]. Si adatta bene sia a monitare sistemi machine-centric, sia a monitorare architetture service-oriented molto dinamiche. Per rendere Prometheus il più affidabile possibile, ogni Prometheus server è standalone, non dipende ne da network storage ne da servizi remoti. L'ecosistema di Prometheus consiste in più componenti, molti dei quali opzionali:

- Il principale Prometheus server che raccoglie e conserva i dati per ogni intervallo temporale.
- client libraries per la strumentazione del codice applicativo.
- un push gateway per supportare azioni effimere.
- exporters special-purpose per servizi come HAProxy, StasD, Graphite, ecc.
- un alertmanager per gestire gli alerting
- altri strumenti di supporto.

Dalle metriche che conserva, Prometheus può anche generare serie temporali derivate da dare come risultati delle query. Ogni time series (serie temporale) è identificata con un metric name e facoltativamente da una coppia chiave-valore chiamata label. Esistono principalmente quattro tipi di metriche:

- **Counter**: rappresenta un contatore che in modo monotono aumenta partendo da zero.
- **Gauge**: è una metrica che rappresenta un singolo valore numerico che può arbitrariamente aumentare o diminuire.
- **Histogram**: campiona una metrica e la divide in bucket configurabili. Inoltre offre anche una somma di tutti i valori osservati.
- **Summory**: ogni campione offre una somma di tutte le osservazione effettuate più un contatore che rappresenta quante ne sono state fatte.

Query Prometheus offre un linguaggio di query chiamato PromQL che permette agli utenti di selezionare ed aggregare i dati in tempo reale. Il risultato può essere visto come un grafico o come una serie di dati, può essere visto come tabella di dati sulla UI di Prometheus via browser, oppure possono essere consumati da sistemi esterni attraverso la sua API HTTP. Esistono vari tipi di dati che possono essere ritornati da una query PromQL.

- **Instant vector**: una seri di campioni, uno per ogni serie temporale, che hanno lo stesso timestamp.

- **Range vector**: una serie di campioni sequenziali in un certo range temporale.
- **Scalar**: un singolo valore di tipo *float*.
- **String**: una stringa.

InfluxDB InfluxDB è un database di serie temporali, creato per gestire carichi di lavoro in scrittura e di query [7]. È utilizzato principalmente come storage per le applicazioni che gestiscono una grossa mole di dati come ad esempio il caso del monitoraggio nell'ambiente DevOps, sistemi per il monitoraggio di dati in tempo reale e applicazioni che gesticono metriche.

Le caratteristiche principali sono:

- Datastore con alte performance specificamente creato per serie temporali. Il motore utilizzato permette di ricevere tanti dati e comprimerli velocemente.
- È scritto completamente in Go (Linguaggio open-source sviluppato da Google). Compilato tutto in un singolo file binario senza dipendenze esterne.
- Offre una API HTTP con alte performance in scrittura e di query.
- Supporto a diversi plugin per l'ingestione di dati con altri protocolli.
- Un linguaggio di query SQL-like creato per fare query di dati aggregati.
- I tag che permettono di indicizzare le serie di dati in modo da fare query veloci ed efficienti.
- Retention policy per far 'scadere' i dati bloccati in modo efficiente.
- Vengono fatte periodicamente delle query in modo automatico per rendere più efficienti le query sui dati aggregati che vengono richieste più di frequente.

Integrazione con Prometheus L'utilizzo principale nel corso della tesi dell'InfluxDB è stato come backend storage per Prometheus. È stato creato un database per raccogliere i dati da Prometheus, a quel punto si è dovuto modificare la configurazione di Prometheus per impostare la scrittura e lettura in remoto verso questo database.

Con la situazione così configurata i dati sono conservati nell'InfluxDB ma l'utente finale può continuare ad interrogare Prometheus per recuperare i dati utilizzando il suo linguaggio di query ed il suo modo di aggregarli.

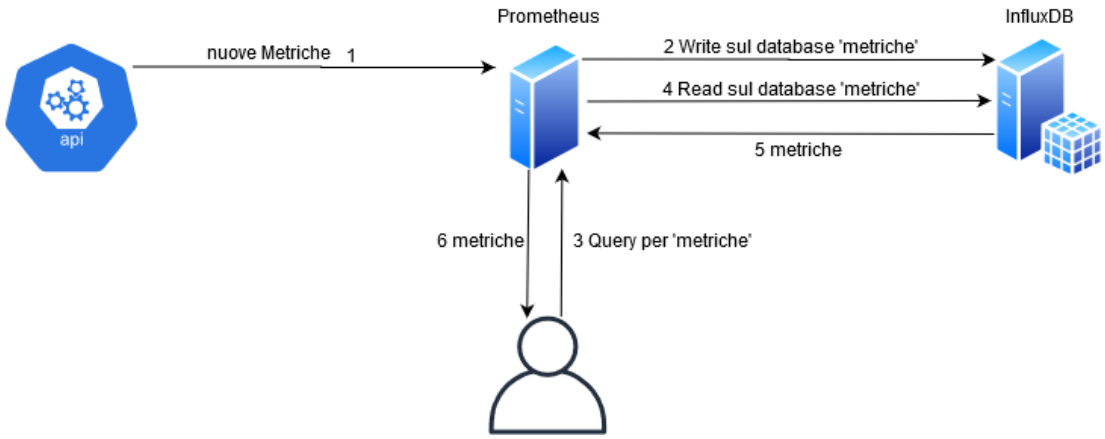


Figura 2.1: Integrazione di Promtheus con l'InfluxDB

Grafana Grafana permette di interrogare, visualizzare, avvisare e capire le metriche, non importa dove esse siano conservate. In particolare, Grafana ha la possibilità di utilizzare Prometheus come fonte per le metriche. In questo modo è possibile creare una dashboard dov'è possibile tenerne sotto controllo l'andamento. Il blocco di visualizzazione fondamentale è il panel, per ogni dashboard è possibile specificare la fonte dei dati e le query per estrarne i dati. Ad esempio il Graph panel permettere di tracciare un grafico su quante metriche e serie si vogliono. Un altro tipo molto utilizzato è il Singlestat che perme la visualizzazione di un solo dato alla volta.

Distributed traces Le Distributed trace offrono un modo per monitorare e capire il comportamento di ogni richiesta così per come loro scorrono attraverso la mesh. Permettono di comprendere le dipendenze e le sorgenti di latenza dei servizi nella mesh.

Istio supporta i Distributed trace attraverso gli Envoy proxy. Essi generaneraro automaticamente trace span per conto delle applicazioni per cui fanno da proxy, richiedendo solo che le applicazioni inoltrino una request context appropriata.

Istio supporta vari backend di tracing tra cui Zipkin, Jaeger, LightStep e Datadog. Gli operator controllano la frequeza di campionamento per generare le tracce. Questo permette di controllare il numero e la frequenza dei dati di tracing prodotti dalla mesh.

Proprio per questo motivo sono stati introdotti e studiati nel corso della tesi per offrire uno strumento in più al team DevOps nella ricerca e risoluzione degli errori che si possono presentare nelle applicazioni da rilasciare.

Jaeger Jaeger è un sistema di Distriuted tracing open source [8]. È utilizzato per monitorare e fare troubleshooting di sistemi distribuiti basati su microservizi, includendo:

- propagazione di Distributed context.
- monitoraggio di trasazioni distribuite.
- Root cause analysis.
- analisi di dipendenza dei serivizi.
- Ottimizzazione delle performance/latenza.

L'unità fondamentale per Jaeger è uno span, è caratterizzato da un nome, il tempo di inizio dell'operazione e la sua durata. Gli span possono essere nidificati per modellare relazioni di tipo causale.

kiali

Kiali aiuta a definire, validare e osservare le connessioni dei microservizi nella mesh di Istio [9]. Visualizza la topologia dei servizi nella mesh, offrendo visibilità sulle loro caratteristiche. Kiali offre visione sulle componenti della mesh a differenti livelli, partendo dal livello applicazione fino ai carichi di lavoro. Inoltre, è possibile includere Jaeger per offrire Distributed tracing.

Graph Il pannello Graph visualizza la topologia dei servizi nella mesh. Mostra le interazioni tra i servizi e a quale velocità viaggia il traffico tra loro, in modo da rendere subito visibile quale parti possono essere fonte di errore.

Il grafico mostra quali servizi sono configurati con Virtual Service e circuit breaker. Identifica eventuali problemi di sicurezza scovando il traffico che non è stato configurato come normale. È possibile visualizzare il traffico tra i componenti attraverso delle animazioni o guardando delle metriche.

Detail Views Kiali offre la possibilità di vedere una lista, filtrata, di tutte le definizioni dei servizi nella mesh. In ogni vista c'è lo stato, i dettagli, la sua definizione come file YAML e dei link alla visualizzazione nella mesh. Sono possibili vedere:

- Servizi
- Applicazioni
- Workload
- Configurazioni di Istio

Detail: Metrics Ogni vista offre una dashboard per visualizzare le metriche. Esse sono selezionate su misura in base al livello attenzionato, applicazione, workload o sevizio.

Per applicazioni e workload mostrano request, response metrics e il traffico in uscita e entrata. Per i servizi viene mostrato request e response time per traffico in entrata e uscita.

2.3 Spinnaker

Spinnaker è una piattaforma open-source, multi-cloud di continuous delivery che aiuta nel rilascio di cambiamenti nel software offrendo alta velocità e affidabilità [10].

È composto da una serie di microservizi indipendenti alcuni dei quali opzionali [11].

- **Deck**: l'interfaccia utente visibile da browser.
- **Gate**: gateway per l'API di Spinnaker, sia l'interfaccia utente che chi chiama l'API deve passare da questo componente.
- **Orca**: è l'orchestratore, gestisce le pipeline.
- **Clouddriver**: è il responsabile di tutte le mutating calls al cloud provider e per l'indexing/caching delle risorse rilasciate.
- **Front50**: è utilizzato per salvare in modo persistente i metadati delle applicazioni, pipeline, progetti e notifiche.
- **Rosco**: è utilizzato come bakery. Produce le immagini per vari cloud provider.
- **Igor**: serve per poter attivare continuous integration job in sistemi come Jenkins nelle pipeline.
- **Echo**: bus di eventi di Spinnaker, per poter inviare notifiche e agire come webhook in entrata per servizi come GitHub.
- **Fiat**: servizio di autorizzazione di Spinnaker.
- **Kayenta**: permette test automatici come lo stage canary analysis in Spinnaker.
- **Halyard**: servizio di configurazione di Spinnaker.

Spinnaker offre due set di funzionalità:

- application management
- application deployment

2.3.1 Application management

Si può utilizzare Spinnaker per gestire e vedere le risorse nel cloud. Una *Spinnaker application* modella il concetto di collezione di servizi(riferiti spesso come applicazioni o microservizi). I concetti principali utilizzati da Spinnaker per descrivere un servizio sono application, cluster e server. Load balancer e firewall descrivono come il servizio è esposto agli utenti.

Application Un'applicatione in Spinnaker è una collezione di cluster, che a loro volto sono collezioni di server group. Include, inoltre, i firewall ed i load balancer.

Un'applicazione rappresenta il servizio che si andrà a rilasciare usando Spinnaker, tutte le configurazioni di quel servizio e tutte le infrastrutture su cui è in esecuzione.

Cluster Si possono definire dei cluster, che sono l'unità logica di raggruppamento di server group in Spinnaker. Non sono da confondere con i Kubernetes cluster, che possono essere inclusi al suo interno.

Server Group I Server Group sono l'unità base per le risorse e identificano gli artefatti rilasciabili (es. Docker image, source location) e configurazioni base come il numero di instanze, autoscaling policy, metadata ecc. Questa risorsa è opzionalmente associata ad un Load Balancer ed un Firewall. Quando rilasciato, un Server Group è una collezione di istanze del software in esecuzione.

Load Balancer A ogni Load Balancer è associato un protocollo per il traffico in ingresso e un range di porte. Bilancia il traffico tra le istanze del suo Server Group. È possibile abilitare dei controlli sullo stato di salute dei load balance permettendo di stabile dei criteri sui cui basarsi e specificare gli endpoint su cui farli.

Firewall Un Firewall definisce quale traffico ha accesso alla rete. In sostanza è un set di firewall rule definite da un IP range e un protocollo di comunicazione (Es. TCP) oltre che ad un range di porte.

2.3.2 Application deployment

Utilizzando le caratteristiche degli application deployment di Spinnaker è possibile costruire e gestire una pipeline per fare continuous delivery.

Pipeline La pipeline è il costrutto fondamentale per il controllo del rilascio di applicazioni in Spinnaker. Consiste in una sequenza di azioni, nominati stage. Si possono passare parametri da uno stage all'altro lungo la pipeline.

Le pipeline possono partire con un'azione manuale oppure è possibile configurare un trigger automatico a seguito di un evento. Si possono configurare le pipeline per lanciare notificare (Es. email) agli interessati in vari punti della sua esecuzione.

Stage Uno stage in Spinnaker è un'unità della pipeline, descrive un'azione che la pipeline eseguirà. Si può creare una qualunque sequenza di stage, non è necessario seguire un certo tipo di ordine, anche se alcuni stage possono essere più comuni di altri. Spinnaker ne offre di vari tipi, quelli utilizzati nel corso della tesi sono:

- **Canary Analysis**: utilizzando il modulo *Kayenta* è possibile eseguire dei test automatici per verificare le prestazioni del canary confrontandoli con una baseline.
- **Kubernetes Deploy**: creare o modificare un oggetto Kubernetes da uno yaml manifest.
- **Find Artifacts From Resource**: prelevare un artefatto da una risorsa già presente in Kubernetes in modo tale da renderne disponibile l'utilizzo/lettura negli altri stage della pipeline.
- **Kubernetes Delete**: eliminare un oggetto Kubernetes.

Deployment stategies Spinnaker tratta i rilasci cloud-native come costrutti speciali gestendone l'orchestrazione come ad esempio fare veriche sul loro stato di salute, disabilitare server group e attivarne di nuovi. Spinnaker supporta strategia red/black, con rolling red/black e canary come rilasci attivi.

Capitolo 3

Fase di Test

Come detto nel capitolo introduttivo nella fase di Test si va a compilare il codice dell'applicazione, si stimano le risorse necessarie alla sua esecuzione per poi testarne le funzionalità [fig 3.1].

In questo capitolo viene spiegato come è stato affrontato tutto ciò nel corso della tesi, guardando le problematiche che si sono presentate e le soluzioni adottate.

3.1 Pipeline Jenkins

Nel corso della tesi sono state utilizzate delle Declarative Pipeline. In ogni Pipeline c'è una prima parte in cui vengono definite le variabili di ambiente, il repository da utilizzare e le credenziali d'accesso.

Per rendere la Pipeline customizzabile è stato creato un file txt in cui si possono impostare dei parametri per personalizzarne l'utilizzo. In questo caso parliamo principalmente delle credenziali d'accesso a Docker, sia per il repository pubblico che privato e le credenziali d'accesso a Kubernetes. Vengono anche prese da questo file le query con cui recuperare da Prometheus le metriche utili alla valutazione della versione attualmente in fase di test e quella live.

È stata creata un Pipeline principale definita con un Jenkinsfile che poi di volta in volta nei vari stage va a richiamare delle Pipeline che vanno ad eseguire il compito nello specifico.

Applicazioni di Test Ho utilizzato due applicazioni per eseguire dei test sulla Pipeline. La prima è un semplice sito web utilizzando Spring come framework. La seconda, come test più complesso ho utilizzato un sito di e-commerce, Prestashop.

L'utilizzo di due applicazioni non era strettamente necessario, ma ho seguito questo approccio per aiutarmi nello sviluppo della Pipeline in quanto le due applicazioni mi hanno permesso di provarla in vari casi limite.

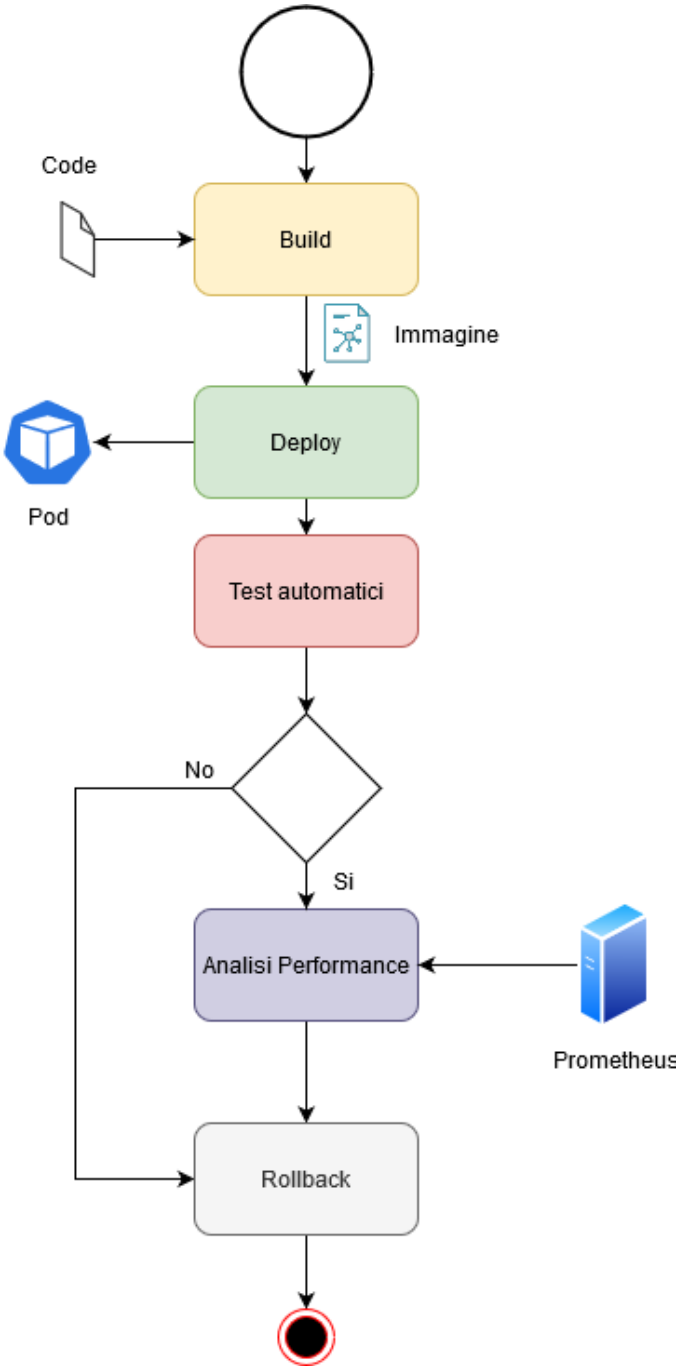


Figura 3.1: Schema della Pipeline della Fase di Test

L'applicazione più semplice, creata da me, mi è risultata più utile in quanto avendo accesso al codice è stato possibile provare la fase di build nei casi più estremi e verificarne il corretto comportamento, così come nella fase di classificazione in cui mi è stato più facile inserire bug o rallentamenti e verificare come l'algoritmo di classificazione si comportasse al cambianto delle performance.

Prestashop, invece, essendo un'applicazione più complessa mi è stata utile nel provare test più profondi in Selenium e JMeter e vedere il comportamento della Pipeline nell'interpretarne i risultati ottenuti.

3.2 Build e Deploy

3.2.1 Build

La Pipeline Build viene chiamata nel momento in cui la Pipeline principale arriva nello stage "Build". Ha il compito di compilare il codice e pushare la build in un docker registry privato.

Compilazione Il caso preso in esame è quella di un'applicazione la cui compilazione viene fatta attraverso l'utilizzo di *Maven*. Jenkins permette di incorporare Maven all'interno del suo applicativo rendendone possibile la gestione della versione, in questo modo non ha bisogno di affidarsi ad una installazione esterna con una versione che potenzialmente potrebbe non essere sufficientemente avanzata, o al contrario esserlo troppo per l'applicazione da compilare.

In fase di preparazione è possibile nelle impostazioni di Jenkins associare ad ogni versione di Maven un tag, è poi possibile utilizzarlo nella sezione 'tools' della Pipeline per indicare quale versione utilizzare.

Nel caso la compilazione fallisca la Pipeline 'Build' fallisce e ne viene riportato l'errore nella Pipeline principale che bloccherà la sua esecuzione. I log di compilazione vengono copiati nel log della Pipeline 'Build' in modo tale da poter scovare la sorgente dell'errore.

Se invece la compilazione non dà problemi si passa all'istruzione successiva.

Docker Registry Privato L'applicazione dovrà essere rilasciata in un cluster Kubernetes, quindi eseguita all'interno di un Container Docker. Questo presuppone la creazione di un'immagine che contenga l'applicazione, ed un repository privato da cui recuperarla.

Una considerazione va fatta al riguardo dell'aggettivo 'privato' che ho dato precedentemente al repository utilizzato in questa fase. Essendo questo un ambiente di Test l'ho pensato come un ambiente di sviluppo, quindi con build intermedie di cui non si vuole far conoscere l'esistenza al pubblico e quindi ci può essere la

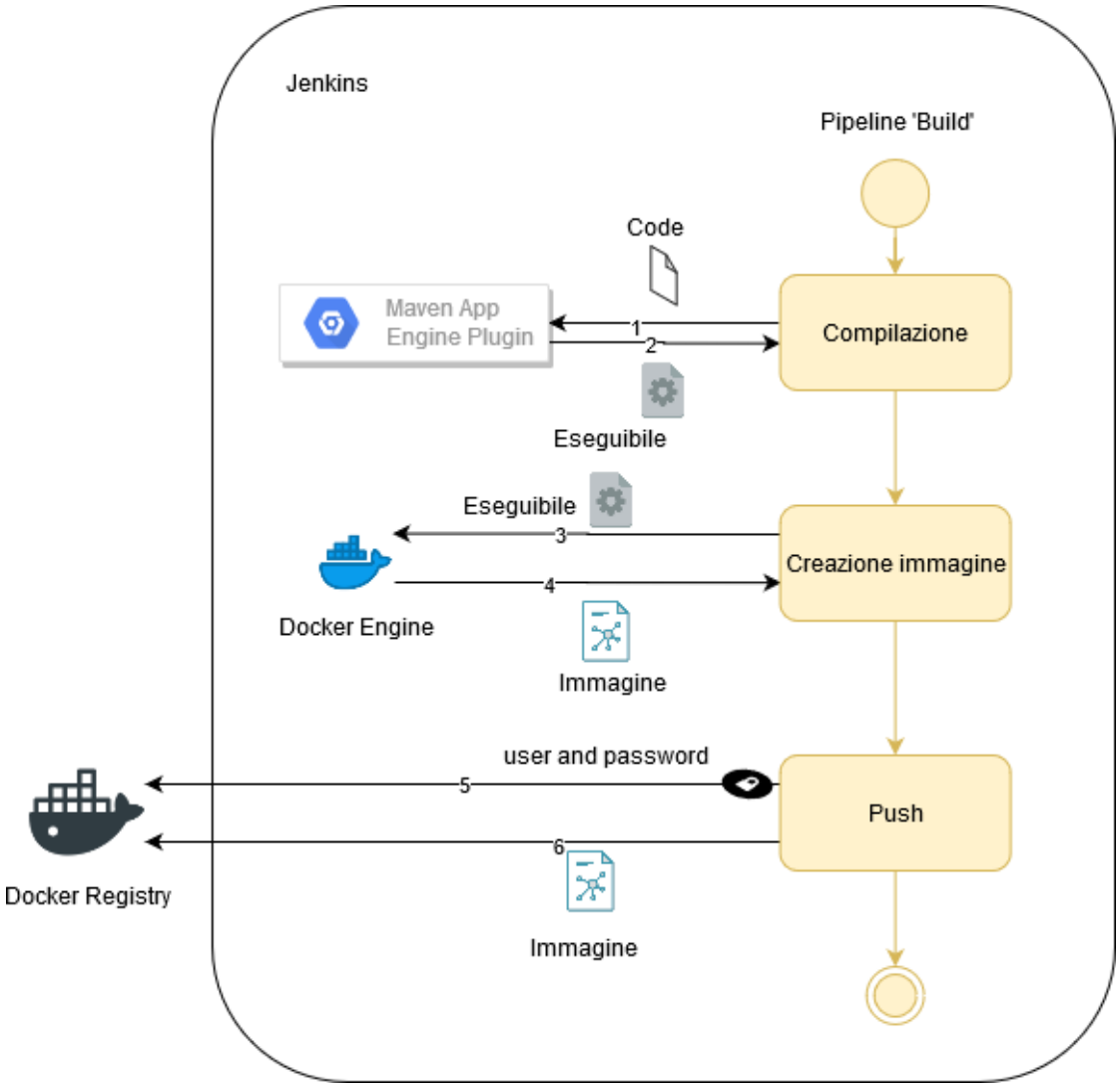


Figura 3.2: Schema della Pipeline 'Build'

volontà nell'avere un secondo registry su cui depositare queste immagini. Non esiste alcuna limitazione nel codice su che tipo di registry esso sia, è solo una convenzione data da me per distringuere quelli che potenzialmente potrebbero essere due registry totalmente diversi, cioè quello utilizzato in fase di Test e quello della fase di Produzione. Gestendo le credenziali d'accesso nel file txt i due registry possono anche coincidere.

Il servizio di Docker Registry viene lanciato in un container Docker. Precedentemente è stato creato un sistema di autenticazione Basic con Apache e viene creato un username e password per potervi accedere.

Per poter accedere al Docker Registry manca però che i certificati abbiano il trust nei nodi che dovranno utilizzarli, cioè nei nodi Kubernetes, in quanto dovranno accedere al registry per recuperare l'immagine da utilizzare per la creazione dei container, e nell'ambiente in cui è eseguito Jenkins dato che ha bisogno di accederevi per potervi fare il push dell'immagine. Ciò è dovuto al fatto che il certificato è self-signed in quanto non è stata scomodata alcuna autorità pubblica per poterlo creare.

Trust certificato per Jenkins Il certificato pubblico utilizzato al momento della configurazione tls del Docker Daemon viene semplicemente copiato nella root dei certificati del sistema utilizzato. Il passaggio per quanto sia semplice necessita comunque di certi accorgimenti, come il conoscere la posizione in cui mettere il certificato e i nomi da dare alla cartella o al certificato stesso in quanto può essere necessario indicare l'indirizzo IP e la porta del server per il quale verrà utilizzato.

Creazione Immagine e push sul Registry Il Dockerfile per creare l'immagine Docker deve essere presente nel repository Git. Con il seguente comando che utilizza il CLI di Docker viene creata l'immagine dell'applicazione ed associata ad il numero di versione e la repository, queste due informazioni vengono impostati nel file txt.

```
docker build <percorso> [-t repository:version]
```

A questo punto si esegue il login al registry privato con autenticazione attraverso nome utente e password. Nel comando viene deve essere specificato anche l'indirizzo IP e la porta su cui è in ascolto.

```
docker login --username <username> --password <password>
\ [indirizzo server con il registry]
```

Adesso è tutto pronto per fare il push sul Docker Registry.

```
docker push <repository:version>
```

3.2.2 Deploy

La Pipeline Deploy viene richiamata dalla Pipeline principale al momento dello stage 'Deploy'. Ha il compito di rilasciare nel cluster Kubernetes la versione di Test con tutti gli oggetti dell'API di Kubernetes necessari per renderla totalmente funzionante per le fasi successive.

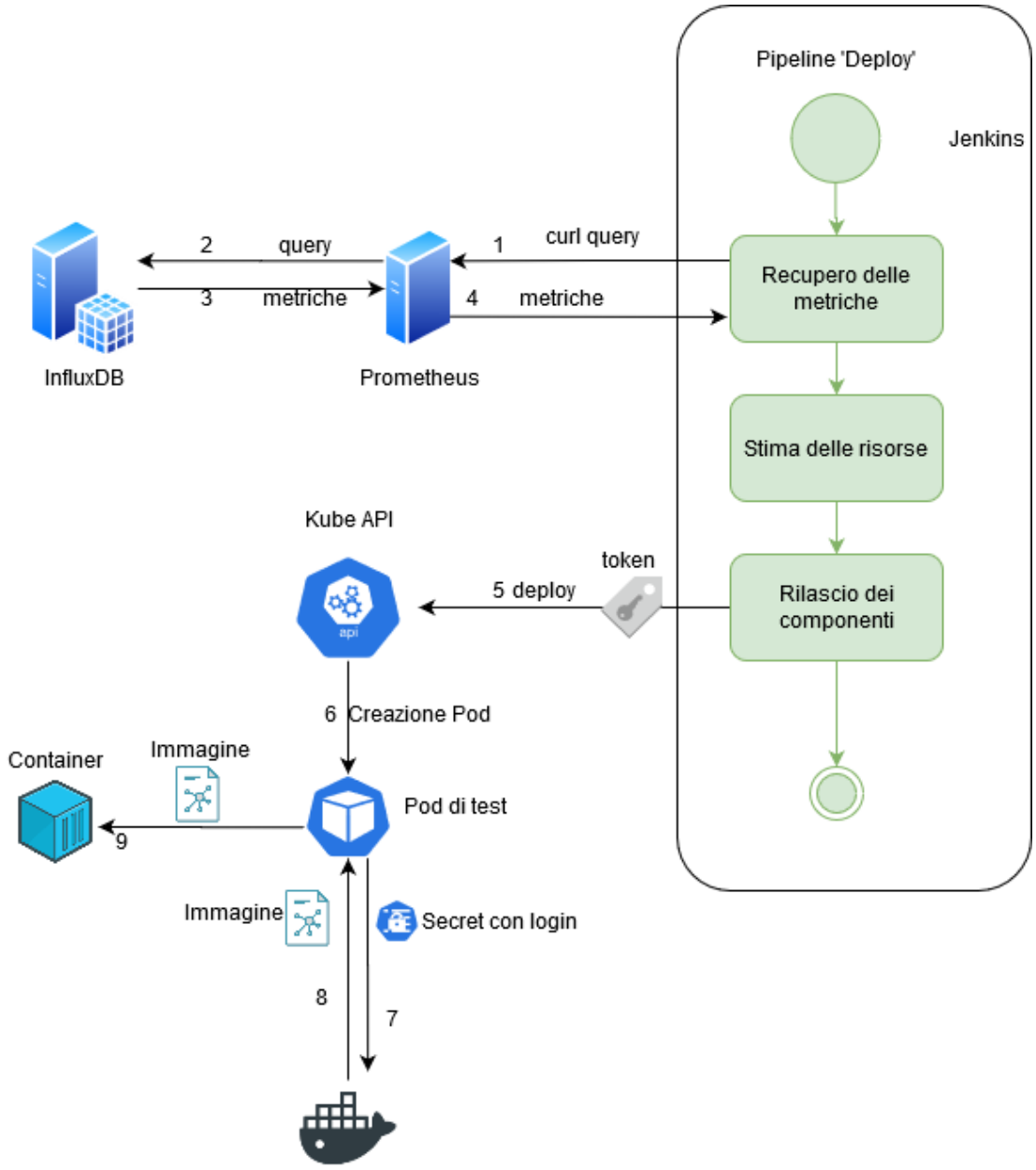


Figura 3.3: Schema della Pipeline 'Deploy'

Recupero immagine dal Docker Restry Kubernetes per poter rilasciare l'applicazione ha bisogno della sua immagine che in questo momento si trova nel Docker Registry privato a cui però ha bisogno di autenticarsi. Per poterlo fare si va a utilizzare un'astrazione di Kubernetes, i Secret. Essi permettono di conservare e gestire dati sensibili come ad esempio le password.

Il Secret può essere creato per essere visibile solo per un namespace oppure per tutto il cluster, nel caso della tesi è stato creato per essere utilizzato solo nel namespace che rappresenta l'ambiente di Test.

Se viene impostato come Secret per tutti i namespace di default non c'è bisogno di altro se non applicare lo yaml del Deploy dell'applicazione, se invece non è messo come default va aggiunta una sezione nel file yaml in cui si specifica di utilizzarlo per recuperare l'immagine.

Avendo questi elementi non appena Kubernetes avrà bisogno dell'immagine andrà a recuperare le credenziali dal Secret per poi utilizzarli per autenticarsi al registry. È inoltre necessario aver eseguito il trust dei certificati da utilizzare, la procedura seguita è la stessa fatta per Jenkins in ogni nodo del cluster Kubernetes, può cambiare l'ambiente di utilizzo ma il concetto è lo stesso.

Autenticazione su Kubernetes Adesso che è possibile per Kubernetes recuperare l'immagine dal registry manca per Jenkins la possibilità di connettersi alla sua API. Kubernetes ha bisogno di un processo di autenticazione prima di poter rilasciare dei componenti in un suo cluster. Esistono vari metodi per farlo, quello utilizzato nella tesi è attraverso l'utilizzo di un *ServiceAccount* Kubernetes.

I Service Account rappresentano degli utenti gestiti dall'API di Kubernetes. Sono associati ad uno specifico namespace, in questo caso è stato associato al namespace che rappresenta l'ambiente di Test. Inoltre sono associati a delle credenziali di accesso conservati in un *Secret*, che sarà montato anche nei Pod permettendo ai processi nel cluster di parlare con l'API di Kubernetes. Nel caso della tesi le credenziali contenute nel Secret corrispondono ad il certificato pubblico dell'API e il JWT (JSON Web Token) firmato dallo stesso certificato. Con il JWT sarà poi possibile autenticarsi con il Service Account.

Una volta che si ha l'utente è necessario indicare i privilegi che possiede, in Kubernetes questo ruolo viene svolto dai Kubernetes ClusterRole. Essi definiscono una classe di utenti ai quali vengono associati una determinata serie di azioni possibili all'interno del cluster Kubernetes.

Creato un ClusterRole per Jenkins viene fatto il binding del ClusterRole ed il ServiceAccount per associare Utente e Privilegi. Adesso il cluster Kubernetes è pronto ad accogliere le richieste di Jenkins da remoto.

Jenkins però non ha ancora conoscenza di come ad accedere a Kubenetes. Nel pannello relativo agli account gestibili da Jenkins se ne crea uno in cui si aggiunge il token utilizzato in precedenza.

Così configurato Jenkins nel momento in cui viene utilizzata una istruzione del CLI di Kubernetes si connetterà in remoto all'API utilizzando il token presente nell'account indicato, Kubernetes riconoscerà il token ed associerà alla chiamata il ServiceAccount. Se l'operazione richiesta è permessa dal ClusterRole associato allora essa verrà eseguita. In caso contrario l'operazione verrà rifiutata e la connessione chiusa.

Recupero metriche Nella fase di Deploy si devono anche impostare le risorse da utilizzare per l'applicazione che verrà rilasciata. In fase di sviluppo può essere complicato avere una stima precisa su quante risorse necessiti l'applicazione quindi è necessario farne una stima.

Normalmente questa procedura è fatta manualmente dal team DevOps, nella tesi si è provato a dare una soluzione al problema che non richieda aiuti esterni. La soluzione proposta è quella di prendere le risorse attualmente in utilizzo dalla versione precedente attualmente live ed aperta al pubblico applicandogli un fattore di margine. Il ragionamento che c'è dietro è quello che tra una versione e l'altra difficilmente ci sarà una richiesta di risorse molto diversa, il solo fattore di margine dovrebbe bastare ad avere delle risorse sufficienti.

Le metriche da recuperare sono quelle relative al consumo di RAM e CPU in una finestra temporale che è possibile selezionare dal file txt. Una volta prese viene fatta una media tra i vari campioni a cui viene applicato il margine per determinare le risorse base che servono all'applicazione.

Rilascio dell'applicazione Dai dati di RAM e CPU viene impostato il valore impegnato subito per l'applicazione mentre i valori limiti vengono impostati moltiplicando i valori base per un fattore uguale a 5. Questi dati vengono inseriti nel file yaml che contiene i componenti necessari all'esecuzione dell'applicazione.

A questo punto il file yaml viene applicato e i componenti vengono rilasciati nel cluster Kubernetes.

```
withKubeConfig( caCertificate: '', credentialsId: "${KUBE_CRED}",
    serverUrl: "${KUBE_URL}") {
    sh "kubectl apply -f <file yaml con i componenti necessari>"
}
```

È possibile vedere come la richiesta a Kubernetes nella Pipeline debba essere incapsulata in un costrutto 'withKubeConfig' che serve per specificare quale account Kubernetes (con all'interno il token) utilizzare per la richiesta all'API.

3.3 Test automatici

Con l'applicazione rilasciata nell'ambiente di test è ora possibile eseguire una serie di test che verificano la correttezza delle funzioni che svolge e di valutarne le performance sotto stress.

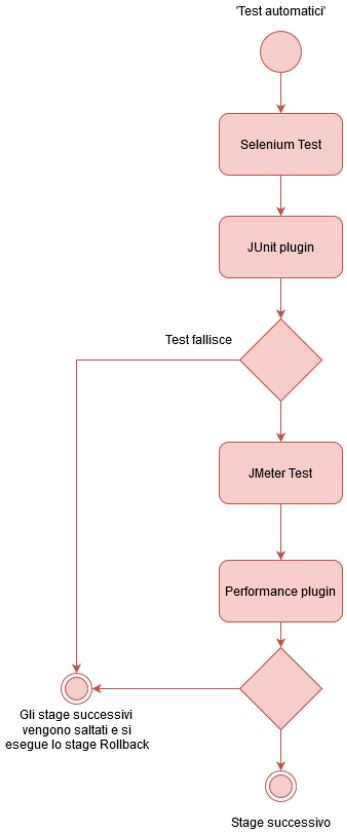


Figura 3.4: Schema della parte di Pipeline 'Test automatici'

3.3.1 Test Selenium

La Pipeline SeleniumTest viene avviata nel momento in cui la Pipeline principale raggiunge lo stage 'SeleniumTest'.

Adesso l'applicazione è stata rilasciata nell'ambiente di Test. Prima di continuare con altre analisi è importante capire se l'applicazione svolge il proprio lavoro, cioè c'è bisogno di verificare le sue funzionalità. Dato che le applicazioni prese in esame sono delle web application è stato scelto come framework di test *Selenium*. Selenium riproduce l'interazione di un utente con un browser, è possibile andare a verificare

la presenza di componenti html nella pagina e muoversi al suo interno. Il test Selenium viene fornito dallo sviluppatore e viene messo nel repository Git.

Analisi del Risultato Una volta eseguito il test è necessario un tool per leggerne il risultato. Il risultato del Selenium Test viene riportato in formato XML. Per analizzarlo è stato installato il plugin JUnit per Jenkins, che offre:

- Uno grafico in cui riporta il risultato del test nel tempo.
- Uno storico di tutti i test precedentemente effettuati.
- I report dei test precedenti in cui è possibile visualizzarene i log.

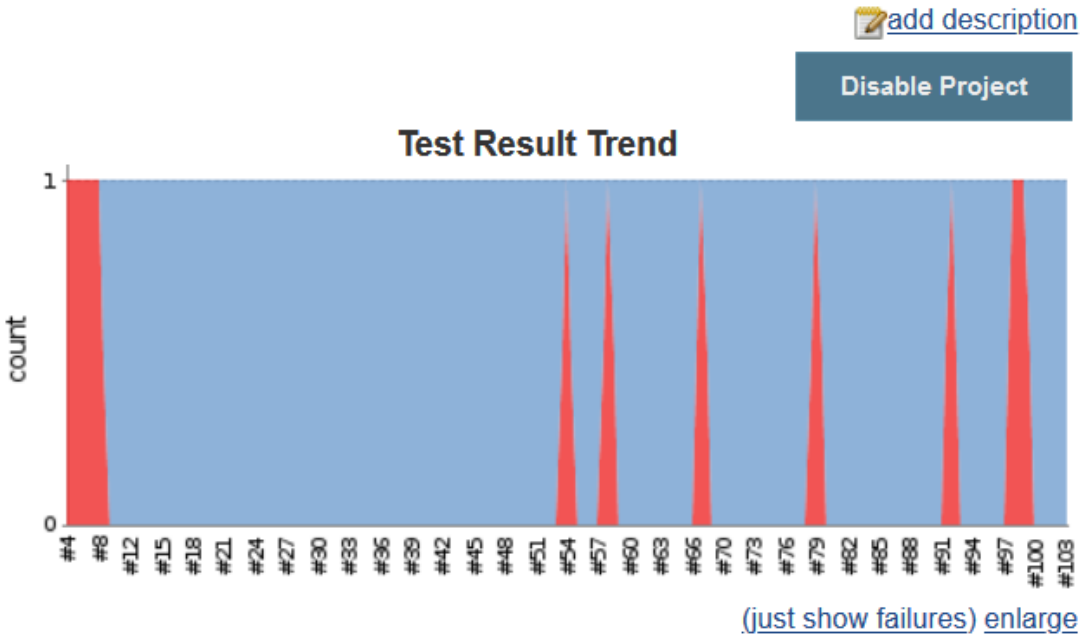
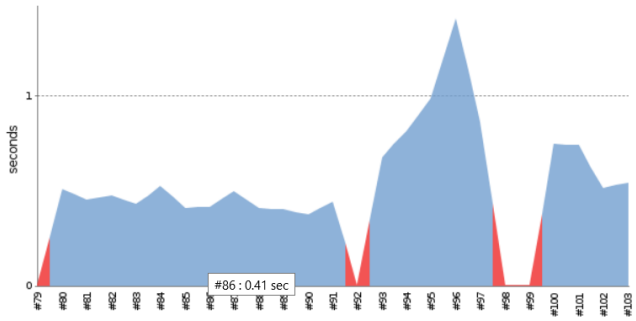


Figura 3.5: Risultati nel tempo dei test Selenium

Propagazione del Risultato Alla fine della Pipeline SeleniumTest il risultato viene riportato alla Pipeline principale. Se il test fallisce viene impostato un flag che farà proseguire la Pipeline fino alla fine ma senza eseguire gli altri stage tranne lo stage Rollback che pulirà l'ambiente di Test. Se invece dà un risultato positivo la Pipeline prosegue allo stage successivo. Il report del Test oltre che visualizzabile con JUnit viene stampato nei log della Pipeline Selenium Test.

History for contextLoads



show count

Build	Test Description	Test Duration	Test Result
test-selenium-new #103		0.54 sec	Passed
test-selenium-new #102		0.51 sec	Passed
test-selenium-new #101		0.73 sec	Passed
test-selenium-new #100		0.74 sec	**Fixed**
test-selenium-new #99		2 ms	Failed
test-selenium-new #98		5 ms	**Regression**
test-selenium-new #97		0.86 sec	Passed
test-selenium-new #96		1.4 sec	Passed

Figura 3.6: History con tutti i report dei test Selenium

Test Result

1 failures (±0)

1 tests (±0)
Took 9.2 sec.
add description

All Failed Tests

Test Name	Duration	Age
com.example.demo.DemoApplicationTests.contextLoads		
Stack Trace	2 ms	2
Standard Output		
Standard Error		

Figura 3.7: Dettaglio sul report di un singolo test nella historys

3.3.2 JMeter Test

Una volta che si è testata l'applicazione dal punto di vista funzionale è necessario testare il suo comportamento se sotto stress. Per svolgere questo compito si è scelto JMeter. JMeter è un software open source scritto in Java, creato per verficare il comportamento di un'applicazione ad un certo carico e misurarne le performance. Il test JMeter deve essere contenuto nel repository Git in cui si trovano i file della Pipeline. Una funzione collaterale è quella di creare del traffico verso l'applicazione in modo tale da poterne recuperare le metriche che verrano utilizzate nelle analisi fatte durante le fasi successive.

Analisi del Risultato I risultati salvati in formato JLT vengono analizzati con il comando *performanceReport* della Pipeline. Questa funzione è resa disponibile dal Performance Plugin. Con questo comando è possibile impostare il parser da utilizzare (in questo caso *JMeterPaser*) e si possono impostare vari parametri per interpretare il risultato. Ad esempio con *errorFailedThreashold* è possibile impostare il numero di chiamate fallite che determinano il fallimento del test. Si possono mettere più parametri insieme o utilizzarne solo uno.

Come con JUnit viene creato uno storico con i risultati ed i report dei test precedenti. Sono disponibili dei grafici sulla percentuale di errori delle chiamate fatte da JMeter, Il Response time ed il Throughput. Quelli del risultato attuale vengono anche stampati nei log della Pipeline TestJMeter.

Propagazione del Risultato Come fatto per il test Selenium il risultato del test viene riportato alla Pipeline principale. Se il test è andato a buon fine la Pipeline continua con le fasi successive, se invece fallisce vengono saltati tutti gli stage rimanenti tranne l'ultimo, Rollback, per pulire l'ambiente di Test.

3.4 Analisi delle performance

Ora che l'applicazione ha passato i test base devono esserene analizzate le *performance* e verficare che siano abbastanza buone per farla passare alla fase di Produzione.

È possibile scegliere se utilizzare un algoritmo di classificazione di machine learning per determinare se l'applicazione debba passare alla fase di Produzione oppure fare un'analisi matematica delle metriche utilizzando una funzione matematica. È stata data una doppia possibilità in quanto l'algoritmo di classificazione di machine learning viene contattato come servizio esterno su un server, esiste la possibilità che il gruppo di lavoro non abbia le capacità o non voglia utilizzare questa strada. Per questo motivo è stata introdotta la possibilità di avere una soluzione totalmente interna alla Pipeline che non necessità componenti esterni. Inoltre, una soluzione

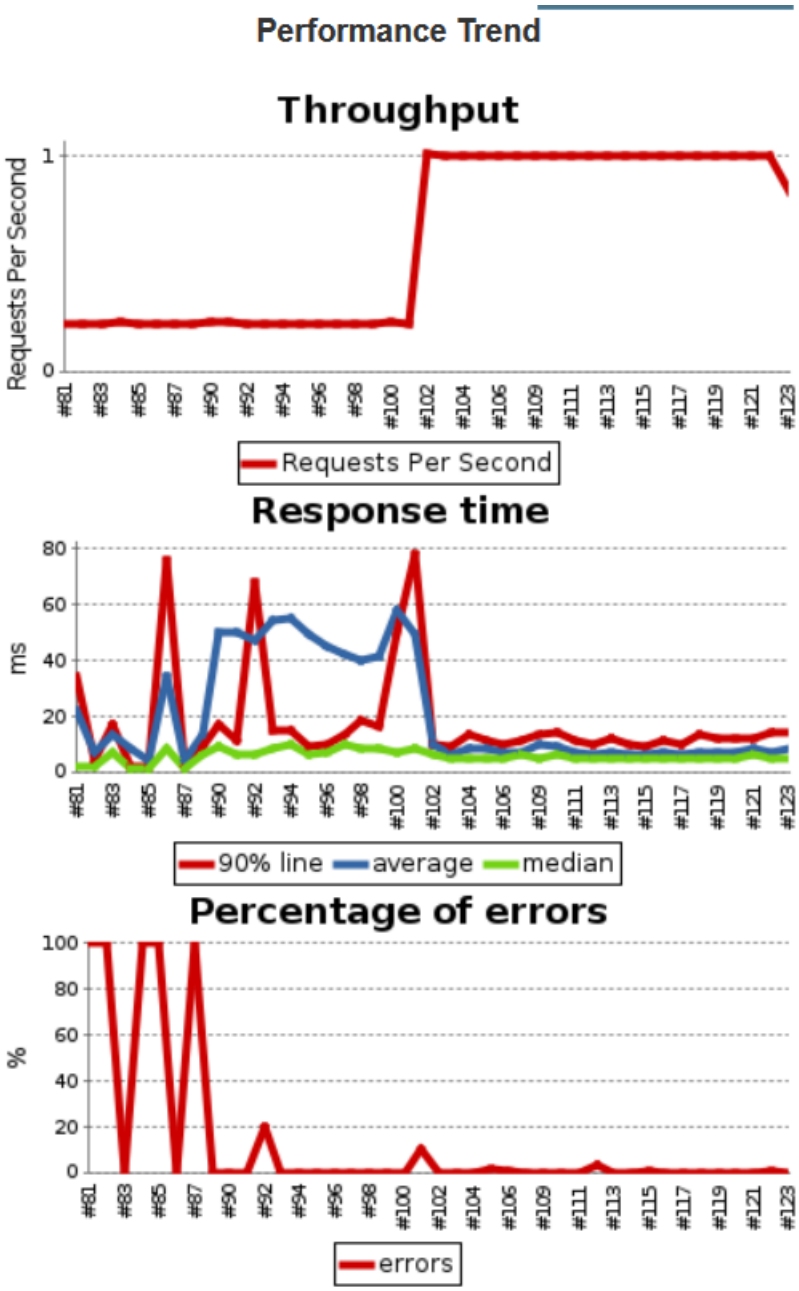


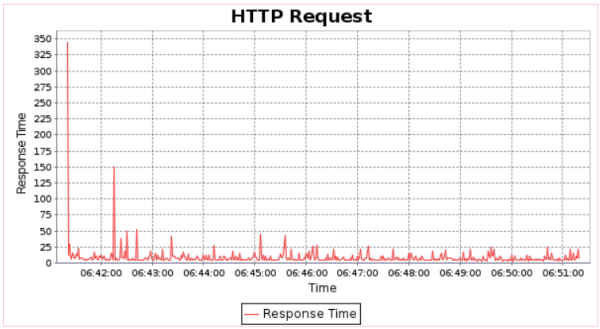
Figura 3.8: Risultati dei test JMeter nel tempo

Comparison with previous build

URI	Samples	Average (ms)	Min(ms)	Median(ms)	Line 90.0(ms)	Max(ms)	Http Code	Errors (%)	Average (KB)	Total (KB)
HTTP Request	500 -500	8 +1	3 0	5 0	14 0	345 -36	200 503,200	0.0 % -0.1 %	0.3 0.0	148.94 -148.97
All URIs	500 -500	8 +1	3 0	5 0	14 0	345 -36		0.0 % -0.1 %	0.3	148.94

Figura 3.9: Confronto i risultati dell'ultimo test JMeter ed il precedente

Trend report: jmeter.jlt
URI: HTTP Request



URI: HTTP Request

URI	Samples	Average (ms)	Min(ms)	Median(ms)	Line 90.0(ms)	Max(ms)	Http Code	Errors (%)	Average (KB)	Total (KB)
HTTP Request	500 -500	8 +1	3 0	5 0	14 0	345 -36	200 503,200	0.0 % -0.1 %	0.3 0.0	148.94 -148.97

Figura 3.10: Dettagli sull'ultimo test JMeter effettuato

con il machine learning diventa interessante nel caso in cui il numero di metriche da analizzare diventa grande, ciò può portare ad avere una classificazione più veloce ed affidabile rispetto all'analisi matematica delle metriche. Questa scelta viene fatta impostando il flag **ML** nel file txt. È possibile anche utilizzare la modalità con la funzione matematica in una prima fase per costruire il dataset per l'algoritmo di machine learning. Lo scenario sarebbe il seguente:

1. Gli sviluppatori sono sul punto di rilasciare un major update che aumenta le funzionalità cambiando il profilo delle performance.

2. Analisi delle metriche con la funzione matematica per i primi cicli in modo da poter utilizzare comunque la Pipeline ma contemporaneamente creare il dataset per l'algoritmo di machine learning.

3. Dataset abbastanza ampio per poter essere utilizzato nella fase di addestramento dell'algoritmo di machine learning.

4. Utilizzo della modalità con machine learning.

Il *numero di funzioni* (*N_FUNCTION*) è impostabile all'inizio della Pipeline e sarà contenuto nei campioni all'interno del dataset. Questo per introdurre una distinzione tra i profili di performance ritenuti buoni tra le varie versioni, in quanto può essere legittimo che una verisione che aumenta le funzione svolte dall'applicazione abbia consumi maggiori per alcune risorse.

È necessario selezionare delle metriche che permetteranno di capire come l'applicazione si sta comportando. Non si sono volute inserire le query per selezionare le metriche direttamente nel codice in modo tale da dare più libertà possibile a chi andrà ad utilizzare la Pipeline. La Pipeline è stata scritta per poter gestire un numero di query variabile e inseribili nel file txt. L'unica limitazione è nel nome che gli deve essere dato in quanto verranno lette seguendo una certa sintassi. Deve essere specificato il numero di query che si vuole utilizzare per recuperare le metriche nel campo **N_QUERY**.

Le query devono essere scritte successivamente con questa sintassi.

```
QUERY\_<1...n>\_NEW":"<QUERY>
```

Con <1...n> che è un numero sequenziale che parte da 1 ed arriva ad N_QUERY e ":" come separatore tra il nome e <QUERY> che è la query vera e propria da inviare a Prometheus.

È importante tenere a mente il numero associato alla query in quanto verrà utilizzato per accopiare la query con altri parametri che possono essere impostati a seconda della modalità scelta.

Questa fase di selezione delle metriche è intesa come passo iniziale dell'utilizzo della Pipeline e non più modificabile dopo un primo utilizzo per l'applicazione, pena la perdita del lavoro svolto fino a questo punto. Ciò è dovuto al fatto che con queste metriche si va a costruire il dataset per l'algoritmo di machine learning di classificazione, una volta cambiate le metriche i dati più vecchi che non possiedono queste metriche risultano inutilizzabili a meno che non si voglia implementare un metodo di costruzione degli attributi sul dataset prima di utilizzarlo per l'addestramento.

Per la tesi sono stati scelti:

- **Response time** perché è un parametro che può andare a inficiare sull'esperienza utente.
- **CPU** in quanto un consumo anamalo di CPU può indicare la presenza di errori o comunque rappresentare un costo giudicato eccessivo per la piattaforma.
- lo stesso discorso fatto per la CPU vale anche per la memoria **RAM** in quanto anche questa una risorsa fondamentale il cui aumento anomalo può essere giudicato negativamente.

3.4.1 Analisi Metriche

Nel caso in cui venga scelta questa opzione si passa ad un confronto tra le metriche dell'applicazione di test e quella attualmente live. Da questo confronto vengono calcolati dei valori che ne indicano la bontà del loro andamento, per poi essere combinati da una funzione per calcolare se l'applicazione nel totale ha passato la fase di test con un successo, con uno stato di warning oppure uno stato di error.

Dati iniziali Per poter utilizzare questa procedura sono necessari dei dati aggiuntivi rispetto a quelli base nel file txt.

- Query per recuperare le metriche della versione live, nella nomenclatura devono seguire la stessa numerazione delle query base in modo tale da poterle accoppiare nelle analisi successive, seguono una sintassi simile a quelle base, QUERY_<1...n>_OLD.
- Per ogni metrica, seguendo sempre la stessa numerazione, si deve impostare la differenza tra la versione di test e live, in percentuale, oltre il quale si determina un peggioramento delle performance tale da far scattare il flag di error e warning.
- Per ogni metrica va selezionato quale andamento ne indica il peggioramento:
  - '+' se un aumento della metrica ne indica un peggioramento.

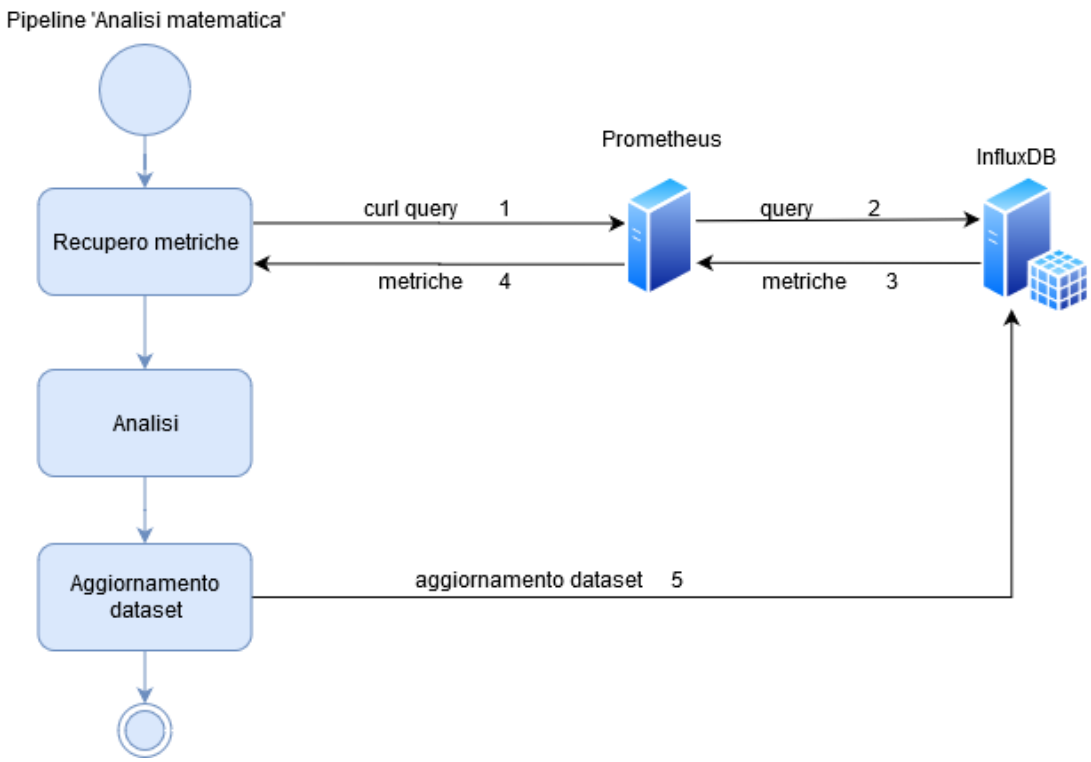


Figura 3.11: Schema della Pipeline nella parte 'Analisi Metriche'

  - '- se una diminuizione della metrica ne indica un peggioramento.
  - '+-' o '-+' se sia un aumento che una diminuizione della metrica ne indica un peggioramento.

- Per ogni metrica va impostato il peso che un suo peggioramento ha sulla valutazione totalmente.

- I valori limite della funzione che calcola le performance dell'applicazone oltrepassati i quali vengono settati lo stato di error o warning.

Calcolo valori Viene calcolata la media per ogni metrica sia per la versione di Test che per la sua controparte nella versione live. La differenza in percentuale tra le due ne diventa il *value*. Valori positivi del value ne determinano un peggioramento. In base all'andamento selezionato nel file txt si effettua l'inversione del segno del value.

- '+' non si inverte il segno in quanto un incremento della metrica sta a significarne un suo peggioramento.

- '-' si inverte il segno.

- '+-' o '-+' si inverte il segno se value ha un valore negativo, in questo modo si esprime un peggioramento della metrica nel discostarsi da quella live sia nel caso di aumento che peggioramento.

I vari value vengono confrontati con i limiti di error e warning, nel caso in cui vengano oltrepassati viene stampato un messaggio in cui si avverte l'utente.

Si è scelto di non far fallire il test sull'intera applicazione nel caso in cui una metrica passi il limite di error in quanto si è cercato di dare più importanza alla valutazione delle metriche nel loro insieme tenendo conto dei relativi pesi.

Risultato della classificazione La funzione viene applicata sia per calcolare il limite di error che di warning. In entrambi i casi ogni value viene diviso per il limite, moltiplicato per il proprio peso e diviso per il peso totale. Ognuno di questi valori viene sommato ed il risulto è il valore della funzione.

$$func = \sum_{i=1}^{N\_QUERY} \frac{value_i \times WEIGHT_i}{PERC_i \times TOTAL\_WEIGHT}$$

A questo punto si aprono tre scenari.

- Se il valore della funzione relativa all'error è superiore al limite impostato nel file txt allora viene settato a true un flag che indica il fallimento del test, *errorFlag*.

- Se il valore della funzione relativa al warning è superiore al limite impostato nel file txt allora viene settato a true un flag che indica un potenziale errore nell'applicazione, *warningFlag*.

- Se nessuna funzione passa il valore limite all'ora l'applicazione viene considerata buona per passare alla fase successiva.

Aggiornamento dataset Viene creato un campione che contiene il valore di ogni metrica per un determinato timestamp, il numero di funzioni svolte dall'applicazione e la classe. La classe viene impostata ad 1 nel caso in cui l'applicazione passa alla fase successiva, 0 nel caso contrario.

Ogni campione viene inviato in un database su InfluxDB che contiene il dataset da utilizzare per la modalità in machine learning.

3.4.2 Machine Learning Classification

È possibile utilizzare un servizio esterno con un algoritmo di classificazione di Machine Learning che classificando i vari campioni dovrà decidere se l'applicazione è abbastanza buona per poter passare alla fase successiva o no. Come esempio di algoritmo di machine learning per la tesi si è scelto un classificatore con una fase di learning di tipo *supervisionato*.

Ciò vuol dire che vengono forniti all'algoritmo una serie di dati di addestramento (dataset), questi dati contengono sia quello che sarà l'input che verrà dato alla macchina (in questo caso i campioni delle metriche) sia il risultato (la classe di appartenenza). Un algoritmo di questo tipo andrà a stimare la funzione incognita di classificazione attraverso l'analisi dei dati di addestramento. La classificazione che deve essere fatta è di tipo binario (1/0), '1' il profilo delle performance è ritenuto sufficiente per passare alla fase di produzione, '0' invece se no.

Server Machine Learning Per prima cosa mi sono posto il problema di dove andare a posizionare il classificatore, cioè se inserirlo all'interno del Pod di Jenkins oppure presupporlo come un servizio esterno. La strada scelta è stata quella di un servizio esterno per una serie di motivi.

Avendo questa divisione netta tra il Pod di Jenkins e il classificatore è possibile modificare l'algoritmo del classificatore o il suo dataset senza dover modificare e riavviare il Pod di Jenkins. L'indirizzo IP e porta del classificatore vengono presi dal file txt in modo tale da poterli cambiare a piacimento senza dover modificare la Pipeline. Lo svantaggio principale è quello di non avere la portabilità di Jenkins e classificatore in una soluzione unica.

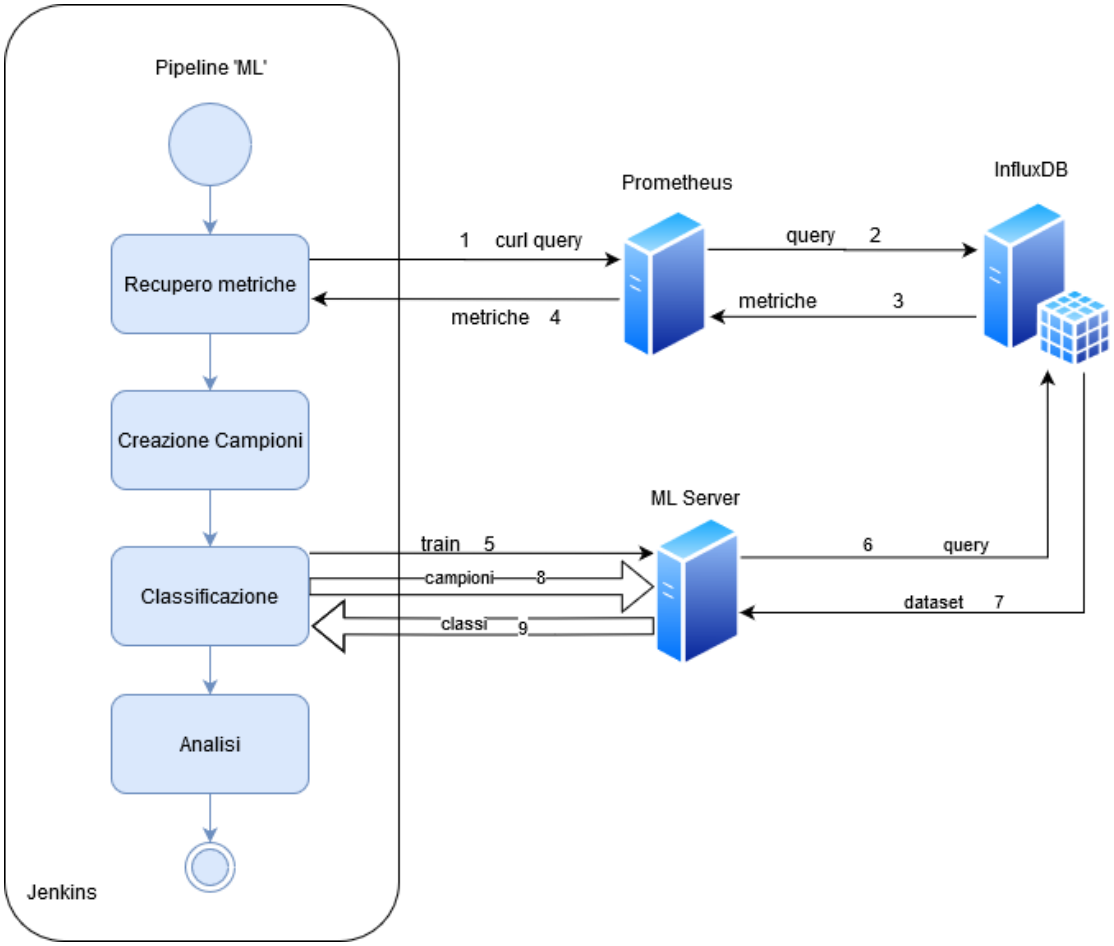


Figura 3.12: Schema della parte di Pipeline 'ML Classification'

È stato creato un server in python sempre in ascolto per ricevere i campioni da classificare. Le funzioni che deve avere il server per poter essere utilizzato dalla Pipeline sono due.

La prima è che i campioni devono essere ricevibili tramite una http POST sull'indirizzo IP e porta. Il nome associato ai dati nella POST deve seguire una certa sintassi per poter comunicare correttamente con la Pipeline.

```
--data QUERY<1...n>=<dato>
```

I campioni contengono al loro interno un dato per ogni metrica recuperata dalla Pipeline che hanno lo stesso timestamp. Il numero associato al nome del dato deve essere uguale a quello scritto nel file txt della Pipeline. Solo in questo modo è possibile inviare tutti i dati nell'ordine corretto dalla Pipeline al server.

La seconda funzione è che deve essere possibile dall'esterno attivare l'addestramento dell'algoritmo prendendo l'ultima versione aggiornata del dataset dall'InfluxDB. Deve poter essere fatto con una POST sull'indirizzo del server nel path '/train'.

XGBoost Per l'algoritmo di classificazione di Machine Learning nella tesi viene utilizzata la libreria XGBoost. "XGBoost è una optimized distributed gradient boosting library" [12]. Offre un algoritmo di tree boosting parallelo.

Algoritmo Un decision tree, *albero decisionale*, si basa principalmente su tre concetti.

- **Nodo**: test di un attributo rispetto ad un valore.
- **Ramo**: risultato dell'ultimo test che viene riportato ad un altro nodo o foglia.
- **Foglia**: nodo terminale che predirrà il risultato.

In pseudocodice l'algoritmo di addestramento di un modello è il seguente.

1. un attributo viene messo come radice dell'albero.
2. Il training set viene diviso in dei subset in modo tale che ognuno contenga dati con lo stesso valore per un attributo.
3. Ripetere il punto 1 e 2 finche ogni subset raggiunge un nodo foglia in ogni ramo dell'albero.

Un algoritmo di machine learning, come ad esempio un albero decisionale, semplicemente addestra un singolo modello sul dataset fornito e lo utilizza per farne predizioni. Il *boosting* ha un approccio di tipo iterativo. Molti modelli vengono

combinati insieme per creare quello finale. Invece di addestrare i modelli isolati uno rispetto all'altro, vengono fatti in successione, con ogni nuovo modello viene creato per correggere gli errori fatti da quello precedente. I modelli vengono aggiunti fino a che non è più possibile migliorarli. Il *gradient boosting* si basa sul predire i residui, cioè gli errori, dei modelli precedenti.

Risultato del classificatore Per prima cosa viene eseguito l'addestramento dell'algoritmo prendendo l'ultima versione del dataset. Dopo di che i risultati delle metriche vengono ottenuti facendo delle query a Prometheus. Per ogni metrica vengono presi i valori con lo stesso timestamp e impachettati in un 'campione' a cui viene aggiunto anche il numero di funzioni svolte dalla funzione. Se anche una sola metrica non ha un valore per quel timestamp il campione viene scartato. A questo punto ogni campione viene inviato al classificatore che deciderà se classificarlo come 1 o 0 come scritto in precedenza. Dopo che tutti i campioni sono stati classificati sono possibili tre casi.

- **Error**: se i campioni classificati come 1 sono sotto il 70% allora l'applicazione viene ritenuta non valida per passare alla fase successiva e viene settato a true un flag che ne indica il fallimento, *errorFlag*.

- **Warning**: se i campioni classificati come 1 sono tra il 70% ed il 90% allora l'applicazione viene ritenuta valida per passare alla fase successiva anche se viene settato a true un flag che indica che comunque un problema ci può essere, *warningFlag*.

- **Success**: se i campioni classificati come 1 sono oltre il 90% allora l'applicazione viene ritenuta valida per la fase successiva e viene impostato un flag che ne indica il successo.

3.4.3 Rollback

In base ai flag settati in precedenza, in questo stage, avviene la decisione se mandare l'applicazione in fase di produzione o meno. Al termine di ciò viene pulito l'ambiente di test per non lasciare tracce che potrebbero danneggiare una successiva esecuzione della Pipeline.

Pulizia dell'ambiente di Test Sia che l'applicazione passi alla fase di produzione sia che venga fatta ritornare agli sviluppatore, l'ambiente di Test viene pulito. Come fatto per lo stage Deploy viene effettuata la connessione al cluster Kubernetes ed utilizzando il file yaml del Deploy vengono eliminati i componenti rilasciati.

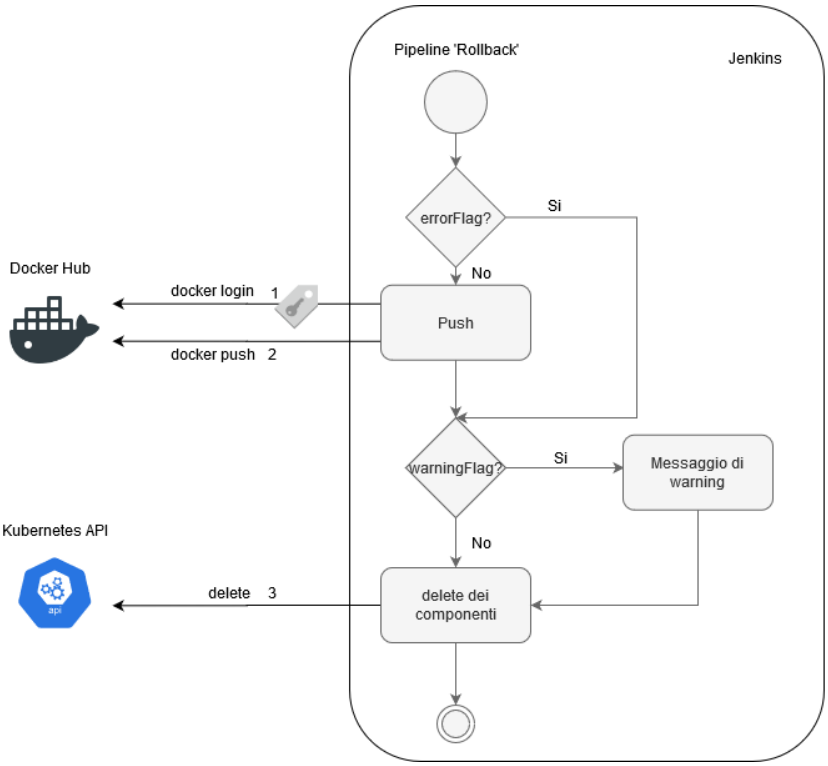


Figura 3.13: Schema della Pipeline nella parte 'Rollback'

Decisione finale Se l'*errorFlag* è uguale a true allora la Pipeline stampa un semplice messaggio di errore e si conclude. Se invece ad essere a True è l'*warningFlag* l'applicazione passa alla fase successiva e viene stampato un messaggio per avvisare che il risultato dell'analisi non è ottimale. Se invece nessuno dei flag è uguale a True l'applicazione passa semplicemente alla fase succesiva.

Nel caso specifico della tesi ho deciso che il passaggio alla fase di Produzione coincide con il push dell'immagine dell'applicazione su *Docker Hub*. Questo perché come verrà spiegato nel capitolo successivo è stato impostato un trigger sui push su DockerHub per far avviare la Pipeline della fase di Produzione. I dati di login, del numero di versione e del branch sono recuperabili dal file txt. Per effettuare il push all'atto pratico, oltre alle credenziali d'accesso differenti, non cambia nulla rispetto alla procedura eseguita nello stage Build.

DockerHub Docker Hub è un servizio offerto da Docker per trovare e condividere immagini Docker. È il docker registry che di base è utilizzato nell'ambiente di Docker.

Il suo ampio utilizzo è dovuto per le funzionalità che offre.

- È possibile fare il push e pull delle immagini sulle repository create dall'utente.
- Si può gestire un sistema di accesso privato alle repository ed alle immagini.
- Sono messe a disposizione della comunità delle immagini ufficiali curate da Docker per Sistemi Operativi (Utilizzate principalmente come punto d'inizio per le immagini create dagli utenti), veloci soluzioni per l'utilizzo dei più popolari linguaggi di programmazione, DB ecc. Inoltre è garantito che gli update di sicurezza di queste immagini siano fatte il prima possibile. Sono messe a disposizione inoltre diverse versioni per ogni immagine in modo tale da poter scegliere la soluzione più appropriata.
- I Venditori esterni a Docker possono mettere a disposizione immagini di alta qualità disponibili al pubblico, le immagini certificate includono un servizio di supporto ed è garantita la compatibilità con Docker Enterprise.
- È possibile fare la build di immagini direttamente da GitHub e farne il push su Docker Hub.
- E, utilizzato nel corso dello sviluppo dell'elaborato di tesi, sono disponibili dei Webhook per integrare la possibilità di creare dei trigger in seguito al push su una repository.

Capitolo 4

Fase di Produzione

Come detto nel capitolo introduttivo in questa fase si rilascerà l'applicazione al pubblico. Non è più possibile eseguire test approfonditi sull'applicazione in quanto ora è a disposizione degli utenti e ciò potrebbe portare ad un peggioramento della loro esperienza d'uso. L'unica cosa che è possibile fare è seguire una strategia di rilascio che permetta di continuare ad analizzare la situazione e intervenire sull'applicazione senza andare ad impattare sull'esperienza degli utenti. A questo scopo si è seguito la strategia del *Canary Deployment*.

Canary Deployment [13] Il canary deployment consiste in un rilascio 'lento' della nuova versione di un'applicazione. Quello che si cerca di fare è ridurre il rischio di rilasciare una nuova versione che può impattare negativamente sul carico di lavoro. Per raggiungere questo obbiettivo si va a rilasciare la nuova versione rendendola visibile agli utenti in modo incrementale. Ogni volta che la nuova versione passa i test sulle performance si va a incrementare la percentuale di traffico che riceverà fino al raggiungimento del 100% del traffico.

È necessario avere un load balancer che permetta di dividere il traffico ed inviarne una parte sulla nuova versione. Questo non è possibile farlo utilizzando solo Kubernetes, per questo è stato introdotto anche Istio che come indicato nella sezione apposita grazie all'utilizzo delle Destination Rule e Virtual Service permette di implementare un canary deploy.

Nomenclatura Nel corso di questa fase andrò ad utilizzare dei termini per identificare le varie versioni in gioco.

- **Pillar**: indica la versione precedente attualmmente disponibile al pubblico.
- **Canary**: indica la nuova versione che è sottoposta al canary deployment.

- **Baseline**: per avere un confronto tra le metriche del Canary e del Pillar ho voluto introdurre la Baseline. L'applicazione ha il numero di versione uguale al Pillar, ma riceverà la stessa percentuale di traffico del Canary, in questo modo si potrà avere un confronto tra due versioni che però sono sottoposte allo stesso carico di lavoro.

4.1 Perché Spinnaker

L'introduzione di Spinnaker in questa fase è dovuta principalmente all'utilizzo del modulo *Halyard*. Esso permette una valutazione delle performance del Canary eseguendo dei *Canary Analysis*.

Kayenta Kayenta è un modulo di Spinnaker che permette di implementare un Canary Analysis automatico. In questa analisi si vanno a confrontare le metriche del Canary e della Baseline, il suo risultato è un giudizio sulle performance delle due versioni indicandone la migliore. Se le performance del Canary risultano migliori rispetto a quelle della Baseline allora il giudizio finale sarà positivo, nel caso contrario invece sarà negativo.

L'analisi viene eseguita in due fasi:

- In un primo momento Kayenta raccoglierà le metriche sia dalla Baseline che dal Canary, nel caso della tesi esse vengono raccolte attraverso una query a Prometheus che andrà a prelevarle dal database nell'InfluxDB.

- Nella seconda fase le metriche della Baseline e del Canary vengono confrontate e si esprime un giudizio. Questa parte viene eseguita da un'ulteriore modulo interno a Kayenta, il *Decisore*.

È possibile fornire un'implementazione custom del Decisore, quella utilizzata nella tesi è quella offerta di default, il **NetflixCAJudge**. Il suo funzionamento è sintetizzabile in 4 fasi.

- **Validazione dei dati**: all'inizio ci si assicura che in ogni metrica ci siano dei campioni da analizzare, questo controllo avviene sia per le metriche della Baseline che per quelle del Canary. Se non sono presenti la metrica viene etichettata con NODATA. Al momento di eseguire l'analisi le metriche etichettate così vengono saltate passando alla metrica successiva. Dato che può non essere un errore che non ci siano dati per una metrica l'analisi non ne darà un giudizio negativo.

- **Pulizia dei dati**: in questa fase vengono preparati i dati ricevuti. Lo scopo principale è quello della gestione dei campioni mancanti. È possibile

impostare più strategie da adottare in questi casi in base al tipo di metrica. I dati mancanti rappresentati come 'NaN' possono essere sostituiti con zero o rimossi. Facoltativamente questa fase può eliminare gli outlier prima di passare all'analisi vera e propria.

- **Comparazione delle metriche**: qui avviene l'analisi vera e propria confrontando le metriche del Canary con quelle della Baseline. Il risultato è una classificazione per ogni metriche che indica se ci sono differenza significative tra i due. Ogni metrica viene classificata come Pass (se ok), High (differenza elevata), Low (differenza minima). Il test utilizzato è di tipo statistico non parametrico, in particolare dopo un'analisi del codice del Decisore. disponibile su GitHub, è risultato che utilizza il test di Wilcoxon-Mann-Whitney.
- **Computazione dello score**: ora che ogni metrica è stata classificata viene computato lo score che rappresenta quanto il Canary e la Baseline sono simili, lo score viene calcolato per ogni gruppo come il numero totale di metriche etichettate come 'Pass' diviso il numero totale di metriche nel gruppo. In base al peso del gruppo e del numero di gruppi viene dato lo score al canary analysis, se oltrepassa il valore minimo impostato allora il giudizio è positivo, nel caso contrario il giudizio è negativo.

Canary Analysis Un Canary Analysis è un tipo di stage della Pipeline di Spinnaker che permette di eseguire un test sulle prestazioni della versione Canary rispetto alla Baseline. È possibile impostare vari parametri per personalizzare l'analisi.

Per la tesi è stato impostato un intervallo di tempo per il test in cui raccogliere le metriche di *30 minuti*, questo perché è il tempo minimo consigliato dalla documentazione per avere un'analisi significativa se si utilizza il Decisore di default. Tempi più lunghi permettono di eseguire più analisi in successione ed avere analisi su un numero maggiore di campioni.

L'applicazione al momento di un canary analysis è stata appena rilasciata o comunque ha appena avuto un incremento di traffico, dai vari test eseguiti ho notato che necessita di almeno 5 minuti di *warmup* prima che le metriche si stabilizzino su valori stabili. Facendo partire subito l'analisi nei primi campioni si presentano numerosi outlier per poi stabilizzarsi. Perciò è stata ritardata di 5 minuti la raccolta delle metriche prima di effettuare il test.

Anche se per non avere tempi di esecuzione eccessivamente lunghi non è stato sfruttato nell'elaborato finale, è possibile far eseguire più analisi all'interno del tempo stabilito dividendolo in più slot temporali. Questi slot ovviamente devono sempre essere maggiore di 30 minuti per rendere ogni analisi significativa.

È stato impostato un *tempo di campionamento* di 15 secondi per le metriche in modo tale da farlo coincidere con il tempo di campionamento di Prometheus.

Una feature del Canary Analysis è quella di poter scegliere se prelevare le metriche da una certa finestra temporale fissata nel passato impostandone tempo di inizio e tempo di fine, oppure andare a campionare le metriche *in tempo reale*. La scelta seguita nella tesi è quella di utilizzare le metriche in tempo reale in quanto l'inizio di una nuova analisi nella Pipeline sviluppata coincide con un incremento di traffico verso il Canary e la Baseline, ciò comporta che le metriche utili all'analisi non possono che essere create in quel momento.

Infine è possibile impostare due parametri per determinare il giudizio dato dal Decisore:

- **Marginal**: se lo score di uno dei test del Decisore è sotto questa soglia al Canary Analysis viene assegnato un giudizio negativo.

- **Pass**: se lo score dell'ultimo test del Decisore è sopra questa soglia allora al Canary Analysis viene assegnato un giudizio positivo, se invece lo score dell'ultimo test è inferiore a questa soglia il giudizio assegnato è negativo.

Le informazioni relative al Canary Analysis vengono salvate in un *Canary Report* e saranno presenti:

- Un **grafico** dell'andamento della metrica per ogni coppia, metrica del Canary e la sua controparte della Baseline.

- Un **grafico** in cui si mostra per ogni metrica la differenza tra la metrica del Canary e la sua controparte della Baseline.

- L'**etichetta** assegnata dal Decisore per ogni metrica.

- La **'deviazione'** che ha ogni metrica del Canary rispetto alla sua controparte della Baseline.

Nel Canary Analysis per poter selezionare le metriche da prelevare ha bisogno di un ulteriore astrazione di Spinnaker, i *Canary Config*.

Canary config Un Canary Config è un'astrazione di Spinnaker con il quale è possibile selezionare le metriche da utilizzare per un Canary Analysis, se dividerle in gruppi ed il peso che ognuno avrà nella valutazione finale.

È possibile selezionare vari provider da cui reperare le *metriche*, quello utilizzato nel corso della tesi è Prometheus. Per ogni metrica viene specificato quale andamento ne determina un peggioramento (incremento, decremento oppure entrambi i casi).

Sono possibili due strade per specificare come prelevare le metriche:

Figura 4.1: Canary Report di un Canary Analysis

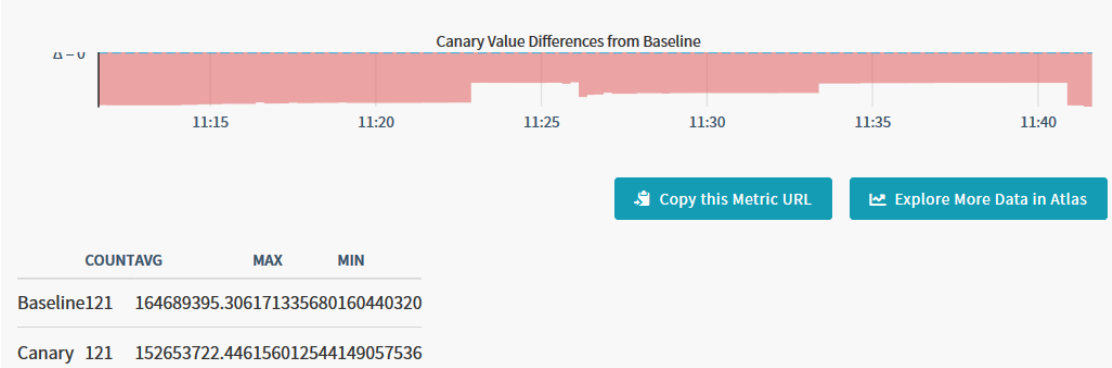


Figura 4.2: Continuo del Canary Report di un Canary Analysis

- Nel primo caso si segue una procedura guidata in cui si seleziona il provider, la metrica da utilizzare, i filtri da adottare e se eventualmente sono necessarie delle operazioni di raggruppamento (Somma ecc.)

- La seconda modalità, che è quella utilizzata nel corso della tesi, è quella di scrivere direttamente una query in PromQL da utilizzare per recuperare la metrica da Prometheus, l'unica limitazione riscontrata è che il risultato ottenuto dalla query non deve contenere label. È possibile ottenere questo risultato con dei piccoli accorgimenti nella formulazione della query.

Per ogni metrica va selezionata la strategia da adottare nel caso di valori 'NaN', è possibile eliminare i valori o sostituirli con zero.

Le metriche possono essere raggruppate in *gruppi*. Ogni gruppo può rappresentare un certo aspetto delle performance dell'applicazione. Ad ognuno viene assegnato un peso, l'unica limitazione è che la somma dei pesi di tutti i gruppi deve essere uguale a 100.

Connessione a Prometheus Per potersi connettere a Prometheus è necessario per prima cosa abilitarlo come storage delle metriche su Spinnaker. Successivamente si deve creare un account in cui si specifica l'indirizzo IP e la porta da utilizzare per potersi connettere. In questo modo, quando ci saranno da recuperare le metriche da Prometheus, Spinnaker utilizzerà in automatico quell'account per conoscere l'indirizzo su cui inviare la query.

Pipeline di lavoro La Pipeline di lavoro della fase di produzione è schematizzabile in tre fasi principali.

- **Inizializzazione**: dove vengono create tutte le strutture e gli oggetti necessari per l'esecuzione della Pipeline, in questa fase vengono rilasciati la Baseline ed il Canary.

- **Canary Analysis**: qui avviene il loop del canary deploy dove ad ogni incremento del traffico verso il Canary viene analizzato l'andamento delle sue performance rispetto a quelle della Baseline.

- **Termine**: in questa fase si decide se promuovere o meno il Canary come nuovo Pillar, per poi pulire l'ambiente facendolo tornare ad uno stato stabile.

Connessione a Kubernetes Spinnaker nel corso della Pipeline dovrà rilasciare ed eliminare oggetti dell'API di Kubernetes. Per questo motivo è neccessario connettere Spinnaker a Kubernetes. Per farlo è necessario:

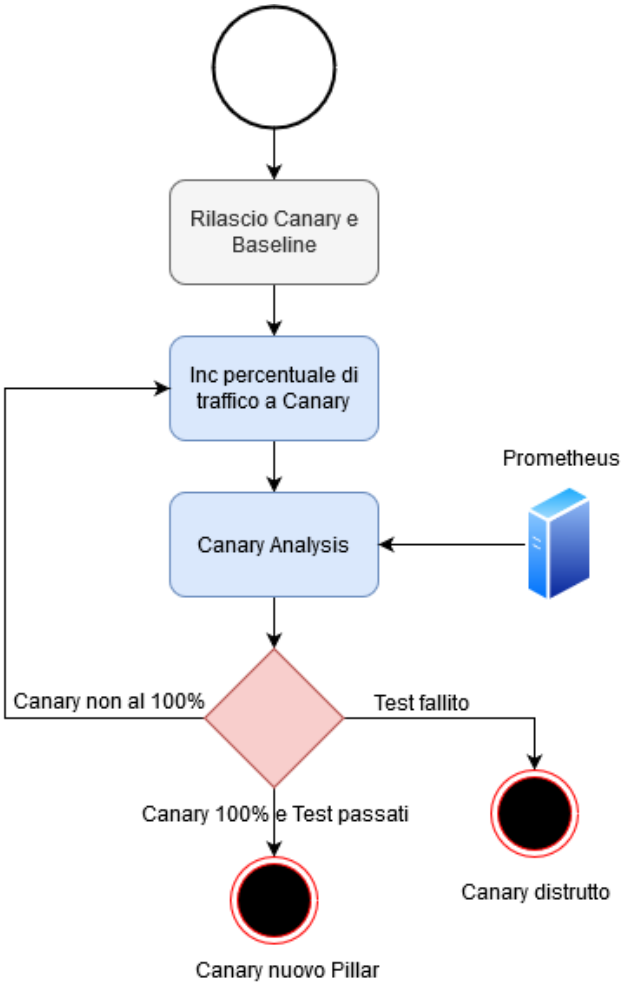


Figura 4.3: Schema della Pipeline della fase di produzione

1. Abilitare Kubernetes come provider su Spinnaker.

2. Creare un account Kubernetes in cui inserire il **context**.

Il context è contenuto nel file kubeconfig di Kubernetes. Un kubeconfig descrive il cluster, gli utenti ed i meccanismi di accesso. Il context è l'elemento del kubeconfig che raggruppa i parametri di accesso. Il context aggiunto all'account permette di decidere su quale cluster e con quale account Spinnaker andrà a rilasciare gli oggetti dell'API di Kubernetes.

A questo punto ogni volta che Spinnaker andrà in un suo stage della Pipeline ad interagire con Kubernetes recupererà le informazioni dal context nell'account Kubernetes indicato e svolgerà le sue azioni.

Connessione a Jenkins Come strumento di CI nella Pipeline di Spinnaker si utilizza Jenkins. La procedura per poter connettere Jenkins e Spinnaker si può dividere in due fasi, una in cui si lavora su Jenkins e l'altra in cui si configura la connessione su Spinnaker.

Su Jenkins la procedura che si è fatta è la seguente:

1. Abilitare l'**API** di Jenkins che di default non è attiva.

2. Abilitare l'accesso da remoto all'API e garantire agli utenti i **privilegi** per poterci interagire.

3. Creare un **account** su Jenkins con username e password con cui accedere all'API.

Su Spinnaker invece bisogna:

1. Abilitare Jenkins come strumento di **CI**.

2. Creare un account con l'**indirizzo IP e porta** su cui contattare l'API di Jenkins.

3. Aggiungere all'account le **credenziali d'accesso** dell'account creato in precedenza su Jenkins.

Per proteggere Spinaker e Jenkins dalla vulnerabilità di tipo CSRF si è condiviso tra loro un crumb, questo è il metodo consigliato sulla guida ufficiale di Spinnaker [14]. Bisogna abilitare la protezione da CSRF all'account creato su Spinnaker relativo all'accesso a Jenkins, è inoltre necessario installare il plugin Strict Crumb Issuer su Jenkins.

Una vulnerabilità di tipo CSRF (Cross-site request forgery) in sintesi è l'appropriazione da parte di un utente malevolo di una sessione preesistente autenticata da

un in precedenza utente legittimo. Una soluzione a questa vulnerabilità, utilizzata dal plugin installato su Jenkins, è lo scambio di un token all'apertura della sessione di comunicazione che identifica l'utente che ne è proprietario. Lo scopo è quello di utilizzarlo aggiungendolo ad ogni richiesta in modo tale da riconoscere che il mittente è lo stesso utente che ha creato la sessione. L'utente malintenzionato, in quanto non è a conoscenza del token scambiato inizialmente, non potrà appropriarsi della sessione.

4.2 Inizializzazione

In questa fase si va a preparare l'ambiente per l'esecuzione del Canary Analysis. Ma prima di ciò è necessario creare un meccanismo per far partire la Pipeline una volta che un'applicazione è pronta per passare dalla Fase di Test svolta con una Pipeline su Jenkins alla Fase di Produzione. Una volta in esecuzione, la Pipeline deve poter rilasciare la Baseline ed il Canary recuperando però le informazioni corrette riguardanti il numero di versione da rilasciare per entrambe e le risorse da assegnare al Canary.

4.2.1 Trigger e Recupero dati

Come anticipato nel capitolo relativo alla fase di Test la Pipeline di Spinnaker si deve avviare ogni volta che c'è un push sulla repository dell'applicazione in esame su Docker Hub.

Connessione a DockerHub Allo stesso modo di Jenkins e Kubernetes per potersi connetere a Docker Hub è necessario per prima cosa abilitarlo come provider su Spinnaker. A questo punto si va a creare una account in cui Spinnaker potrà recuperare le informazioni per potercisi connettere. Le informazioni sono:

- **Indirizzo IP e Porta**: l'indirizzo a cui Spinnaker si dovrà connettere per avere accesso al docker registry, nel caso di Docker Hub indirizzo IP e porta vanno sostituiti con *'index.docker.io'*.

- **Username e Password**: le credenziali di accesso con cui accedere al docker registry.

- **Repositories**: vanno elencate tutte le repository di interesse, tra queste sicuramente ci dovranno essere quelle di tutte le applicazioni che vengono sottoposte al processo di aggiornamento alla nuova versione con questa Pipeline.

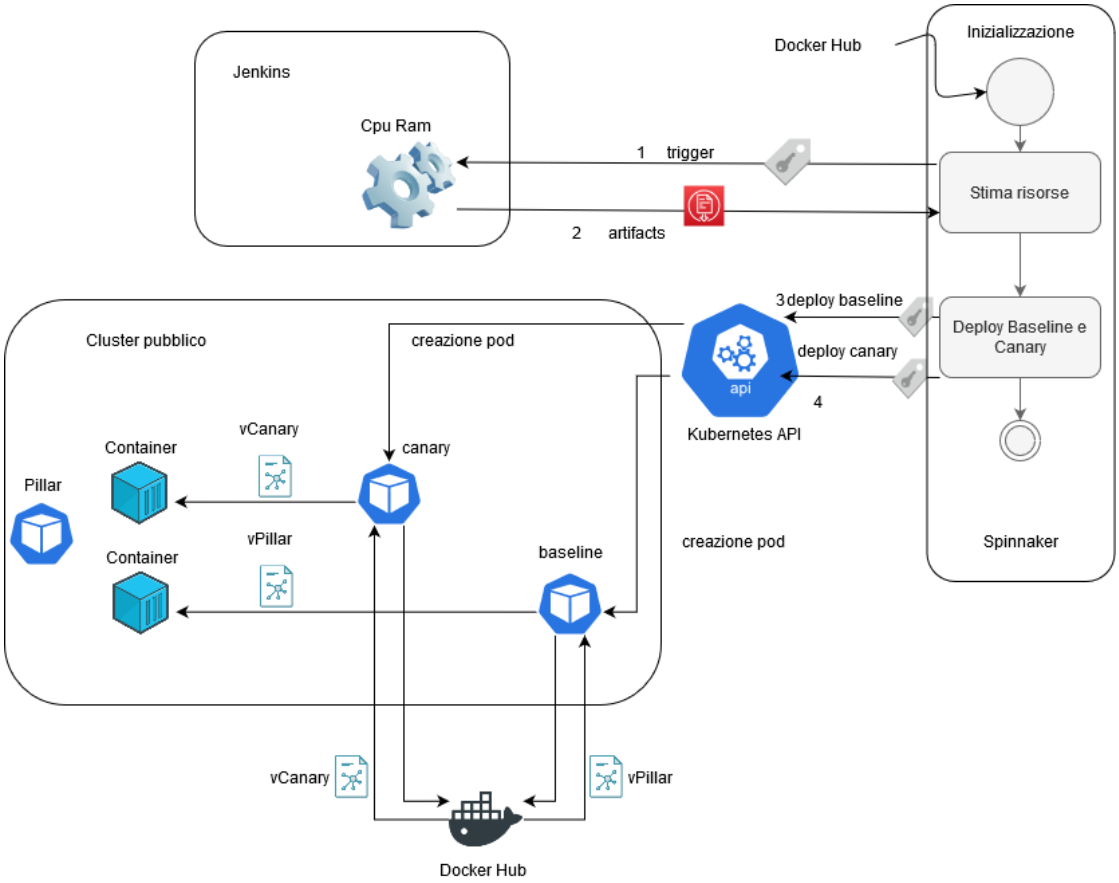


Figura 4.4: Schema dell'Inizializzazione

Impostazione Trigger Nel primo stage delle Pipeline di Spinnaker, il Configuration Stage, è possibile aggiungere dei trigger per far avviare la Pipeline. Per la tesi ho dato queste impostazioni:

- Ho selezionato come tipo di trigger, **'Docker Registry'**.
- Ho selezionato il docker registry, che in questo caso è quello dell'account creato in precedenza che indirizza verso **Docker Hub**.
- A questo punto vengono elencati tutti i **repository** disponibili su quell'account, tra questi va scelto quello che contiene le varie versioni dell'applicazione presa in esame.

Con queste impostazioni ogni nuovo push sul repository dell'applicazione farà avviare la Pipeline. Per questo motivo viene inserita l'istruzione di eseguire il push dell'immagine su Docker Hub alla fine della Fase di Test in caso di successo.

Recupero della versione del Pillar Manca un'informazione importante prima di poter rilasciare la Baseline, cioè quale versione dell'immagine dell'applicazione si deve recuperare? Questa informazione è già presente nel cluster in quanto è la stessa versione del Pillar. L'idea è quella di 'cercare' nel file yaml del Pillar la versione da utilizzare per l'immagine della Baseline. Per fare ciò si utilizza un certo tipo di stage, i *'Find Artifacts From Resource'*.

In questo tipo di stage si seleziona l'account Kubernetes corrispondente al cluster desiderato. A questo punto è possibile selezionare il namespace, il tipo di oggetto dell'API ed il nome assegnato, il risultato dello stage sarà un artifact contenente le sue informazioni. Il namespace sarà quello utilizzato per l'ambiente di Produzione per poi andare ad indicare di recuperare un oggetto di tipo 'Deploy' con il nome 'Pillar'.

Quando si andrà a rilasciare la Baseline sarà sufficiente indicare come immagine

```
${#stage("FindArtifacts").context["artifacts"][0]["reference"]}
```

Con 'FindArtifacts' il nome che ho dato allo stage in cui faccio l'operazione di 'Find Artifcats From Resource'.

Quello che viene fatto da questa istruzione è semplicemente andare a recuperare l'id dell'immagine dall'artifact.

Stima delle risorse necessarie per il Canary Per lo stesso ragionamento fatto nella Fase di Test è necessario stimare le risorse da assegnare alla nuova versione che sta per essere rilasciata.

Invece di creare un metodo nuovo o creare un Pipeline dedicata su Spinnaker si è scelto di andare ad utilizzare la stessa Pipeline di Jenkins che va ad eseguire

questo lavoro. Come visto nei paragrafi precedenti Spinnaker e Jenkins sono in grado di comunicare e possono iniziare delle Pipeline sia di uno che dell'altro.

Ciò è quindi possibile farlo inserendo uno stage di tipo 'Jenkins'. Nelle impostazioni di questo tipo di stage si va a configurare quale Jenkins master si deve utilizzare e quale Pipeline far partire. Il Jenkins master è possibile selezionarlo tra quelli per cui è stato creato un account come detto in precedenza. Una volta selezionato il Jenkins Master la lista delle Pipeline disponibili viene caricata ed è possibile selezionare quella desiderata da far partire.

L'ultimo problema è come far *comunicare* i dati recuperati dalla Pipeline di Jenkins a Spinnaker. L'unico modo messo a disposizione è la comunicazione attraverso i *Property File*. Questo tipo di file è utilizzato specialmente in Java principalmente per conservare la configurazione delle applicazioni. Consiste in una coppia chiave-valore, entrambi salvati come stringhe. Una volta stimate le risorse nella Pipeline di Jenkins viene creato il file 'data.properties' in cui vengono aggiunti come proprietà i parametri:

- **cpuReq**: CPU da assegnare subito all'applicazione, cioè la quantità di CPU minima per far partire l'applicazione nel cluster.
- **cpuLim**: CPU massima richiedibile dall'applicazione.
- **ramReq**: RAM da assegnare subito all'applicazione, cioè la quantità di RAM minima per far partire l'applicazione nel cluster.
- **ramLim**: RAM massima richiedibile dall'applicazione.

Una volta caricati questi valori nel file .properties, deve essere esposto dalla Pipeline come risultato. Ciò viene fatto nella parte finale della Pipeline in cui si indica:

```
archiveArtifacts artifacts: 'data.properties'
```

Dal lato di Spinnaker è invece disponibile nello stage di tipo 'Jenkins' una sezione in cui si indica il nome del Property File che è previsto ricevere dalla Pipeline di Jenkins. Se il Property File viene ricevuto e ne passa la verfica del formato lo stage viene considerato un successo. Se il Property File è mancante alla fine dell'esecuzione della Pipeline o il formato del file non è conforme allo standard lo stage fallisce.

4.2.2 Rilascio

Ora che sono state recuperate le informazioni fondamentali per poter avviare la Pipeline è possibile rilasciare tutti gli oggetti necessari al suo proseguo come le due versioen Baseline e Canary e gli oggetti di Istio per poter permettere il controllo del traffico dei dati.

Rilascio Baseline Il rilascio di oggetti su Kubernetes avviene attraverso uno stage di tipo 'Deploy (Manifest)'. Deve aver selezionato il cluster Kubernetes in cui rilasciare gli oggetti dell'API, è possibile selezionarli tra quelli per cui si è creato un account come spiegato nei paragrafi precedenti.

Dato che il tipo di manifest da rilasciare si conosce prima dell'esecuzione della Pipeline ho scelto di utilizzare il metodo di rilascio 'statico' offerto dallo stage. In questa modalità si va a scrivere il manifest direttamente nello stage seguendo la sintassi di un file yaml.

La Baseline viene rilasciata all'interno del namespace in cui si identifica l'ambiente di Produzione, la label associata per il Service è la stessa del Pillar. Questo particolare non influirà sul traffico in sè in quanto la sua distribuzione avverrà grazie agli oggetti dell'API di Istio.

Il label version della Baseline non è uguale al numero di versione effettivo dell'immagine. Ciò viene fatto per avere una continuità di utilizzo della Pipeline. Pillar, Baseline e Canary come label di versione hanno rispettivamente 'pillar', 'baseline' e 'canary', in questo modo i tag delle query e le impostazioni degli oggetti dell'API di Istio rimangono costanti nel tempo. Questa scelta si basa sul pensiero che per il sistema non è importante conoscere l'effettiva versione rilasciata, dato che si prende il caso nel cui non ci saranno mai canary deploy di più versioni contemporaneamente, l'unica cosa importante è poter distinguere i tre tipi di rilasci che avvengono in quel momento senza badare alla versione dell'immagine su Docker Hub. Ciò sarà importante da indicare solamente nel momento in cui Kubernetes dovrà recuperare l'immagine da Docker Hub.

L'immagine nel file yaml della Baseline viene indicata come descritto nel paragrafo del 'Recupero della versione Pillar' [4.2.1].

Rilascio Canary Lo stesso tipo di stage è stato utilizzato anche per il rilascio del Canary. Le differenze sostanziali sono il cambio del label version in cui si va a indicare 'canary' per il motivo detto nel paragrafo precedente, l'impostazione delle risorse che è possibile utilizzate recuperate con la Pipeline di Jenkins e come selezionare l'immagine da utilizzare.

Per quanto riguarda le risorse da allocare esse vengono recuperate dal File properties ritornato dalla Pipeline di Jenkins. La sintassi è la seguente:

```
${#stage("Jenkins-get-ram-cpu")["context"]["<nome-proprità"]}
```

Con 'Jenkins-get-ram-cpu' che è il nome dato allo stage in cui si avvia la Pipeline di Jenkins.

Invece la versione da utilizzare per il Canary è la stessa che ha causato il trigger della Pipeline. Questa informazione è possibile recuperarla utilizzando la sintassi:

```
${trigger["tag"]}
```

In questo modo per indicare l'immagine da utilizzare si scrive

```
<repository-dell'applicazione>:${trigger["tag"]}
```

Rilascio degli Oggetti di Istio La gestione del traffico verso le tre versioni è gestita utilizzando gli oggetti di Istio. Vengono rilasciati una Destination Rule, un Virtual Service ed un Gateway. Il rilascio di questi oggetti avviene con le stesse modalità del rilascio del Canary e della Baseline utilizzando uno stage di tipo 'Deploy (Manifest)'.

Destination Rule Nella Destination Rule si va a suddividere il traffico in subset, ciò viene fatto secondo il label 'version'. Quindi il traffico viene diviso per andare verso i Pod con i label 'canary', 'baseline' e 'pillar'.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: color
  namespace: color
spec:
  host: color.color.svc.cluster.local
  subsets:
    - labels:
        version: baseline
      name: baseline
    - labels:
        version: canary
      name: canary
    - labels:
        version: pillar
      name: pillar
```

Figura 4.5: Destination Rule per il Canary Deploy

Virtual Service Nel Virtual Service viene implementata la suddivisione del traffico, inviandolo in percentuale ad ognuna delle direzioni indicate nella Destination Rule. Per iniziare il canary deploy viene suddiviso il traffico inviandone:

- **80%** verso il Pillar.
- **10%** verso il Canary e la Baseline.

La scelta di iniziare partendo direttamente dal 10% verso il Canary è dovuta al fatto che per me non era possibile inviare una mole di traffico tale da poter partire da una percentuale minore e poter effettuare delle analisi significative. A seconda della quantità di traffico verso il servizio sarà necessario verificare la percentuale minima che si può adottare per iniziare il canary deploy.

Nel mio caso specifico il problema nasceva dalla metrica del Response time. Indicativamente per i miei settaggi Prometheus effettuava un *campionamento* ogni 15 secondi. Dalle mie osservazioni per poter ricevere un campione dalla metrica fornita da Istio è necessario che avvenga un certo numero minimo di transizioni in quei 15 secondi.

Tenendo conto che il numero di transazioni da me iniziate erano dell'ordine delle centinaia al secondo per tutti e tre i Deploy, per ricevere una lettura significativa e stabile nel tempo sul response time ho riscontrato che quel 10% era il traffico minimo. È da sottolineare l'aggettivo *stabile* perché dalle mie prove è venuto fuori che se le transizioni in quei 15 secondi non sono sufficienti si verifica la presenza di numerosi outlier che vanno a inficiare sull'analisi del Canary Analysis. Se il numero di transizioni è ancora inferiore la maggior parte dei campioni può ritornare nulla, rendendo l'analisi meno affidabile in quanto effettuata su un numero di campioni esiguo.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: color
  namespace: color
spec:
  gateways:
    - color-gateway
  hosts:
    - '*'
  http:
    - route:
        - destination:
            host: color.color.svc.cluster.local
            port:
              number: 8082
            subset: pillar
          weight: 80
        - destination:
            host: color.color.svc.cluster.local
            port:
              number: 8082
            subset: canary
          weight: 10
        - destination:
            host: color.color.svc.cluster.local
            port:
              number: 8082
            subset: baseline
          weight: 10
```

Figura 4.6: Virtual Service per il Canary Deploy

Gateway Per completare il tutto viene aggiunto un Gateway per il traffico in ingresso. Sarà poi il Virtual Service insieme alla Destination Rule a dividere il traffico. Il proxy server su cui viene applicato questo Gateway è il proxy di Istio, nel particolare di questa tesi il traffico deve essere inviato sulla porta 8087.

4.3 Canary Analysis

Ora che il Canary, Baseline e gli oggetti di Istio per inoltrare correttamente il traffico sono stati rilasciati si può iniziare il ciclo di canary deploy vero e proprio.

Flusso di lavoro Il ciclo principale del canary deploy è sintetizabile in questa serie di istruzioni:

1. Analisi delle prestazioni del Canary rispetto alla Baseline.
2. Aumento del traffico verso il Canary e la Baseline.
3. Fino a quando il Canary non raggiunge il 50% ritorno al punto 1.
4. Tutto il traffico viene indirizzato verso il Canary.

Nella tesi gli step di incremento del traffico verso il Canary e la Baseline sono:

1. 80% Pillar 10% Canary 10% Baseline, che è il punto di partenza con cui inizia il canary deploy.
2. 50% Pillar 25% Canary 25% Baseline.
3. 0% Pillar 50% Canary 50% Baseline.

In base al risultato di ogni Canary Analysis si sceglie se far continuare il canary deploy o meno. Non appena un singolo Canary Analysis dà un giudizio negativo si considera conclusa la sperimentazione della nuova versione con un esito negativo. Se invece tutta la sequenza di analisi hanno risultati positivi il canary deploy si conclude con successo.

Aumento del traffico verso il Canary La soluzione adottata per cambiare la distribuzione del traffico fra i tre Deploy è quella di modificare le percentuali scritte nel Virtual Service. Dopo ogni Canary Analysis c'è sempre uno stage di tipo 'Deploy (Manifest)' il quale va ad applicare la modifiche.

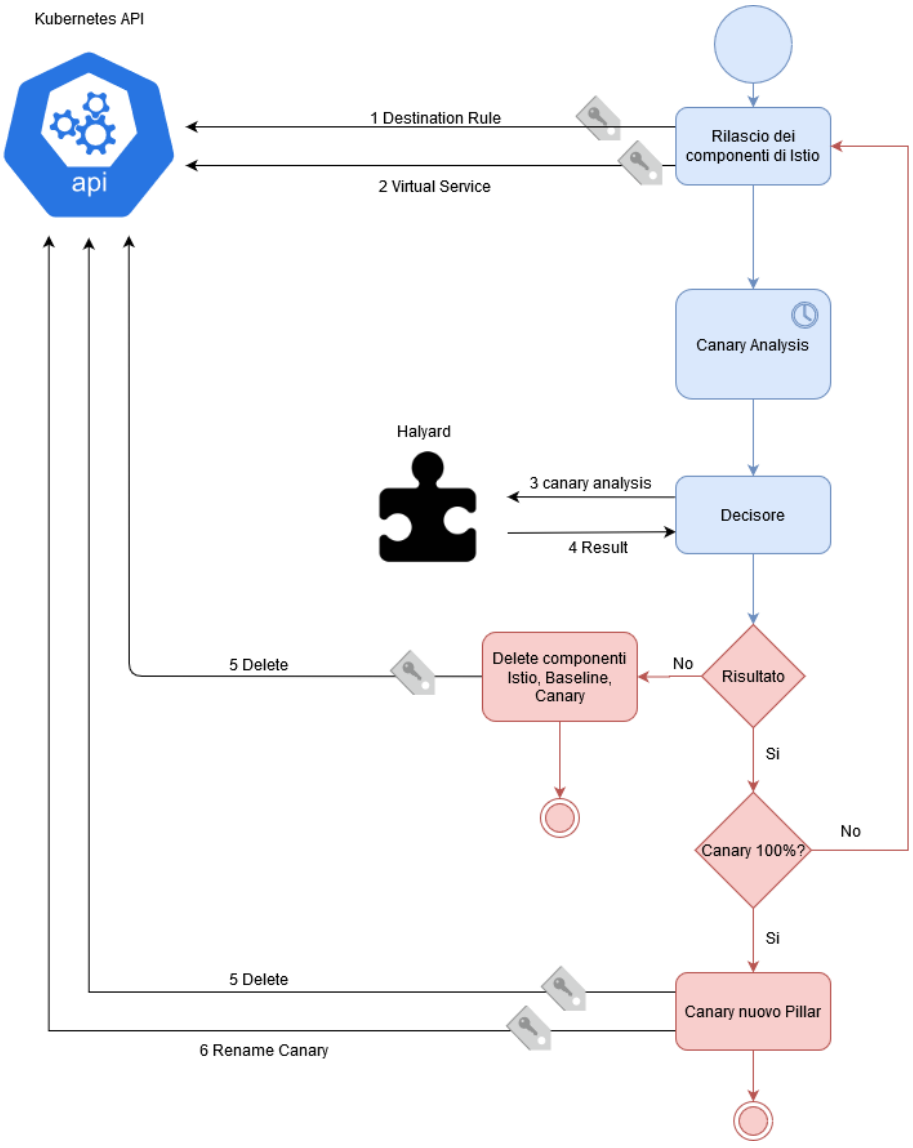


Figura 4.7: Schema della sezione della Pipeline relativo al 'Canary Analysis' la sua conclusione

Metriche selezionate per la tesi Le metriche selezionate per la tesi ed inserite nel Canary Config sono:

- **Response Time**: tempo di risposta ad una richiesta effettuata dall'utente. Un suo incremento ne determina un deterioramento. Rientra tra i parametri che rilevano l'efficacia delle performance di rete dell'applicazione.

- **N° 500 Error**: numero di error 500 che restituisce il server. Un incremento degli errori ne determina un peggioramento. È inserito nel gruppo delle metriche relative alle performance di rete.

- **CPU**: utilizzo della CPU da parte dell'applicazione. Un suo incremento indica delle performance peggiori. Rientra tra le metriche che indicano le performance dell'applicazione sull'impiego delle risorse hardware del cluster.

- **RAM**: memoria RAM occupata dall'applicazione. Un suo incremento ne determina un peggioramento. Fa parte del gruppo che valuta le performance dell'applicazione sul consumo delle risorse hardware del cluster.

I pesi da assegnare ai due gruppi dipendono dall'obbiettivo del team di sviluppo. Nel caso della tesi si è data più importanza alle performance dell'applicazione verso gli utenti, quindi le metriche delle performance di rete assegnando un 67% di peso al primo gruppo ed un 33% al secondo.

Lettura risultati del Canary Analysis La lettura dei risultati del Canary Analysis è importante per poter gestire il proseguo della Pipeline. In questo modo è possibile decidere se ritornare ad uno stato precedente o concludere con il Canary che diventa il nuovo Pillar.

Il metodo adottato per leggere i risultati del Canary Analysis e modificare l'esecuzione della Pipeline è stato quello di inserire una condizione sull'esecuzione di ogni stage. la sintassi è la seguente:

```
${#stage('CanaryAnalysis10%')['status'].toString()=='SUCCEDED'}
```

Con 'CanaryAnalysis10%' il nome dello stage in cui si esegue il Canary Analysis in cui il Canary riceve il 10% del traffico.

Spinnaker non permette di creare uno stage che permetta di dividere la Pipeline in due branch ed eseguirne solo uno dei due. Inoltre se la condizione di esecuzione di uno stage non è rispettata, esso non viene eseguito ma gli altri stage a valle lo saranno. Per questo motivo la condizione è da inserire in tutti gli stage succesivi al Canary Analysis.

4.4 Termine

Nella fase finale della Pipeline si conclude il ciclo del Canary Analysis e si porta il sistema in uno stato stabile, cioè pronto per una eventuale nuova esecuzione della Pipeline.

Canary Analysis fallisce Se il Canary Analysis fallisce si devono eliminare le tracce della nuova versione riportando il sistema allo stato precedente. Dopo che un'analisi dà risultato negativo vengono inseriti degli stage di tipo *'Delete (Manifest)'* e di tipo *'Deploy(Manifest)'*.

Gli stage 'Delete (Manifest)' sono utilizzati per connettersi al cluster Kubernetes ed eliminarne degli oggetti dell'API. In questi stage viene specificato il cluster su cui operare selezionando l'account Kubernetes con cui effettuare la connessione. Deve essere specificato il namespace in cui si trova l'oggetto dell'API ed il suo nome. La procedura è la seguente:

1. Il traffico viene portato al 100% verso il **Pillar**.
2. Viene eliminato il Deploy della Baseline.
3. Viene eliminato il Deploy del Canary.
4. Viene eliminato dal Virtual Service la parte in cui viene descritta la distribuzione del traffico.
5. Viene eliminata la Destination Rule che non risulta più necessaria in quanto non esistono più i subset verso i quali inviare il traffico.

La procedura descritta ha questa determinata sequenze perché è quella che garantisce che gli utenti finali abbiano il minor numero di problemi relativi al ritorno allo stato precedente. Questo perché si va prima a dirottare tutto il traffico verso la versione che rimarrà come definitiva dopo l'esecuzione della Pipeline, cioè il Pillar, che essendo già in esecuzione non avrà problemi a farsene carico. Una volta taglito l'accesso agli utenti verso la Baseline ed il Canary, i loro Deploy possono essere eliminati.

Canary Analysis ha successo Nel caso in cui tutti i Canary Analysis abbiano successo lo stato finale della Pipeline deve trasformare il Canary come nuovo Pillar eliminando gli oggetti dell'API inutili. Come nel caso in cui un Canary Analysis dia risultato negativo vengono utilizzati degli stage di tipo 'Deploy (Manifest)' e 'Delete (Manifest)'. La procedura è la seguente:

1. Il traffico viene portato al 100% verso il **Canary**.

2. Viene eliminato il Deploy della Baseline.
3. Viene eliminato il Deploy del Pillar.
4. Viene eliminato dal Virtual Service la parte in cui viene descritta la distribuzione del traffico.
5. Viene eliminata la Destination Rule che non risulta più necessaria in quanto non esistono più i subset verso i quali inviare il traffico.
6. Il Canary viene rietichettato come Pillar.

L'idea dietro questa sequenze di operazioni è la stessa di quella precedente. L'unica differenze sta nel rietichettare alla fine il Canary in quanto essa diventa la nuova versione 'ufficialmente' in produzione redendo l'ambiente semanticamente uguale allo stato precedente definito come 'stabile'. In questo modo l'ambiente è pronto per una nuova esecuzione della Pipeline.

Capitolo 5

Conclusioni

Ora che è stato spiegato come è stata affrontata la tesi è venuto il momento di trarre le conclusioni sul lavoro svolto. In questo capitolo si andrà a verificare i costi che un'implementazione di questo tipo comporta, gli obbiettivi raggiunti ed i possibili miglioramenti che si possono fare all'intero processo.

5.1 Analisi dei consumi

Le strutture aggiunte al cluster per permettere un'operazione di questo tipo non sono da trascurare. Ognuna ha un consumo in termini di risorse hardware che dovranno essere comparate ai benifici che comportano.

5.1.1 Hardware disponibile

Prima di parlare dei costi ecco una breve panoramica sull'hardware su cui è stata sviluppata la tesi.

Architettura Il progetto della tesi è stato eseguito in un cluster Kubernetes formato da due nodi, un Master ed un Worker. Entrambi i nodi hanno la stessa configurazione hardware.

Le risorse però sono sovradimensionate rispetto a quelle utilizzate per il progetto. Questo è dovuto al fatto che il cluster Kubernetes era utilizzato contemporaneamente per lo sviluppo di un secondo progetto ugualmente dispendioso. Le risorse effettivamente disponibili erano quindi la metà di quelle elencate.

CPU Il processore montato per entrambi i nodi era un Intel(R) Xeon(R) Gold 5115 CPU @ 2.40GHz con 4 core. Dato che l'unità di misura fondamente per la

gestione della CPU su Kubernetes è il mcpu e 1000 mcpu rappresentano un core, si avevano a disposizione 4000mcpu per nodo.

RAM La memoria RAM montata per ogni nodo era di 16GB. Kubernetes permette di andare a dividere la RAM per Deploy con una definizione del byte quindi non ci sono stati problemi nella loro assegnazione.

5.1.2 Strutture Accessorie

In questa parte si andranno ad analizzare i consumi dei componenti utilizzati come supporto oltre a verificare i consumi di Jenkins e delle varie tecnologie associate all'esecuzione della sua Pipeline.

Jenkins Jenkins è stato inserito nel cluster Kubernetes come un container. Si è cercato di creare un'immagine che permetta di essere autosufficiente per l'esecuzione della Pipeliene della Fase di Test. Oltre a Jenkins, quindi, sono stati inseriti:

- **Docker**: per poter creare l'immagine dell'applicazione nello stage 'Build' e poter interagire con i vari Docker Registry.
- **Selenium** e un browser web (nel caso specifico **Firefox**: per permettere l'esecuzione del relativo test.
- **JMeter**: nei consumi è da inglobare anche l'esecuzione di un'applicativo Java come JMeter.

InfluxDB Si è utilizzato InfluxDB in modo tale da poter conservare le metriche prodotte da Prometheus. In questo modo è possibile consultare le metriche in una finestra temporale passata per confrontarle con quelle della versione in esame nella Fase di Test.

È stato utilizzato anche per il dataset necessario per l'addestramento dell'algoritmo di Machine Learning di classificazione.

Minio Per poter installare Spinnaker si è dovuto aggiungere un sistema di storage per salvare le impostazioni delle applicazioni e le Pipeline create. La scelta è ricaduta su Minio che ha ovviamente un costo di risorse all'interno del cluster Kubernetes. Spinnaker offre la possibilità di utilizzare soluzione in cloud per questo compito come ad esempio Google Cloud Storage (GCS).

Consumi

nome	CPU (mcpu)	RAM (Mb)
Jenkins (durante Sel Test)	150	1258
InfluxDB	20	81
Minio	10	51
Totale	180	1390

5.1.3 Istio

L'utilizzo di Istio in questa tesi è stato strettamente legato al canary deploy per il suo modo di gestire il traffico rispetto alla versione base di Kubernetes. Ciò però comporta dei costi, sarà necessario valutare se questi costi sono possibili da affrontare o se si ritenga sia necessario per lo sviluppo.

Benefici L'introduzione di Istio permette di poter gestire la divisione del traffico secondo i label version. È inoltre possibile assegnare una percentuale esatta di traffico da inviare ad ognuno.

Questo permette di fare un canary deploy più 'preciso' per quanto riguarda il traffico da destinare al Canary, senza tuttavia andare a modificare il Deploy della versione precedente e adottare una qualche procedura particolare per quella nuova. Tutto questo è possibile osservarlo nella descrizione della Fase di Produzione nel capitolo precedente.

Soluzione Alternativa La soluzione alternativa ad Istio è quella di utilizzare gli strumenti base di Kubernetes 'giocando' sul numero di Pod attivi nel cluster per dividere il traffico tra loro. Questa soluzione si basa sul come è gestito il traffico in entrata su Kubernetes. Il traffico inviato verso un Service viene inoltrato a tutti i Pod contenenti il label selezionato al suo interno. Impostando un metodo di Load Balancer in Round Robin il traffico verrà distribuito equamente verso tutti i Pod selezionati.

Questo può essere sfruttato a proprio vantaggio per creare un canary deploy. Rilasciando Canary, Baseline e Pillar con la stessa etichetta il traffico viene diviso equamente tra i tre. Supponendo che venga creato un solo Pod per ognuno il traffico verrà diviso in questo modo: 1/3 verso il Canary, 1/3 verso la Baseline ed 1/3 verso il Pillar. Modificando il numero dei Pod è quindi possibile creare una divisione del traffico tra i tre. Ad esempio pensando allo step iniziale proposto nella tesi per poter dividere il traffico al 10% per il Canary, 10% per la Baseline e 80% per il Pillar basta creare 1 Pod per il Canary, 1 Pod per la Baseline e 8 Pod per il Pillar.

Questa metologia porta ovviamente a due principali svantaggi:

1. Si va a dover modificare il Deploy delle tre versioni, quindi si va a creare un consumo di risorse magari non necessario in quanto si devono creare delle repliche 'inutili'. È inoltre possibile che la creazione di un certo numero minimo di Pod sia necessario per poter gestire le risorse al meglio, ciò comporta una creazione ancora maggiore dei Pod delle altre versioni per arrivare alla percentuale desiderata.
2. Le percentuali devono tener conto di questa metodologia, utilizzare percentuali non perfettamente divisibili diventa proibitivo in quanto si dovrebbero creare troppe repliche per attuarle.

Consumi Di seguito una tabella con i consumi relativi ai vari componenti di Istio registrati nel mio ambiente di test.

nome	CPU (mcpu)	RAM (Mb)
Istio Citadel	20	32
Istio Galley	200	128
Istio Ingress Gateway	100	128
Istio Pilot	500	2176
Istio Policy	110	128
Istio Sidecar Injector	10	32
Istio Telemetry	800	1134
Istio Tracing	10	32
Kiali	10	40
Prometheus	10	64
Grafana	20	32
Jaeger	10	32
Totale	1790	3958

Conclusioni personali In definitiva ritengo la soluzione alternativa utile per canary deploy più semplici con percentuali studiate per non creare troppe repliche. Una realtà più avanzata dovrebbe poter sostenere il costo di utilizzo di Istio slegando la quantità di repliche dalla gestione del traffico in entrata ed in questo modo rendendo più efficiente e semplice il canary deploy. Sommato al fatto che Istio offre anche altre funzionalità che possono essere utilizzate per altri progetti nel cluster mitigando quindi i costi legati al singolo progetto.

5.1.4 Spinnaker

L'utilizzo di Spinnaker nella tesi è legato alle sue capacità di rilascio in contrasto a soluzioni che utilizzano solamente Jenkins sia per la Fase di Test che per la Fase di Produzione.

Benefici È da precisare per prima cosa che Spinnaker non ha nè l'intenzione nè le capacità di sostituire Jenkins come strumento per l'integrazione continua. Spinnaker offre supporto solamente come strumento per il continuous delivery su vari cloud-provider (in questo caso, Kubernetes). Spinnaker avrà sempre bisogno di strumenti come Jenkins per gestire una fase di continuous integration.

I benefici apportati da Spinnaker riguardano princiapalmente il possedere nativamente il supporto a vari cloud-provider, una dashboard leggibile e potente, e la sua capacità di gestire in modo facilitato i rilasci di un'applicativo.

Soluzioni alternative Molti team non utilizzano Spinnaker per il CD ma utilizzano Jenkins. Jenkins, come visto nei precedenti capitoli, permette di creare delle Pipeline ed in questo modo è possibile interagire con Kubernetes per fare CD e nello specifico un canary deploy. Tutti i passaggi visti nel capitolo 'Fase di Produzione' possono essere implementati oon una Pipeline di Jenkins.

Per le parti di rilascio si può comunicare con l'API di Kubernetes e fare applicare i Manifest per creare gli oggetti desiderati. Per quanto riguarda la parte del Canary Analysis è possibile ricreare il Decisore nel codice della Pipeline o creare un software esterno che ne esegua i compiti con cui la Pipeline può interagire, un po' come visto per l'algoritmo di ML di classificazione.

Consumi Di seguito una tabella con i consumi dei vari componenti di Spinnaker. Ognuno responsabile di varie funzionalità come scritto nel capitolo 'Tecnologie Utilizzate'.

nome	CPU (mcpu)	RAM (Mb)
Spin Clouddriver	20	1650
Spin Deck	10	36
Spin Echo	10	446
Spin Front50	32	534
Spin Gate	13	460
Spin Kayenta	25	423
Spin Orca	98	534
Spin Redis	61	93
Spin Rosco	10	335
Totale	279	4511

Dipendentemente se la Pipeline era in esecuzione o meno le misurazioni cambiano. Nei momenti di esecuzione in cui la Pipeline di Spinnaker andava ad eseguire gli stage in cui doveva eseguire il Canary Analysis arrivava a consumare anche 1000 mcpu.

Conclusioni Personali A differenza della parte di Istio non ritengo che Spinnaker sia un elemento centrale per effettuare un canary deploy. I servizi offerti sono ricreabili creando delle Pipeline su Jenkins. Ma secondo me sono indubbi i vantaggi nell'utilizzo di Spinnaker che con la sua interfaccia utente ed il suo modo di interfacciarsi con Kubernetes semplifica di molto l'intero processo per il team DevOps.

Non è nemmeno da trascurare come Spinnaker renda estramemente facile personalizzare i vari parametri della Pipeline attraverso l'interfaccia dei vari stage. Ciò rende utilizzabile la Pipeline per la messa in produzione anche dal personale che non è stato formato sullo sviluppo di software come richiedono invece le Pipeline di Jenkins.

5.2 Obbiettivi raggiunti

In questa sezione si andranno a vedere gli obbiettivi raggiunti rispetto a quelli prefissati all'inizio dello svolgimento della tesi.

5.2.1 Fase di Test

Nella Fase di Test si è riuscito ad automatizzare il processo di build, deploy e test dell'applicazione. In generale però non si è trovata una formulazione univoca di questa fase. Sarà sempre necessario una fase preparatoria che andrà a stabilirne alcuni passaggi in base all'applicazione presa in esame:

- Le **metriche** da selezionare per poter valutare le performance dell'applicazione non sono uguali per tutte. È necessario uno studio in cui si vanno a selezionare quelle più appropriate per le funzioni svolte dall'applicazione.

- Le **query** da utilizzare per recuperare le metriche non sono sempre uguali anche se le metriche recuperate esprimono lo stesso tipo di dato. Questo è dovuto per il modo in cui esse vengono scritte in Prometheus, i tag utilizzati cambiano in base all'ambiente in cui vengono rilasciate le applicazioni ed i nomi a loro associati. Per ogni applicazione è necessario riscrivere le query per renderle conformi al sistema.

- Se si vuole utilizzare la modalità con analisi matematica delle metriche è necessario uno studio per valutare i **limiti** oltre i quali si stabilisce che l'applicazione ha delle performance negative. Questi limiti devono essere calcolati per ogni applicazione in quanto il profilo delle performance ritenuto accettabile può cambiare in base alle funzionalità svolte.

- Per effettuare il processo di build dell'applicazione è necessario modificare la Pipeline a seconda dello strumento necessario, nella tesi lo strumento ideale è risultato essere Maven in quanto le applicazioni prese in esame erano WebApplication create utilizzando Spring, ma in generale può non essere vero.

Stima delle risorse Uno degli obbiettivi principali era quello di stimare le risorse che necessita l'applicazione per la sua esecuzione. Una soluzione a questo problema è stata fornita nel corso della tesi con *risultati accettabili*. Le applicazioni di test rilasciate con questo sistema non hanno avuto problemi durante la loro esecuzione, le risorse sono sempre state adeguate per le funzioni svolte senza riscontrare performance negative.

L'unico problema riscontrato è una quantità di risorse assegnate che in alcuni casi può risultare *eccessiva* rispetto a quelle realmente necessarie. In questi casi il problema risulta essere il limite minimo di risorse assegnate in quanto Kubernetes, tenendole in considerazione come risorse minime per l'esecuzione dell'applicazione, potrebbe andare a bloccare il rilascio di altre applicazioni nel cluster in quanto dai suoi conteggi non risulteranno esserci risorse sufficienti anche se presenti realmente.

Il lato positivo che ho riscontrato è che questo tipo di problematica **non** ha un effetto a *'cascata'*, le successive stime non vanno a tener conto delle risorse impegnate da Kubernetes ma solamente di quelle effettivamente utilizzate dall'applicazione. Quindi non si va a creare il problema in cui dopo vari cicli con questo errore il cluster risulta saturo con una sola applicazione che in realtà utilizza una minima parte delle risorse assegnate. Il problema rimane circoscritto al solo ciclo in cui avviene l'errore senza propagarsi.

Classificazione Per quanto riguarda classificare le performance dell'applicazione nella Fase di Test sono state proposte due soluzioni che in caso possono funzionare anche in modo complementare.

L'analisi matematica delle metriche, in input riceve tutte le metriche selezionate e con il calcolo di una funzione in cui si stima la qualità delle performance l'applicazione viene classificata come buona per la fase successiva o meno. Lo svantaggio di questa soluzione è riassumibile principalmente in due punti:

1. È dipendentente dalle capacità della funzione utilizzata, che in questo caso va a determinare attraverso una media pesata la distanza tra le performance dell'applicazione sottoposta ai test e quella corrente.

2. Come anticipato precedentemente è necessario uno studio dei limiti da impostare su ogni metrica per avere una classificazione più vicina possibile a quella desiderata.

L'analisi con **l'algoritmo di classificazione di machine learning** riesce a compiere il proprio dovere di classificare l'applicazione ma comporta un principale svantaggio che risiede nella creazione del *dataset*. È necessario, o avere già disponibile un dataset con tutti campioni classificati in precedenza oppure è necessario aver eseguito per un certo numero di volte la Pipeline con la modalità di analisi matematica.

5.2.2 Fase di Produzione

Per la Fase di Produzione l'obbiettivo era quello di trovare una strategia di rilascio che permettesse di diminuire l'impatto che un cambio di versione repentino può comportare. Utilizzando il Canary Deploy e le funzionalità di Spinnaker si è riusciti a raggiungerlo. Anche in questo caso però si ripropone il problema di aver creato una Pipeline che necessita di una fase precedente di studio in quanto anche qui sarà necessario studiare le metriche necessarie per una corretta valutazione dell'applicazione.

Canary Deploy La strategia adottata è stata quella del Canary Deploy. Anche se non eliminando completamente la possibilità che una parte degli utenti possa trovarsi ad avere un'esperienza negativa per un errore nella nuova versione dell'applicazione sicuramente ne limita di molto la possibilità rispetto al rilascio in one-shot. Il suo problema principale sta nella totale dipendenza dalla capacità di testing. Se il test utilizzato è poco selettivo un'applicazione potrebbe riuscire ad arrivare ad avere una percentuale estesa di pubblico anche se non ottimale dal punto di vista delle performance. Se invece il test è troppo selettivo un'applicazione potrebbe essere continuamente bocciata anche a fronte di performance accetabili.

Canary Analysis Questa funzione di testing nella tesi è stata affidata al Canary Analysis di Spinnaker affidantosi al Decisore di default. Dai test effettuati è risultato efficiente nell'identificare delle performance non ottimali rispetto a quelle della Baseline. Un problema individuato è che è particolarmente sensibile alla presenza degli outlier. Infatti si è reso necessario aggiungere i 5 minuti di warmup iniziale per diminuirne il numero. Inizialmente ciò non era stato fatto e per la prima analisi, in cui il Canary era appena stato rilasciato, la probabilità di un fallimento del test era molto alta.

5.3 Possibili miglioramenti

In questa sezione si andranno a proporre i miglioramenti possibili alle Pipeline ed alle tecnologie utilizzate che però non sono state implementate per mancanza di

tempo a disposizione.

5.3.1 Fase di Test

Nella Fase di Test i miglioramenti principali che sono venuti in mente riguardano principalmente due fasi, quella in cui si vanno a stimare le risorse necessarie all'applicazione per la sua esecuzione e quella in cui si va a classificare l'applicazione per decidere se farla passare alla Fase di Produzione o meno.

Stima delle risorse Per quanto riguarda la stima delle risorse necessarie all'esecuzione dell'applicazione si è pensato alla possibile creazione di un algoritmo di machine learning di classificazione. In input all'algoritmo dovrebbe arrivare il codice dell'applicazione o eventualmente una serie di parametri in cui si vanno a delinearne le funzionalità e le strutture dati necessarie.

Classificazione Per la parte di classificazione dell'applicazione esistono due miglioramenti possibili.

1. Si potrebbe creare una funzione matematica più complessa che tiene conto della presenza degli outlier o in generale della varianza dei campioni.

2. Questo punto potrebbe essere il più interessante, il passaggio dalla modalità di analisi matematica a quello dell'utilizzo del classificatore è gestito manualmente dal team DevOps che opera con la Pipeline. Potrebbe essere interessante creare un meccanismo automatico dove in base alla popolazione del dataset ed alla sua struttura effettui in automatico il cambio di modalità.

5.3.2 Fase di Produzione

Per la Fase di Produzione i possibili miglioramenti riguardano lo studio di strategie di deploy alternativi e l'implementazione di un Decisore custom in cui si sviluppa un test diverso per il Canary Analysis.

Metodi di deploy alternativi Il canary deploy non è l'unica strategia di deploy che si può adottare in questi contesti. Si potrebbe studiare una strategia alternativa nuova in cui gli utenti non vengono obbligati a passare alla nuova versione dal sistema in modo arbitrario.

Decisore custom Potrebbe risultare interessante creare un Decisore custom in cui si sviluppano nuove metodologie per l'analisi del canary deploy. Si potrebbero sfruttare o, altri strumenti matematici, oppure si potrebbe andare ad implementare

un algoritmo di machine learning di classificazione come fatto per la Fase di Test. I problemi da affrontare sarebbero sempre quelli riguardanti la popolazione del dataset e la gestione delle nuove funzionalità introdotte con le nuove versioni che possono portare ad un cambio delle performance.

Bibliografia

[1] Amazon Web Services. *Cos'è DevOps?* 2020. URL: https://aws.amazon.com/it/devops/what-is-devops/.

[2] Docker Team. *Docker Documentation*. 2020. URL: https://docs.docker.com/get-started/overview/.

[3] Kubernetes Team. *Kubernetes Documentation*. 2020. URL: https://kubernetes.io/it/docs/concepts/.

[4] Jenkins Team. *Jenkins Documentation*. 2020. URL: https://www.jenkins.io/doc/.

[5] Istio Team. *Istio Documentation*. 2020. URL: https://istio.io/latest/docs/concepts/.

[6] Prometheus Team. *Prometheus Documentation*. 2020. URL: https://prometheus.io/docs/introduction/overview/.

[7] InfluxDB Team. *InfluxDB Documentation*. 2020. URL: https://docs.influxdata.com/influxdb/v1.8/.

[8] Jaeger Team. *Jaeger Documentation*. 2020. URL: https://www.jaegertracing.io/docs/1.16/.

[9] Kiali Team. *Kiali Documentation*. 2020. URL: https://kiali.io/documentation/v1.9/features/.

[10] Spinnaker Team. *Spinnaker Documentation*. 2020. URL: https://www.spinnaker.io/concepts/.

[11] Spinnaker. *Spinnaker Architecture Documentation*. 2020. URL: https://www.spinnaker.io/reference/architecture/.

[12] XGBoost Documentation Authors. *XGBoost Documentation*. 2020. URL: https://xgboost.readthedocs.io/en/latest/.

[13] Amazon Web Services. *Canary deployment*. 2020. URL: https://wa.aws.amazon.com/wat.concept.canary-deployment.en.html.

[14] Spinnaker Team. *Configure Jenkins and Spinnaker for CSRF protection*. 2020. URL: https://www.spinnaker.io/setup/ci/jenkins/.