Alma Mater Studiorum · Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea Magistrale in Ingegneria Informatica

CASO D'USO DI ARCHITETTURA BIG DATA: DALLO SVILUPPO AL DEPLOYMENT

Tesi di Laurea in Data Mining

Relatore:
Chiar.mo Prof.
Claudio Sartori

Candidato:
Dario Pasquali

Anno Accademico 2016/2017
Sessione III

Abstract

Negli ultimi anni, con la diffusione su larga scala di dispositivi e piattaforme multimediali, è aumentata notevolmente la richiesta sul mercato di servizi sempre innovativi ed aggiornati. Con lo sviluppo di piattaforme cloud, si è aperta la strada all'accumulo, elaborazione e analisi di grandi quantità di dati, utili a offrire servizi e a supportare le decisioni, i cosiddetti Big Data. L'unione di questi due fattori ha portato all'abbandono dei metodi di sviluppo tradizionali a fronte di tecniche adatte al soddisfare la richiesta del mercato. Il movimento DevOps presenta una rivoluzione culturale in grado di supportare il Deployment continuo di nuove versioni tramite una pipeline in grado di garantire qualità e valore del prodotto software.

Lo scopo di questo lavoro è stato sviluppare un caso d'uso di architettura Big Data, applicandone al processo di sviluppo le metodologie sponsorizzate dal movimento DevOps. Nello specifico si è sviluppato un raccomandatore di film, basato sul dataset Movielens M20, strutturando il flusso di elaborazione in quattro processi tali da mostrare i vari tipi di elaborazione del Big Data. Le raccomandazioni sono generate con la tecnica del Collaborative Filtering, fisicamente implementato come Matrix Factorization e ottimizzato tramite l'algoritmo di Alternating Least Square. Si sono implementati due modelli di raccomandazione, uno Soft in grado di raccomandare con MSE di 0.7 e uno binario con accuratezza del 85%.

Il DevOps è stato applicato ai processi e all'architettura di rete, strutturando

quattro pipeline di Continuous Deployment in grado di rilasciare in produzione aggiornamenti in modo totalmente automatico, garantendo il funzionamento e la qualità del prodotto sviluppato. Si è inoltre dimostrato, tramite una stima, il vantaggio economico e temporale del DevOps all'aumentare della complessità del sistema.

Indice

Capitolo 1

Introduzione

Negli ultimi anni il mondo ha subito numerosi e repentini cambiamenti, la diffusione su larga scala di Internet e di dispositivi mobile all'avanguardia ha aperto le porte a numerosi servizi web accessibili a un numero inimmaginabile di utenti. Questo bacino non è solamente composto da un gran numero di individui, richiede anche continuo supporto, continui aggiornamenti, continuo rilascio di features e innovazioni, ad alta velocità e frequenza.

Di conseguenza, anche la gestione di deployment e release è stata stravolta, in passato era uso comune effettuare pochi rilasci di grandi dimensioni, con annessi problemi di integrazione, arresto dei servizi, malcontento degli utenti, uniti a un oneroso carico di lavoro per il personale aziendale. A meno di implementare specifiche tecniche di organizzazione non è quindi possibile raggiungere gli standard attuali del mercato.

Il movimento DevOps, nato nel 2008, appena 10 anni fa, propone una visione innovativa del processo di sviluppo del software, finalizzato alla creazione di una pipeline di Continuous Deployment in grado di gestire rilasci giornalieri ad alta affidabilità e qualità. Il movimento focalizza la sua attenzione principalmente sulle

persone, dirette responsabili del prodotto in ogni sua fase di vita, mirando alla creazione di un team multifunzionale che possa seguire il prodotto dalla progettazione al rilascio.

Il caso d'uso implementato, un raccomandatore di film basato sul metodo del Collaborative Filtering e l'algoritmo Alternating Least Square (ALS), è stato strutturato in modo da rappresentare un esempio generale di architettura e processi tipici di questa branca informatica. Il dataset Movielens M20, contenente 20 milioni di ratings, costituisce una buona base di partenza per illustrare i vantaggi, in termini di gestione dati, apportati dai tools e metodi utilizzati. Le sette pratiche proposte dal movimento DevOps sono state applicate allo sviluppo dei processi e dell'architettura del sistema implementato, abbattendo drasticamente i tempi di rilascio e permettendo di sviluppare un prodotto software di maggior valore.

Questa tesi è stata sviluppata durante un periodo di tirocinio presso l'azienda Data Reply S.p.A., con sede a Torino. L'esperienza, durata circa 5 mesi, è stata fondamentale per analizzare in prima persona il ciclo di vita di un prodotto software in ambiente reale, permettendomi di comprendere al meglio le problematiche e lo possibile soluzioni adottabili.

1.1 Struttura della tesi

Il Capitolo 2 e 3 vogliono dare una visione di background del mondo Big Data e del movimento DevOps, correlati nell'elaborato. Il primo analizza le problematiche di gestione portate da una tal quantità di dati, oltre ad illustrare gli strumenti utilizzati nel progetto. Il secondo invece mette in luce le motivazioni culturali che hanno portato al movimento DevOps, analizzando nel dettaglio le pratiche proposte.

Segue l'analisi del caso d'uso, introdotta nel Capitolo 4. Il capitolo successivo illustra l'architettura implementata, elencando i ruoli e la loro distribuzione all'interno

del cluster. Il Capitolo 6 analizza la pipeline di elaborazione del raccomandatore, soffermandosi sui quattro tipi di processi implementati.

Infine il Capitolo 7 mostra l'ottimizzazione del processo di sviluppo, realizzata applicando i principi DevOps.

Capitolo 2

Stato dell'arte

2.1 Big Data concetti generali

"Con il termine Big Data si intende una collezione di dataset talmente grandi e complessi da essere difficilmente processabili utilizzando database e applicativi tradizionali." [6]

Gartner identifica, con il termine Big Data, un insieme estremamente eterogeneo di dati, strutturati e non, prevenienti da molteplici fonti diverse. Essi, una volta organizzati in maniera opportuna, forniscono una vasta base di informazione che apre le porte a molteplici servizi e prodotti.

2.1.1 V model

Volendo dare una definizione più concreta ed applicabile di Big Data, è utile basarsi sul modello "*The 6 V*" presentato da Gartner. Un insieme di dati, per essere definito Big Data, deve possedere 6 semplici caratteristiche:

Volume

Il Dataset deve contenere una grande quantità di dati, si parla di TeraByte fino a PetaByte o addirittura ExaByte per le applicazioni più onerose. Questa caratteristica mette in luce innumerevoli problemi:

- **Memorizzazione**: Grandi quantità di dati richiedono metodi di memorizzazione capienti, efficienti, sicuri, distribuiti ed efficienti. I dati devono poter essere letti e scritti in maniera rapida e a bassa latenza;
- **Elaborazione**: Non è più possibile utilizzare i tradizionali metodi e pattern di elaborazione, servono nuovi modelli e linguaggi in grado di gestire e processare grandi quantità di dati efficientemente;
- **Trasmissione**: Anche l'infrastruttura di rete deve evolversi per sopportare il carico di dati da trasmettere, garantendo bassa latenza e alto throughput.

La necessità di un'architettura altamente scalabile e a basso costo risulta quindi evidente.

Velocity

I dati non solo sono in grandi quantità, sono anche generati ad alta velocità. Sempre più usuali sono le applicazioni che richiedono acquisizione, elaborazione e restituzione di informazioni in tempo reale (si parla di *"nearly real time"* o addirittura *"real time"*). Anche il minimo ritardo si commuta in ua perdita economica, di conseguenza ogni singolo componente del sistema deve garantire la velocità richiesta.

Variety

I Big Data sono per natura fortemente eterogenei. All'interno dello stesso dataset sono solitamente contenuti dati provenienti da fonti e rappresentanti informazioni radicalmente differenti.

Variability

L'eterogeneità vale anche all'interno della stessa "classe" di dati, essi possono variare ampiamente, è quindi necessario adottare una struttura in grado di far fronte ai possibili cambiamenti dei dati, considerando tutte le possibilità.

Veracity

Lo scopo finale dei Big Data è fornire un valore economico ed informativo che scaturisca in un vantaggio in termini di servizi e decisioni; esso dipende fortemente dalla precisione e accuratezza dei dati, risulta quindi necessario eliminare tutti gli elementi che non apportano un effettivo valore all'insieme. Questo processo di pulizia si esegue però solo al momento del caricamento nella base dati centralizzata, a monte i dati vengono estratti in maniera vorace; ammettendo dati sporchi e incompleti si mette al primo posto la quantità piuttosto che la qualità. Un opportuno processo di Estraction Transformation and Loading (ETL) si occuperà di pulire i dati mettendone in luce l'effettivo valore.

Value

Avendo chiaro in mente l'obiettivo finale dei Big Data, è necessario che essi apportino un un vantaggio a livello di business all'azienda che decide di affrontare un abstraction gap tale da soddisfare le 6 V. Per mostrare il loro effettivo valore, è necessario un sistema di **Visualization** che metta in risalto i vantaggi strategici estratti dai dati. Con i Big Data e, in particolare, con la Business Intelligence, il processo di estrazione delle decisioni è automatizzato e mostrato in maniera opportuna.

2.1.2 Architettura logica

Analizzando attentamente il modello, appare chiaro come alla base del successo di questa tecnologia sia necessaria un'architettura, logica e fisica, capace di memorizzare, elaborare e trasferire una così grande ed entropica quantità di dati.

Data Warehouse

L'elemento chiave dell'architettura logica per Big Data è il **Data Warehouse** (DWH), astrazione utilizzata per indicare un repository di dati appositamente pensato ed ottimizzato per gestire i Big Data e dare supporto attivo all'estrazione di informazioni da essi.

Il DWH conterrà dati provenienti da sorgenti differenti ed utilizzati per molteplici scopi, è quindi necessario che rispetti alcune caratteristiche fondamentali di gestione ed ottimizzazione:

- **Subject-Oriented**: il DWH è focalizzato alla gestione ed organizzazione dei dati in modo ottimale, in vista e funzione delle necessità dell'azienda che li accumula.
- Dati **Integrati** e **Consistenti**: sorgenti differenti sono costantemente integrate e normalizzate allo scopo di formare una vista unificata e standardizzata.
- **Continua Evoluzione**: i dati sono accumulati incrementalmente mantenendo comunque tutto lo storico. Rispetto ai sistemi transitivi tradizionali, il DWH suddivide nativamente i dati in epoche, permettendo di richiedere informazioni appartenenti a un esatto istante temporale passato.
- **Non Volatili**: un dato archiviato nel DWH non sarà mai cancellato, modificato o sovrascritto. L'architettura, fisica e logica, di un DWH è di conseguenza ottimizzata per gestire poche scritture e molte letture.

- **Single Source of Truth (SSOT)**: Ogni informazione viene memorizzata una e una sola volta, fornendo una visione unica relativa a un determinato istante temporale.

Data Lake

Con Data Lake si intende un insieme di dati nel loro formato naturale, senza alcun passo di normalizzazione. E' quindi normale trovare nello stesso Data Lake dati strutturati (JSON, CSV, ...), dati parzialmente strutturati (email, pdf, ...) o addirittura binari come video e immagini.

L'ideologia alla base dei Data Lake si rifà ai concetti di Eterogeneità e Veracità dei Big Data, è più conveniente infatti ingerire tutti i dati possibili, senza preoccuparsi della loro normalizzazione, per poi gestire i casi specifici in modo Subject-Oriented. I dati finiscono per essere organizzati in naturali Data Blobs, secondo la loro struttura, sta poi al sistema di ETL a valle il compito di leggerli secondo lo schema più utile al fine del business, si dice infatti che i Data Lake sono **schema-on-read**.

Data Mart

Il Data Mart è un sistema di storage che contiene i dati specifici per un determinato obiettivo di business, i dati sono presi dal Data Lake, trasformati nel formato più opportuno all'analisi e salvati nel Data Mart. In letteratura vi sono pareri contrastanti riguardo la relazione tra Data Lake e Data Mart:

- Secondo **Inmon**, i DWH sono un'aggregazione normalizzata di tutti i dati del business. In quest'ottica diventa molto semplice il processo di gestione del Data Warehouse, tutti i dati hanno infatti la stessa struttura e, di conseguenza, gli stessi bisogni di estrazione, trasformazione, caricamento e gestione. Il compito di assegnare la struttura più specifica per il singolo business sta ai Data Mart, ridotti a semplici viste sui dati.

- Secondo **Kimball**, invece, i DWH sono generati dall'aggregazione dei singoli Data Mart. I dati sono sempre e comunque suddivisi per aree di competenza, senza alcun livello di normalizzazione. Questa visione rende più complessa la gestione del DWH, a causa della forte eterogeneità dei dati, ma semplifica l'analisi permettendo di elaborare i dati a livello generale, grazie alla separazione per aree di competenza.

2.1.3 Comuni Architetture Logiche

Come già discusso, gestire grandi quantità di dati può essere molto problematico, di conseguenza sono state individuate alcune architetture standard:

Single Layer Architecture

Tutte le architetture, se osservate ad un alto livello di astrazione, si basano sulla Single Layer Architecture. In essa il **Data Warehouse Layer** è visto come un blocco unico che si occupa di gestire ed immagazzinare i dati in modo efficiente, secondo le necessità dei Big Data. I dati inseriti nel DWH provengono da un **Source Layer** che si occupa dell'ingestion e preprocessing. A valle del Data Warehouse Layer, si trova l'**Analysis Layer** responsabile dell'analisi dei dati secondo i fini del business aziendale.

Two Layer Architecture

In questo secondo modello si mette in evidenza la separazione tra processi di tipo transazionale, legati cioè alla raccolta e ingestione dei dati nel DWH, e processi di tipo puramente gestionale ed analitico.

- **Source Layer**: livello puramente fisico dello stack, consiste nell'insieme di tutte le sorgenti di dati, fisicamente memorizzati nella loro struttura originale.

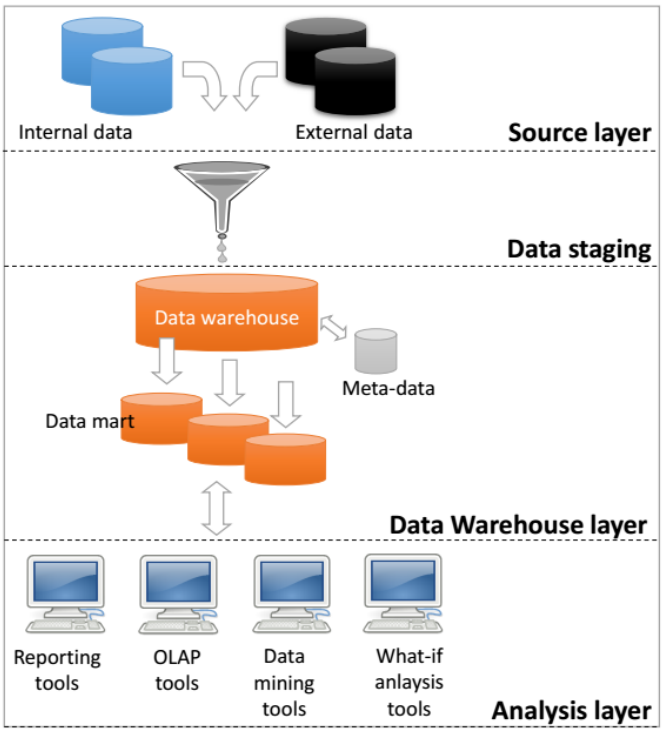


Figura 2.1: Two Layer Architecture

Rispetto al modello precedente, viene messa in risalto la separazione tra **Dati Interni**, raccolti dall'organizzazione proprietaria del DWH (dati di vendita, anagrafiche degli acquirenti,...) e **Dati Esterni**, provenienti da fonti esterne all'organizzazione (lista dei prodotti in voga secondo la moda del momento, ...).

- **Data Staging Layer**: livello in cui si svolge il processo di ETL prima di inserire i dati nel Data Warehouse. Il processo di **Estraction Transformation and Loading** consiste nel estrarre i dati dalla sorgente, trasformarli in modo da adattarli alla struttura del DWH e infine salvarli nel Data Lake.

- **Data Warehouse Layer**: Il DWH non è rappresentato come un unico monolite ma come l'insieme di un Data Lake comune e molteplici Data Mart

specifici per ogni applicazione. Si aggiungono inoltre i Metadata, descrittori dei dati che ne facilitano la gestione ed analisi.

- **Analysis Layer**: Alla base dello stack si trova il livello di analisi, in questo modello sono messi in risalto i tipi di analisi che è possibile eseguire sui dati.

  - **Reporting Tools**, permettono di visualizzare i risultati tramite viste mirate al Decision Making;
  - **OLAP Tools**, On-Line Analytical Process, insieme di tecniche che permettono di analizzare i dati in modo interattivo, real-time (o almeno nearly real-time);
  - **Data Mining Tools**, insieme di strumenti, tecniche ed algoritmi volti ad estrarre informazioni dai dati, solitamente sono applicati in funzione di una determinata applicazione;
  - **What-if analysis Tools**, insieme di tecniche che permettono di prevedere l'occorrenza di eventi basandosi sullo storico degli eventi passati.

Three Layer Architecture

Il Modello a tre livelli è molto simile a quello precedente, rimangono invariati i livelli di Source, Staging, Data Warehouse e Analysis, viene però messo in risalto il processo di trasformazione, parte integrante del ETL. Nelle organizzazioni di grandi dimensioni un problema ricorrente è l'integrazione nel Data Warehouse dei dati (interni) provenienti da differenti Business Unit, dedicate a diversi tasks. Questi dati sono solitamente sufficientemente simili da poter essere confusi da un operatore umano (e quindi potenziale fonte di errori) ma sufficientemente differenti da essere non consistenti o non integrabili.

Il **Reconcilied Layer** mette in luce, a livello di modello, il processo di Trasformazione volto ad ottenere dati Integrati, Consistenti, Corretti, Aggiornati e Dettagliati, utili per il business aziendale. I dati provenienti dalle varie sorgenti vengono estratti e trasformati al fine di modellarli secondo una struttura normalizzata tra le varie Business Unit aziendali. Mettere in risalto questo livello, dato per scontato nei modelli precedenti, previene la definizione di processi di Trasformazione differenti per diverse sorgenti o organizzazioni interne, eliminando i problemi di integrazione alla radice.

2.1.4 Architettura Fisica

Per poter soddisfare i requisiti di sicurezza, affidabilità e prestazioni, si è scelto di gestire i dati in modo replicato e distribuito, costituendo una rete di calcolatori detta Cluster.

Cluster e replicazione

Per **Cluster** si intende un insieme di calcolatori connessi in rete, chiamati **Nodi**, che collaborano per raggiungere un obiettivo comune. I nodi del cluster solitamente condividono le specifiche software e hardware allo scopo di minimizzare lo sforzo tecnico di gestione e standardizzare gli ambienti di sviluppo, test e produzione.

All'interno del Cluster i dati sono gestiti in modo replicato e distribuito:

- Memorizzare lo stesso dato su molteplici nodi permette di gestire in maniera preventiva eventi, casuali e/o intenzionali, che potrebbero compromettere la stabilità del sistema.

- L'elaborazione distribuita permette invece di suddividere il carico di lavoro, gestendo la grande quantità e complessità dei dati senza sovraccaricare una

singola macchina. Lo svantaggio sta nella impredicibilità della sequenza di elaborazione.

Il più comune esempio di architettura logica e fisica per la gestione di un Data Warehouse è l'ecosistema Apache Hadoop. Questo framework offre una moltitudine di tool che coprono tutte le necessità di gestione, elaborazione, monitoring e deploy di sistemi e servizi Big Data.

2.2 Ecosistema Hadoop

Framework che permette di eseguire applicazioni Big Data su un cluster composto da nodi a basso costo, Apache Hadoop è composto da una grande varietà di tool ed applicativi con lo scopo di coprire tutti campi del universo Big Data, dall'estrazione, alla memorizzazione e gestione, sino all'analisi e interrogazione dei dati.

Alla base dell'ecosistema si trovano 3 applicativi, grazie ai quali è possibile gestire i Big Data rispettando i requisiti suddetti.

2.2.1 Hadoop Distribuited File System (HDFS)

Requisito fondamentale per utilizzare Hadoop è avere a disposizione un cluster su cui memorizzare e gestire i dati, HDFS è un file system distribuito, basato su Java, pensato per lavorare su hardware di medio livello. Non serve quindi un calcolatore dotato di eccessiva potenza di calcolo o grande capacità di memorizzazione per eseguire HDFS, grazie alle caratteristiche di distribuzione e replicazione si fa fronte alla potenza con il numero.

HDFS si basa su alcune assunzioni fondamentali:

- Il **fallimento è la norma non una eccezione**, nel momento in cui si sfruttano grandi quantità di istanze piuttosto che una unica istanza potente, è

necessario assumere i fallimenti come parte integrante del progetto. HDFS si prefigge di rilevarli e correggerli in maniera trasparente.

- HDFS è pensato per gestire **processi batch** su **grandi quantità di dati**, massimizza infatti il throughput piuttosto che la bassa latenza. Avendo in mente questa specifica, HDFS è stato progettato per gestire in modo efficiente grandi file, frammentandoli e replicandoli in maniera intelligente. Lo svantaggio sta nella frammentazione in blocchi di dimensione fissa: avendo un file di piccole dimensioni, minori di quelle standard, è comunque necessario allocare un intero blocco, con un ovvio overhead e spreco di risorse. HDFS non è quindi pensato per gestire applicativi real-time o di interazione con l'utente.
- **Write-Once-Read-Many**, una volta che il file è stato creato, scritto e chiuso esso non verrà mai modificato, soltanto letto. In questo modo è possibile garantire un alto throughput in lettura.
- **Distribuire la computazione è più economico che Spostare i dati**: dovendo gestire grandi quantità di dati, è più efficiente distribuire la computazione il più possibile vicino ad essi piuttosto che muoverli dal nodo in cui sono memorizzati. Di conseguenza è necessario un sistema per distribuire i componenti infrastrutturali necessari per una corretta esecuzione dei processi, affiancato a un sistema di serializzazione dei componenti eseguibili.

L'architettura di gestione di nodi del cluster è di tipo Master-Slave:

Name Node

Il master è chiamato **Name Node**, esso si occupa di gestire il namespace del cluster, memorizzando la locazione di ogni file all'interno del file system distribuito. I Name Node si occupano inoltre di gestire la replicazione e distribuzione dei file tra i vari Data Node. Gli utenti si collegano al Name Node nel momento in cui hanno bisogno

di accedere a un file in HDFS, di conseguenza esso è il singolo punto di accesso per ogni richiesta.

Data Node

Gli slaves sono detti Data Nodes, solitamente sono in grande numero e si occupano di memorizzare e organizzare i namespace locali al nodo.

Lo scopo principale del Data Node è quello di garantire che il tempo di risposta alle richieste di lettura e scrittura sul HDFS rispetti i requisiti dell'infrastruttura.

Infine si occupano di realizzare le istruzioni di creazione, eliminazione e rinomina impartite dagli utenti attraverso i Name Node.

2.2.2 Map Reduce

Map Reduce è il componente che si occupa di distribuire la computazione tra i nodi del cluster. Come suddetto, dovendo gestire grandi quantità di dati, è più conveniente muovere le unità di elaborazione verso i Data Node, piuttosto che spostare il file in un nodo di computazione. Grazie a Map Reduce è possibile garantire che i processi di elaborazione vengano serializzati, eseguiti in locale sul determinato Data Node, e il loro risultato venga trasmesso al Client che l'ha richiesto.

2.2.3 YARN

Componente responsabile della gestione delle risorse del cluster e dello scheduling dei componenti procedurali, detti **Jobs**, tra i nodi a disposizione nella rete.

YARN è composto, in maniera simile a HDFS, da due componenti principali:

- Il **Resource Manager** è il master del framework di elaborazione, si occupa di decidere su quale nodo deve essere eseguito il determinato Job (Scheduling) secondo le caratteristiche suddette. Si occupa inoltre di comunicare con i singoli

nodi, monitorando lo stato di esecuzione del job e le risorse a disposizione del sistema.

- Il **Node Manager** è invece collocato sui nodi di elaborazione, si occupa di comunicare con il Resource Manager, ricevere ed eseguire i job, secondo un framework container-based, monitorandone lo stato di esecuzione e di utilizzo risorse.

L'ecosistema Apache Hadoop è composto da innumerevoli altri tools e applicativi, volti a coprire ogni necessità del Big Data Engineering, in questo elaborato ho sfruttato Cloudera CDH per accedere alle potenzialità di Hadoop.

2.3 Cloudera CDH

Cloudera CDH è una piattaforma di distribuzione Open Source dell'ecosistema Hadoop, si occupa inoltre di integrare il framework con tools di terze parti, rendendola un'ottima piattaforma per sviluppare applicativi Big Data a livello Enterprise. Mi limiterò a descrivere brevemente i tools utilizzati in questo elaborato, soffermandomi su quelli di maggiore interesse.

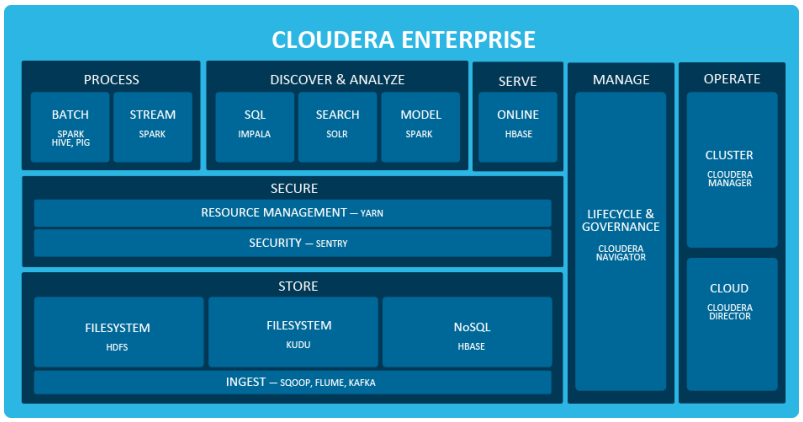


Figura 2.2: CDH Stack

2.3.1 Apache Hive

Apache Hive è un software, sviluppato da Facebook nel 2008, che permette di leggere, scrivere ed organizzare i dati di grandi dataset, quali i DWH, con un linguaggio di query SQL-like. L'elaborazione di Hive si basa sul concetto di Live Long and Process (LLAP) cioè sul mantenere uno stato di cache intelligente della query e del suo risultato, in modo da rendere più efficiente l'interazione multipla con lo stesso sottoinsieme di dati. Di conseguenza, come HDFS sul quale si appoggia, è ottimizzato per eseguire query batch su file di grandi dimensioni.

Il vero vantaggio di Hive sta nel **Hive Query Language**, il linguaggio di query SQL-like. Tramite l'interfaccia di Hive è infatti possibile vedere il dataset non come un file system ma come un sistema di tabelle del tutto simile a un RDBMS. Rispetto ai sistemi tradizionali vi sono però due sostanziali differenze: non esistono i concetti di "relazione" tra strutture dati e "chiave" di identificazione, di conseguenza sono ammessi elementi duplicati.

2.3.2 Apache Impala

Apache Impala [27] è un Massive Parallel Processing (MPP) SQL engine, open source, che mira ad offrire un'interfaccia SQL-like, del tutto simile a Hive, per accedere ai dati memorizzati in un DWH. Mentre Hive è ottimizzato per eseguire query batch su molti dati, Impala si focalizza su rapide query a bassa latenza e alto parallelismo, aprendo la strada al real-time in ambito Hadoop.

Per garantire un tale livello di bassa latenza, Impala non sfrutta Map Reduce ma si basa su un'architettura distribuita, basata su processi daemon che gestiscono l'esecuzione delle query sul singolo nodo del cluster, accedendo direttamente ai file. Impala è fisicamente composto da tre servizi:

- **Impala Daemon**: è il servizio che si occupa di accettare le query dal cliente

e di eseguirle, rimandando l'esecuzione effettiva ai singoli nodi del cluster. Il demone risiede su ogni nodo, l'assegnamento del ruolo di master dipende dalle esigenze della singola query.

- **Statestore Daemon**: si occupa di gestire il trasferimento dei metadati legati ai file accessibili da Impala secondo il protocollo Publish-Subscribe. Vi è un unico Statestore daemon all'interno del cluster.
- **Catalog Daemon**: è il gateway di accesso al metastore dei metadata riguardo i dati, ogni cambiamento viene propagato tra tutti i nodi del cluster.

Grazie ai metadata, Impala riesce a descrivere saggiamente i dati e la loro distribuzione all'interno del cluster, eseguendo query sul demone locale e restituendo i risultati con bassa latenza. Lo svantaggio sta nella staticità di questi metadati, a ogni cambiamento del dataset è infatti necessario invalidarli e rigenerarli, in modo che Impala riesca ad accedere alla versione aggiornata.

2.3.3 Apache Kudu

Apache Kudu [26] è uno storage layer che memorizza i dati in forma colonnare in modo fortemente tipizzato, mantenendo tutte le caratteristiche di hardware, scalabilità e accesso frequente di Hadoop. Grazie alla sua struttura colonnare, Kudu si colloca come una via di mezzo tra la rapidità e comodità di accesso SQL-like di Hive e la gestione a indici si Apache HBase, rendendolo uno strumento ottimale per accedere a grandi quantità di dati, molto simili tra di loro in breve tempo.

Struttura Colonnare

I vantaggi di avere una struttura colonnare fortemente tipizzata sono molteplici, soprattutto nel campo dell'analisi dove è uso comune accedere a poche colonne su una grande quantità di dati:

- Ogni tabella Kudu ha uno schema ben definito, è composta da un numero finito di colonne, ciascuna con un nome e un tipo. Questa struttura schematica si ripercuote sul formato dei dati memorizzati, rendendone semplice l'integrazione nel Data Warehouse;
- Kudu indicizza e permette di accedere in maniera diretta (Random Access) alle singole colonne del dataset;
- Avendo colonne con tipo ben definito, è possibile ottimizzare la memorizzazione dei dati ad esempio adottando un determinato tipo di codifica per una colonna piuttosto che un'altra. In questo modo i file risultati sono più compressi, ottimizzando sia la gestione delle risorse sul file system che la rapidità di risposta alle query (dovendo leggere meno blocchi dal disco);
- A differenza di Impala e Hive, Kudu integra nuovamente il concetto di chiave primaria, in questo modo si garantisce l'unicità e non ridondanza del dato, aprendo inoltre le porte ad update e delete sul singolo elemento della tabella. Kudu ottimizza inoltre il processo di **upsert**, tecnica che unisce il vantaggio di insert e update in un unico comando.
- I dati sono nativamente strutturati secondo partizioni, l'utente infatti è obbligato a indicare il metodo di partizionamento (basato su hash) che ritiene più opportuno in base alle chiavi inserite. Questa caratteristica, associata alla forte tipizzazione, permette di frammentare i file in maniera efficiente e mirata tra i nodi del cluster.

Architettura

L'unità fondamentale dell'architettura Kudu è il **Tablet**, singolo frammento di una tabella replicato e distribuito tra i nodi del cluster. Ogni Tablet mantiene informazioni relative agli altri frammenti della tabella originale. Due sono i ruoli principali

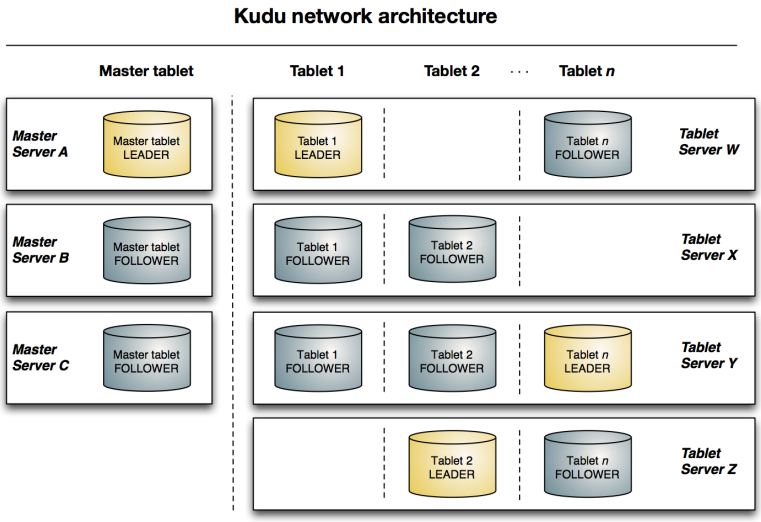


Figura 2.3: Architettura Kudu

che permettono di accedervi:

- I **Tablet Server** memorizzano e premettono l'accesso ai Tablet. Per ogni richiesta del cliente, il Tablet Server che contiene i dati di interesse agisce come leader, ricevendo ed eseguendo le richieste di scrittura sul Tablet. Le letture invece coinvolgono anche gli altri Tablet Server che memorizzando i Tablet legati al leader. Di conseguenza un singolo Tablet Server può servire più Tablet.

- Il **Master** tiene traccia di tutti i Tablet, i Tablet Server e i metadati relativi al cluster (memorizzati in Catalog Tables). Il ruolo di master è, ovviamente, dinamico, per ogni richiesta esiste un solo master, ma il suo ruolo può passare a qualsiasi Tablet Server in base alle necessità di efficienza e sicurezza. Il Master si occupa inoltre di soddisfare le richieste di gestione delle tabelle.

2.3.4 Apache Sqoop

Sqoop è un tool, accessibile da riga di comando, pensato per trasferire in maniera efficiente grandi qualità di dati (bulk) tra datastore strutturati, come un database relazionale, e HDFS. Facendo parte dell'architettura Cloudera può trarre vantaggio da replicazione e distribuzione tipiche del sistema, è quindi possibile eseguire estrazioni parallele, gestite in maniera efficiente dal load balancing di YARN.

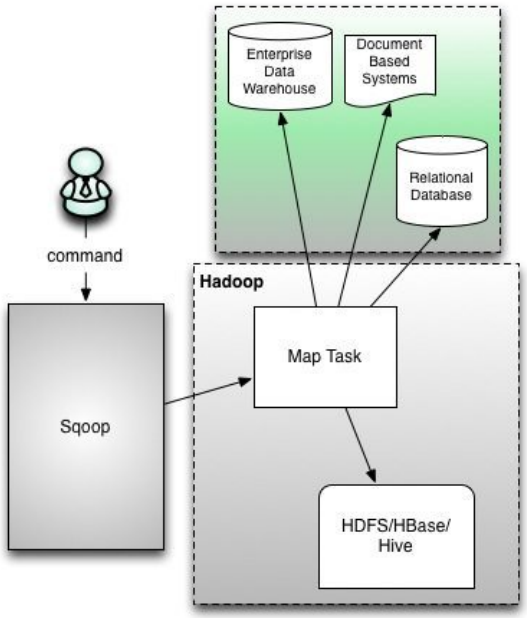


Figura 2.4

Sqoop gestisce le connessioni di input e output in maniera modulare: grazie una vasta gamma di connettori, totalmente estendibili, è possibile utilizzare sqoop con una grande varietà di sistemi esterni.

2.3.5 Apache Spark

Apache Spark è una piattaforma di cluster computing pensata per essere veloce e general-purpose. Spark estende ed ottimizza il modello procedurale di MapReduce

aggiungendo più tipi di computazione, query reattive ed interattive e processi streaming real time. Allo scopo di massimizzare la velocità, Spark aggiunge la possibilità di eseguire computazioni in-memory, il determinato file può infatti essere salvato (cached) e su di esso possono essere eseguiti tutti i passaggi di elaborazione senza la necessità di accedere alla rete o al disco. Spark permette inoltre di integrare differenti tipi di processo all'interno dello stesso flusso di esecuzione, in questo modo è possibile eseguire tutte le elaborazioni richieste dai tradizionali use cases Big Data con un unico tool.

Spark è distribuito come una libreria open source, accessibile in numerosi linguaggi, i più usati sono comunque Python e Scala grazie ai quali Spark può avvalersi delle grandi potenzialità della programmazione funzionale.

Resilient Distribuited Dataset (RDD)

Il RDD è l'unità di dato fondamentale per l'elaborazione di Spark, esso rappresenta una collezione immutabile e distribuita di oggetti. Ogni RDD è suddiviso in molteplici partizioni, ciascuna elaborabile da un nodo differente del cluster, secondo le necessità di efficienza e carico di lavoro del caso. Un RDD può essere creato da file (con la possibilità di accedere ad innumerevoli formati) oppure da collezioni di dati, che vengono parallelizzati e distribuiti.

Operazioni su RDD

Sul dataset è possibile eseguire molte operazioni, oltre alle comuni map e reduce fornite da Map Reduce, il concetto fondamentale da ricordare lavorando con gli RDD è la **Lazy Evaluation**, esistono infatti due tipi di operazioni eseguibili su un RDD:

Trasformazioni La *Map* è un tipico esempio di trasformazione, essa prende un RDD e ne modifica la struttura secondo una determinata funzione di trasferimento, definita dall'utente, restituendo un nuovo RDD e lasciando inalterato quello in input. Esempi di trasformazioni sono: *map(), flatMap(), filter(), select(), distinct(), union(), ...*

Azioni La *Reduce* è invece un tipico esempio di azione, essa prende un RDD e genera un risultato (potenzialmente anche un RDD) secondo una funzione definita dall'utente. Un esempio limite di azione è la *collect()*, essa permette di salvare un RDD in memoria abbandonando la sua struttura a favore di una dipendente dal linguaggio in uso (solitamente una lista o derivati). Questa operazione è comoda quando le trasformazioni ed azioni hanno ridotto il dato a un insieme piccolo di elementi, facilmente gestibili con i tradizionali sistemi di elaborazione. Collezionare un dataset grezzo significherebbe salvare in memoria (locale) una grande quantità di dati solitamente non gestibile.

Secondo la **Lazy Evaluation**, l'esecuzione di Spark è pensata per accedere ai dati solo quando effettivamente necessario, di conseguenza, pensando ai tipi di operazioni suddetti, è necessario sapere l'effettivo contenuto del dato solo alla restituzione di un valore finale, cioè in corrispondenza di una Azione. Le trasformazioni vengono quindi accumulate in maniera intelligente ed applicate in blocco, minimizzando il carico computazionale. Lo svantaggio di questo tipo di elaborazione sta nella necessità di ricostruire lo stato del RDD ad ogni applicazione di azioni; per facilitare questo processo è possibile memorizzare lo stato di un RDD tramite il comando *cache()*, verrà quindi creato uno snapashot del RDD mantenendo statiche le azioni e trasformazioni applicate fino a quel momento.

Sebbene Spark permetta di potenziare ulteriormente le capacità di MapReduce, resta comunque dipendente dalla sua struttura e paradigma di elaborazione, di conseguenza anche l'utilizzatore in fase di programmazione deve mantenere ben presente il funzionamento del engine. Ogni componente utilizzato nella definizione della business logic interna a un processo di Map Reduce deve essere completamente serializzabile.

Un altra possibile fonte di errore e l'estrazione di metriche all'interno della funzione di business logic, tali valori non potranno mai avere valenza globale dato che saranno locali e relativi al solo nodo (non deterministico) sul quale sta avvenendo l'elaborazione.

Ecosistema Apache Spark 2.x

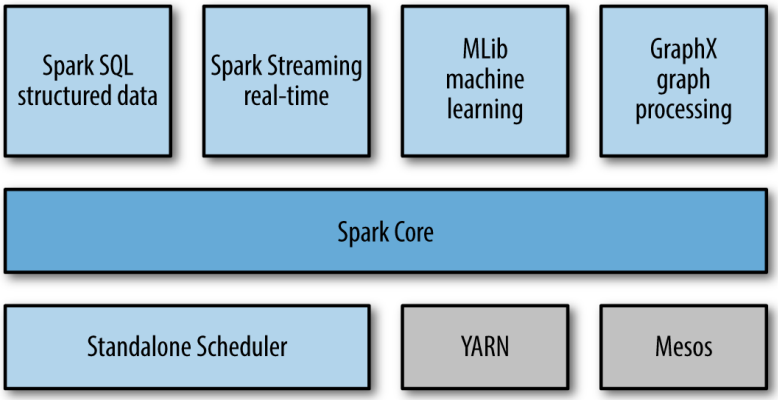


Figura 2.5: Spark stack

L'ecosistema Spark include numerose librerie che coprono le varie sfaccettature del mondo Big Data in uno Stack unificato e Open Source:

Spark Core Il core di Apache Spark non è altro che uno scheduler di job associato a un gestore di risorse, include la definizione di RDD e delle operazioni suddette.

GraphX GraphX permette di lavorare su grafi di interconnessione tra elementi.

Cluster Managers Il Cluster Manager permette di gestire la scalabilità e le risorse dei jobs in esecuzione, YARN è un esempio di cluster manager.

2.3.6 Apache Spark SQL

Apache Spark SQL permette di elaborare dati di qualsiasi tipo come se fossero in struttura tabellare. Integrando la conoscenza della struttura del dato da elaborare nel engine di esecuzione, è possibile ottimizzare il processo per il determinato tipo di dato. Spark SQL permette quindi di accedere alle informazioni utilizzando il linguaggio SQL o HQL, considerandole come se fossero tabelle a tutti gli effetti, in questo modo è possibile processare Dataframes, tabelle locali o addirittura tabelle Hive grazie all'integrazione Hadoop.

L'elemento base, introdotto in Spark 1.6, è il **Dataset**, un insieme di dati del tutto uguale a un RDD, al quale si aggiungono informazioni legate alla struttura del dato in esso contenuto.

DataFrame

Spark SQL potenzia ulteriormente la definizione di Dataset organizzando i dati, già strutturati, in un sistema di colonne nominali del tutto simile a una tabella in un database relazionale. Sebbene i DataFrame aggiungano la possibilità di eseguire query SQL, si ha la perdita di buona parte delle operazioni applicabili agli RDD, di conseguenza questo tipo di dato è molto utile solo quando sia ha che fare con dati naturalmente memorizzati, o da memorizzare, all'interno di storage database-like.

2.3.7 Apache Spark Streaming

Apache Spark Streaming è il componente Spark che permette di processare stream in maniera scalabile, garantendo alto troughput e sicurezza al fronte di fallimenti. Spark Streaming è pensato per ingerire dati provenienti da innumerevoli sorgenti differenti, processarli e metterli a disposizione per successive elaborazioni.

Figura 2.6: Spark Streaming

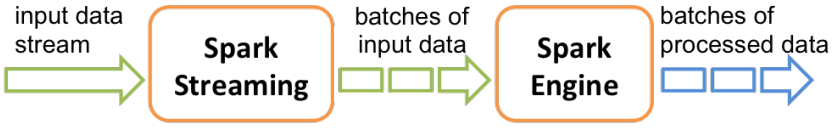


Figura 2.7: Spark Streaming

Come si può notare in Figura 2.7, Spark Streaming non è esattamente uno Stream-Engine, esso infatti accumula i dati in blocchi di dimensione fissa per poi elaborarli in processi batch secondo le specifiche dell'utente. Si noti inoltre come l'elaborazione effettiva del dato sia lasciata allo Spark Engine, con tutti i vantaggi e svantaggi del caso ma, il più delle volte, in maniera trasparente allo stream.

Discretized Streams (DStream)

Suddividere lo stream continuo e real time in blocchi batch consiste di fatti nel discretizzarlo, il DStream è l'unità di dato alla base dell'elaborazione di Spark Streaming

e rappresenta il dato nearly real time in ogni fase della sua ingestione, elaborazione e trasmissione. A livello pratico, un DStream non è altro che una sequenza di RDD associati a un determinato intervallo temporale di campionamento.

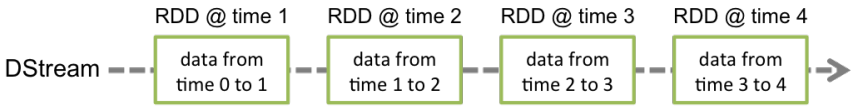


Figura 2.8: DStream come sequenza di RDD

Elaborazione

Il vantaggio di vedere uno stream come un'astrazione del tutto simile a collezioni di RDD, permette di applicare al DStream le stesse operazioni (trasformazioni ed azioni) applicabili a un RDD. Menzione speciale merita la *transform()*, questo metodo permette di applicare una sequenza di trasformazioni ai singoli RDD del DStream. Il tutto mantenendo l'astrazione fornita dai DStream.

Output

La Destinazione dell'elaborazione dipende dalla specifica implementazione e, a causa della sua natura estremamente general-purpose, è leggermente tralasciata all'interno della libreria. Sono forniti alcuni metodi per salvare su file di testo, su Hadoop o come oggetto, oltre al generico *foreachRDD* che applica una operazione ad ogni RDD del DStream di fatto perdendo l'astrazione di stream discretizzato in favore della futura elaborazione.

Input

Per quanto riguarda invece le sorgenti di Stream, punto focale di Spark Streaming, il modulo include due tipologie principali:

Sorgenti Semplici Sono tutte le sorgenti nativamente presenti in qualsiasi ambiente di elaborazione, indipendentemente dalla specifica applicazione:

- **File Streams**: Spark Streaming monitora una directory, con l'unico requisito che il file system di appartenenza sia compatibile con HDFS, processando ogni nuovo file inserito. Lo svantaggio è la sensibilità solo a nuovi file, quindi non è un approccio adatto a file scritti in modo incrementale.
- **Custom Receiver**: E' possibile far fronte al problema suddetto implementando un Custom Receiver, un componente che, ad esempio, resta in ascolto di ogni scrittura su file trasmettendo allo stream gli aggiornamenti.
- **Coda di RDD**: A fini di debug è possibile creare una coda sulla quale Spark Streaming si mette in ascolto.

Sorgenti Avanzate Sono Sorgenti dipendenti dalla specifica applicazione e, di conseguenza, utilizzabili tramite aggiunta di ulteriori librerie e moduli. Alcuni esempi di tali sorgenti sono: Flume, Kinesis e Kafka.

2.3.8 Apache Spark Machine Learning Library (MLlib)

MLlib è il modulo Spark che permette di fare Machine Learning sfruttando le potenzialità di gestione dati di Spark. MLlib è fornita con un set di algoritmi che coprono la stragrandes maggioranza di use cases Machine Learning, ad esempio: classificazione, regressione, collaborative filtering, clustering, dimensionality reduction, neural networks.

Vantaggi dell'integrazione Spark

Computazione Distribuita I processi di Machine Learning sono solitamente batch e iterativi, di conseguenza richiedono che un'istanza si prenda carico dell'elaborazione per un lungo periodo. MLlib, basandosi su Spark, evita il problema alla radice, l'elaborazione infatti è distribuita sui vari nodi del cluster, garantendo inoltre scalabilità orizzontale e la possibilità di usare hardware di medio livello.

Pipeline integrate Solitamente il modello di Machine Learnig è il risultato di una sequenza di operazioni che permettono di estrarre le informazioni significative per il determinato caso d'uso partendo dai dati grezzi. MLlib integra nativamente questo concetto di Pipeline utilizzando come elemento fondamentale di elaborazione il DataFrame, con tutti i vantaggi di accesso SQL suddetti.

Ottimizzazione Gli algoritmi sono ottimizzati per lavorare con DataFrame o RDD (a seconda della scelta progettuale del caso), ad esempio l' Alternating Least Square (ALS), utilizzato in questo elaborato e successivamente discusso esaustivamente, computa raccomandazioni cercando di utilizzare il minor numero di blocchi di memoria possibile, allo scopo di minimizzare l'esecuzione del garbage collector della JVM.

2.4 Confluent Platform

Confluent è una piattaforma di streaming che permette di gestire in maniera semplice ed efficace la grande quantità di messaggi ed eventi interscambiati a real time tra i componenti di un sistema

I sistemi moderni sono ormai composti da un grande numero di componenti inoltre, con la venuta dei microservizi, il numero è aumentato esponenzialmente,

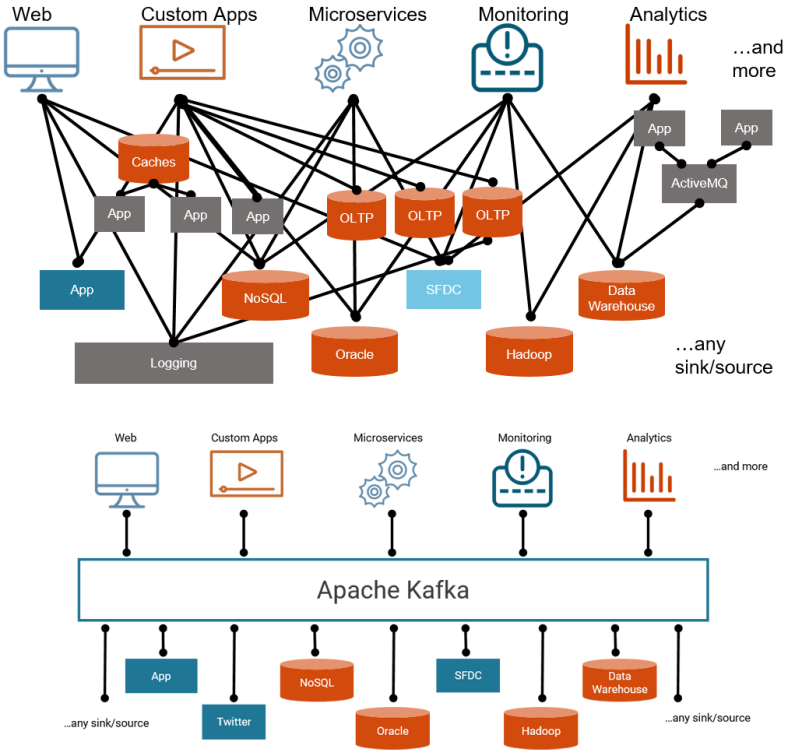


Figura 2.9: Confluent come sistema di messaggistica real time

diventa quindi impensabile gestire l'interazione N-N tra tutti i componenti del sistema con le tecniche tradizionali. Confluent si colloca esattamente in questo use case, apportando la possibilità di gestire il tutto con un unico sistema distribuito di publish-subscribe a Topic.

Confluent offre inoltre un gran numero di connettori di Source e di Sink che permettono di integrare la piattaforma con le più comuni infrastrutture presenti sul mercato. Infine, grazie alla sua componente open source, chiunque può creare il proprio connettore ed integrarlo in maniera facile nell'infrastruttura.

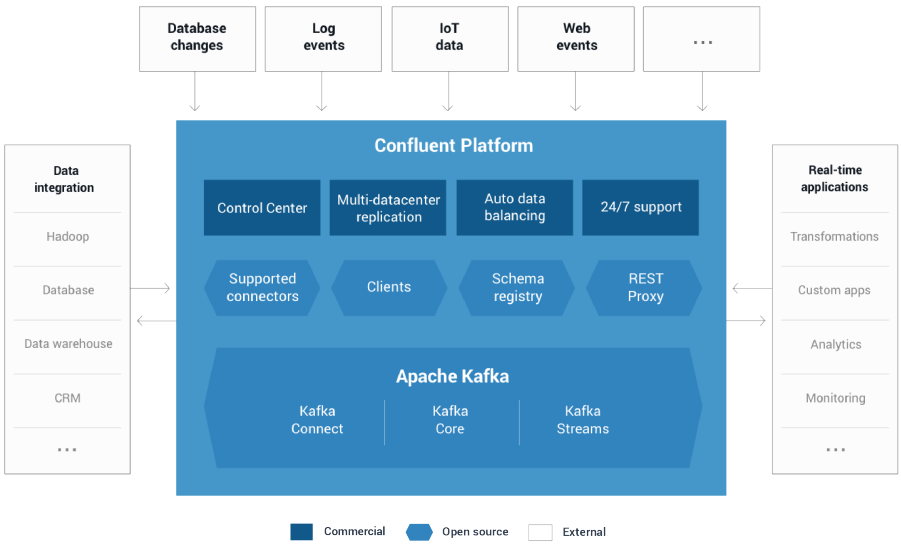


Figura 2.10: Architettura Confluent

2.4.1 Kafka

Kafka, il cuore pulsante di Confluent, è una piattaforma di streaming distribuito che sfrutta il modello publish-subscribe. Inizialmente sviluppato da Linkedin come gestore di log di sistema, è stato ceduto ad Apache che ne ha continuato lo sviluppo open source.

Al momento Kafka offre numerose features:

- Permette di pubblicare stream di elementi (messaggi, eventi, log, ...) di qualsiasi tipo e struttura.

- Organizza gli stream in un sistema di Topics a cui chiunque può sottoscriversi per ricevere tali messaggi.

- Permette di memorizzare gli stream di elementi in modo distribuito, replicato e tollerante agli errori secondo gli standard Big Data.

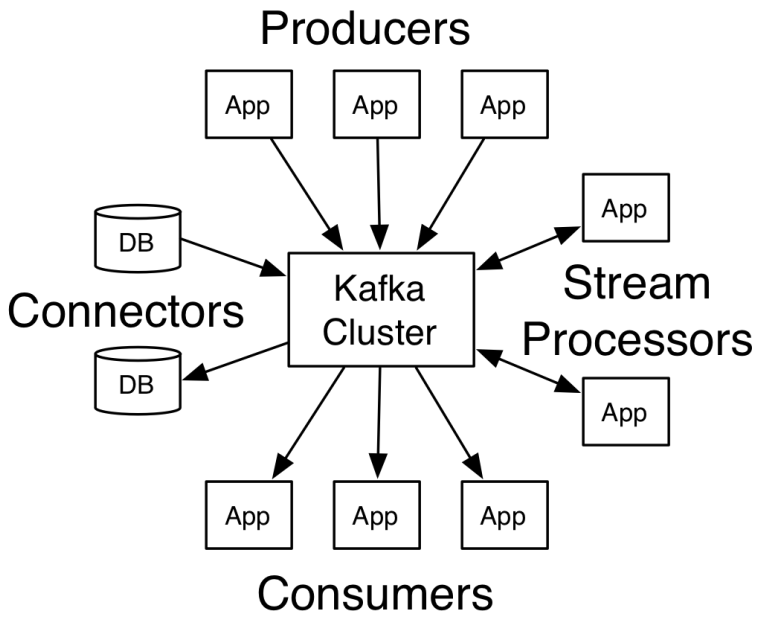


Figura 2.11: Kafka Features

- Permette di processare gli elementi dello stream in maniera del tutto trasparente per il consumatore.

Kafka inoltre garantisce che:

- I records mandati da un produttore siano inseriti nell'ordine di invio;
- I Consumatori leggono i records nell'esatto ordine in cui sono stati inseriti;
- Ogni messaggio è presente e quindi consumato una sola volta;
- Supponendo di avere N repliche dei dati, è tollerata la perdita di N-1 di queste repliche senza compromettere i records.

Topics

Ogni elemento dello stream è strutturato come una tripla composta da *Chiave*, *Valore*, *Timestamp*, in questo modo è possibile identificare in maniera univoca

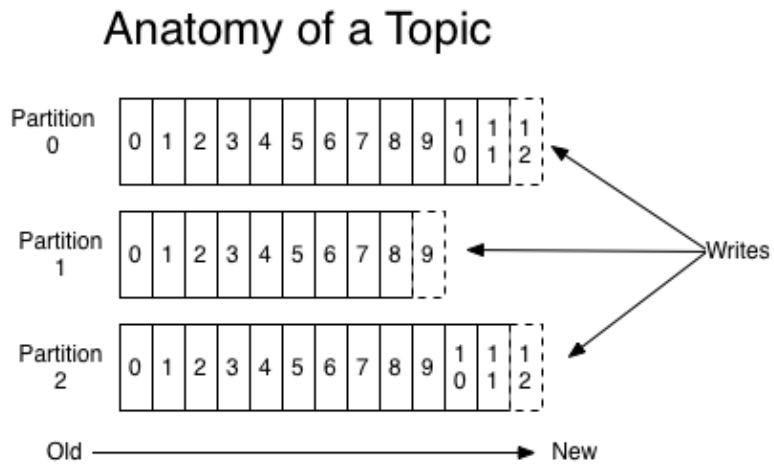


Figura 2.12: Topic

l'elemento.

Per Topic si intende un'insieme di elementi dello stesso tipo, strutturato come una coda. Un Topic è sempre inteso come multi-subscriber, questo significa che, in ogni istante temporale, può esserci un numero non definito di consumatori che leggono gli elementi pubblicati.

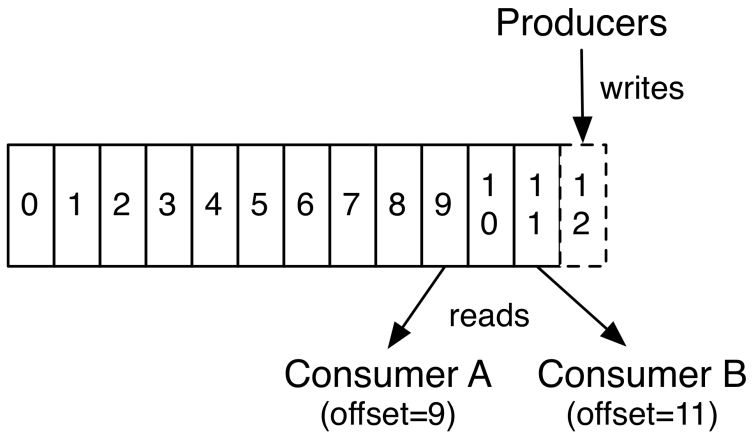


Figura 2.13: Lettura parallela di un Topic

Offset Si noti inoltre che un Topic è nativamente partizionato e distribuito, la chiave suddetta è infatti detta **Offset** ed è relativa alla singola partizione. Tale valore è lo stesso utilizzato dai consumatori per leggere i record dai topics, ed è memorizzato in locale presso il consumatore e aggiornato ad ogni lettura. Questo significa che un Topic è in realtà una struttura dati ad accesso casuale, potenzialmente il consumatore può accedere a qualsiasi offset desideri, a patto che il record sia ancora presente in memoria. Un record infatti viene mantenuto in Kafka solo per un periodo di retain ben definito, anche molto lungo senza compromettere l'efficienza di Kafka.

Consumatori

Ogni topic Kafka è leggibile da un numero non definito di consumatori, aggiornando in autonomia gli offset; use case standard è quello di avere molteplici consumatori appartenenti allo stesso applicativo che consumano dati dai topic in parallelo.

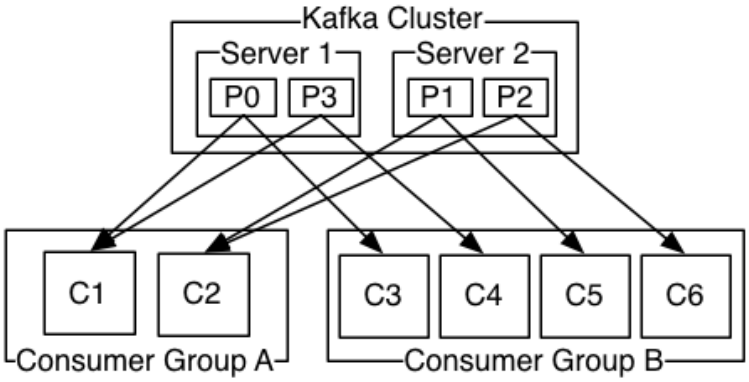


Figura 2.14: Gruppi di Consumatori

Kafka gestisce questo caso tramite i **Consumers Groups**, ogni consumatore infatti deve presentare il nome di gruppo di appartenenza in modo che Kafka possa sapere se esistono altri consumatori dello stesso applicativo ed evitare di fornire più

volte lo stesso record. Per evitare il caso, Kafka semplicemente assegna una partizione differente a ogni consumatore dello stesso gruppo, sfruttando la caratteristica delle chiavi, univoche ed incrementali solo a livello di partizione.

Arricchimento

Infine kafka integra un inseme di API che permettono di modificare, trasformare, arricchire i records presenti nei topics in maniera del tutto trasparente al consumatore. In questo modo non serve implementare un sistema batch che, periodicamente, arricchisce i dati aggregando informazioni da altre sorgenti, il processo può essere fatto in real time fornendo il dato già arricchito direttamente al consumatore.

2.4.2 Confluent Schema Registry

Oltre ad arricchire il sistema di streaming real time distribuito di Kafka con numerosi produttori e consumatori, Confluent aggiunge lo Schema Registry, un sistema che permette a una determinata sorgente di registrare lo schema del dato che sta caricando in modo che i vari consumatori possano conoscerne la struttura ancora prima della connessione al topic, ed adattarsi di conseguenza.

Questa funzione può sembrare banale in piccola scala ma, nell'ottica dello scaling dei servizi di un sistema moderno, è immediato pensare che lo schema dei dati trasmesso sia anch'esso i continua evoluzione. Avere quindi un sistema di metadata centralizzato, versionato e distribuito, assicura che sia possibile effettuare modifiche a livello di produttore senza temere incompatibilità a lato consumatore.

Capitolo 3

DevOps

3.1 Introduzione

Il termine **DevOps**, coniato nel 2008 da Patrick Debois, è il composto aplologico (anche detto "parola macedonia" o "portmanteau") delle parole **Development** e **Operations** rappresentante un movimento, una filosofia, un insieme di tecniche volte ad ottimizzare il processo di sviluppo e rilascio di prodotti software.

Ogni metodo descritto dal movimento non ha come punto focale il prodotto e il suo valore commerciale bensì il soggetto è posto sulle **Persone**.

In questa categoria sono posti tutti gli individui che entrano a far parte del ciclo di vita del prodotto, si collocano non solo il cliente e i consumatori finali, ma anche Manager, Developers, Operations, Testers, IT, QAs, sistemisti, Security, ... Con l'idea che, se le persone lavorano in un ambiente armonico, ben organizzato e privo di stress inutile, allora anche il prodotto finale ne gioverà in termini di qualità e valore per il consumatore.

3.2 Cenni Storici

Per comprendere a pieno le motivazioni e necessità che hanno spinto Patrick Debois a coniare il termine DevOps e a dare il via al movimento, è necessario ricordare qual'era il processo di sviluppo e rilascio del software all'inizio degli anni 2000.

Al tempo, il processo di sviluppo era decisamente più lento, non per la incapacità delle persone, ma per la mancanza di mezzi pratici. Un'azienda standard di dimensione medio-grande rilasciava una nuova versione del proprio prodotto circa una volta all'anno, due per le aziende migliori, la nuova release veniva quindi distribuita tramite CD-ROM, perché all'ora non esisteva ancora un'infrastruttura internet in grado di garantire le velocità di distribuzione e la banda odierna.

3.2.1 Waterfall Model

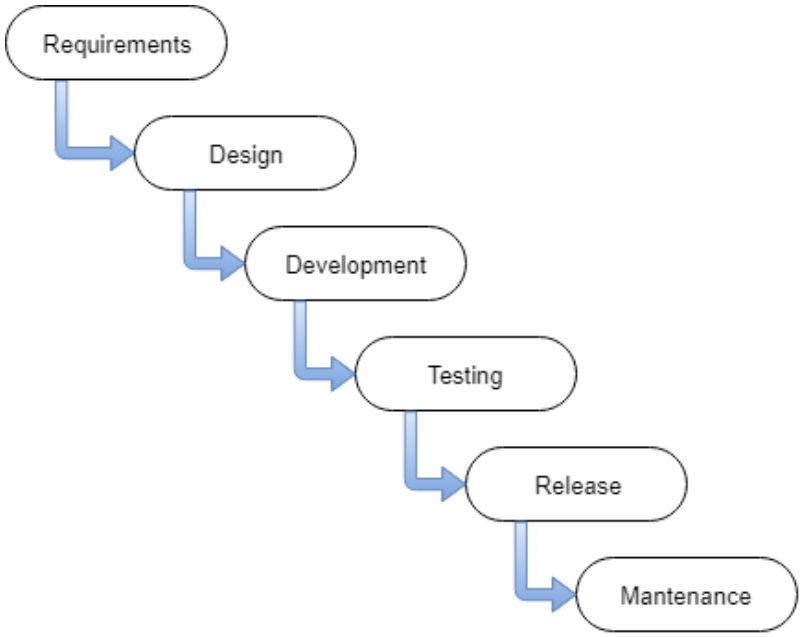


Figura 3.1: Waterfall Model

Il waterfall model è stato il primo modello in uso per lo sviluppo software a

partire dagli anni 60'. Direttamente ereditato dall'industria manifatturiera, presenta un numero più o meno fisso di fasi che portano il prodotto dalla definizione con l'utente fino al suo sviluppo e rilascio. Solitamente il processo è diviso tra Analisi dei Requisiti, Design del modello, Sviluppo, Testing, Release e Maintenance.

Questo modello presenta numerose falle e punti critici, il problema principale però è la totale incapacità di gestire cambiamenti nei requisiti iniziali definiti dal cliente, non è infatti presente alcun tipo di retroazione che permetta di reiterare una fase passata senza dover per forza ripartire dall'inizio del processo di sviluppo.

Inoltre, per poter passare da una fase a quella successiva, e necessario che la prima sia completamente terminata. Per esempio, prima di iniziare i test è necessario che il prodotto sia completo. In caso di test falliti inizia quindi un loop di interazione tra developers e QAs/testers che si trasforma in tempo e denaro persi fino a quando tutti i test non sono validati e il prodotto è pronto per il rilascio.

Infine il feedback del cliente si ha solo e soltanto al termine di tutto il processo, questo significa che un requisito male inteso nella prima fase si propaga per tutto il processo di sviluppo, generando un prodotto invalido.

3.2.2 Metodologia Agile

Nel 2001 nacque la metodologia Agile, un insieme di tecniche e metodi di lavoro volti a risolvere i problemi del modello a cascata.

"We are uncovering better ways of developing software by doing it and helping others do it.

Through this work we have come to value:

Individuals and interactions *over processes and tools*

Working software *over comprehensive documentation*

Customer collaboration *over contract negotiation*

Responding to change *over following a plan*

That is, while there is value in the items on the right, we value the items on the left more."

L'idea alla base di tutto è quella di inserire retroazioni e loop tra le fasi di sviluppo, introducendo il cliente (anche detto Product Owner) e i testers direttamente nella fase di Development. In questo modo è possibile testare il prodotto prima del suo effettivo rilascio finale, ma soprattutto ottenere un feedback da parte del cliente prima che tutto il prodotto sia stato sviluppato.

Per ottenere questo risultato è necessaria una organizzazione aziendale non trascurabile, dalla nascita del modello sono nate molte implementazioni e interpretazioni, prima fra tutte **Scrum**.

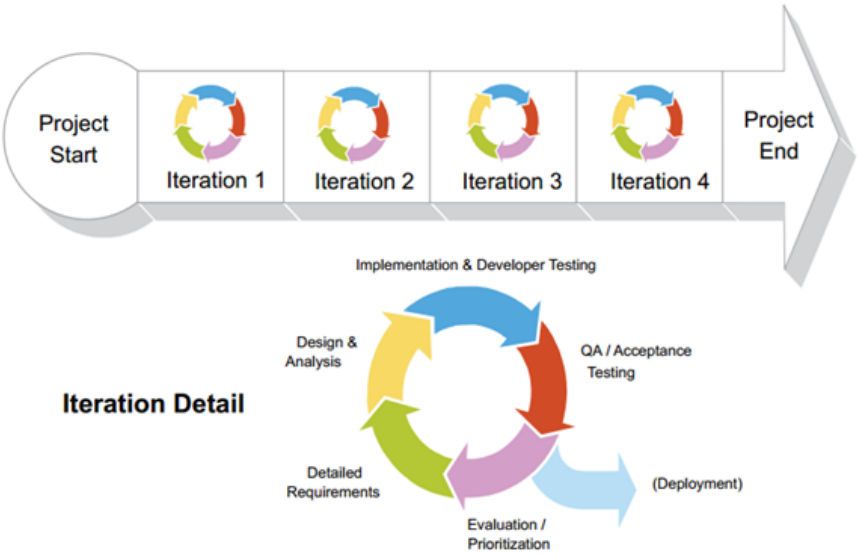


Figura 3.2: Sprint Agile

Scrum si basa sul suddividere il lavoro in piccole unità autocontenute che esplorano tutte le fasi del processo di sviluppo del software, partendo dai requisiti sino al deployment. Rilasciando un prodotto, anche se incompleto e privo di tutte le

funzionalità, si può ottenere un feedback immediato sulla qualità del lavoro svolto, inoltre è possibile eseguire test incrementali sulle funzionalità implementate fino a quel momento.

Tre sono gli "artefatti" fondamentali di cui è composto Scrum:

Product Backlog

Documento che racchiude tutti i task (ordinati in base alla priorità) da svolgere per terminare lo sviluppo del prodotto, questo include i requisiti funzionali e non, le problematiche, i bug trovati, le modifiche del modello e i cambiamenti nei requisiti generati dal feedback col cliente.

Il product backlog è un artefatto in continua evoluzione, al termine di ogni ciclo di lavoro viene infatti aggiornato per allinearsi con le nuove necessità emerse.

Sprint

Singola unità di lavoro nel quale uno o più requisiti vengono eliminati dal product backlog perché portati a compimento. Solitamente si individua una Sprint come alcuni giorni di lavoro ma, in casi estremi, potrebbe essere equivalente anche a una singola giornata.

La Sprint inizia con uno **Stand-up Meeting** nel quale ogni membro del team analizza i task presenti nel product backlog e decide su cosa si dedicherà per quella giornata, segue quindi la sprint effettiva, al termine del quale vi è la **Review**, momento in cui si valuta il completamento o meno dei task assegnati modificando il product backlog di conseguenza.

Sprint Backlog

Equivalente al Product Backlog ma relativo a una singola unità di Sprint.

Il principale problema del modello sta nel non integrare tutte le figure dell'ecosistema di sviluppo all'interno del team Scrum, solo Testers/QAs e Project Owner sono infatti integrati, lasciando Operations e IT all'esterno dell'effettivo team. Il risultato è molto spesso un modello ibrido, solitamente chiamato **Water-Scrum-Fall** e mostrato in figura 3.3, dove gli sprint terminano con la fase di testing, una volta che la qualità funzionale del prodotto è stata assicurata esso viene passato al team di Operations che diventa, inevitabilmente, un collo di bottiglia per il deploy del software sul mercato.

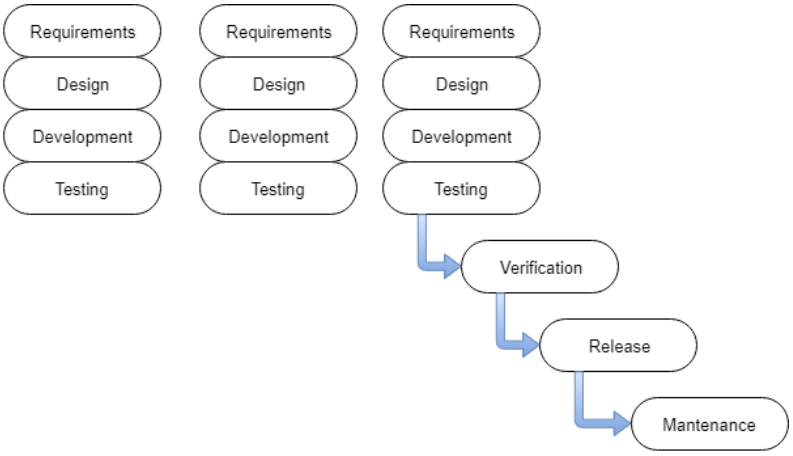


Figura 3.3: Modello Water-Scrum-Fall

3.3 Il Movimento DevOps

In questo esatto scenario si colloca il movimento DevOps.

Il problema messo in luce dalla metodologia Agile è, a livello teorico, facilmente risolvibile; mentre Agile si concentra sull'integrare figure esterne all'interno della fase si Development, è in realtà necessario creare un team multifunzionale che si prenda cura del prodotto software in tutte le fasi del processo di sviluppo, a partire dall'analisi dei requisiti, passando per il rilascio, fino al suo mantenimento al termine della produzione.

Semplicemente creare un team multifunzionale non è però sufficiente, prendere 5/10 persone, provenienti da team e aree funzionali differenti, metterle in una stanza ed aspettarsi che esse collaborino per il "bene superiore" del prodotto, è senza dubbio un'utopia.

Ciò che deve essere radicalmente cambiato è il mindset, la cultura, l'insieme di preconcetti derivati dal modello waterfall che porta le persone a competere piuttosto che a collaborare. Senza un effettivo cambiamento non sarà mai possibile muoversi dal modello odierno, in cui il prodotto è visto come un peso, una responsabilità di cui liberarsi il prima possibile.

E' quindi prima di tutto necessario **Rompere i Silos tra team**, distruggere le barriere, fisiche e mentali, che limitano la condivisione, l'interazione, la collaborazione.

Rispetto al movimento Agile

Volendo confrontare DevOps e Agile potrebbero apparire due pratiche in conflitto, in realtà è esattamente l'opposto. Il DevOps può essere infatti visto come la naturale estensione dell'Agile, mentre Agile ferma il suo lavoro di integrazione allo sviluppo del prodotto, DevOps amplia questa visione, integrando ogni fase del processo, avendo in mente il business generale del sistema e lo stato di lavoro delle persone.

3.3.1 Manifesto

Al giorno d'oggi non vi è un effettivo **Manifesto DevOps**, non è stata ancora definita una linea guida standardizzata che definisce i principi e le metodologie del movimento. Questo è sia fonte di vantaggi che di svantaggi:

Difficile Diffusione Non avere una linea guida condivisa porta inevitabilmente a una molteplicità di modelli differenti. Di conseguenza è estremamente complesso

discriminare quale sia il filone di pensiero più significativo e di valore per il proprio business. L'investimento di tempo iniziale è quindi non trascurabile, aumentando il rischio di abbandonare il progetto accontentandosi modello di lavoro attuale.

Totale apertura alla Discussione D'altra parte, non avere un dogma standardizzato e universalmente accettato, apre le porta alla discussione, all'interazione, al confronto, fondamentali concetti alla base del DevOps. Senza un modello unico e incontestabile non vi saranno limiti all'espressione di pensiero del singolo, portando ogni team a creare il proprio modello, che meglio si adatta alle proprie esigenze, contribuendo quindi ad abbattere le barriere tra le persone. Avendo uno standard, come ha provato Agile con Scrum, prima o poi una discussione verrà conclusa citando il dogma inequivocabile del manifesto, portando a un inutile conflitto e a una limitazione di collaborazione e sviluppo culturale.

3.4 Dev e Ops - Due figure in conflitto

Il movimento DevOps suddivide le figure che prendono parte al processo di sviluppo in due grandi macro-categorie, analizzare le singole responsabilità è utile a comprendere i conflitti che hanno portato al DevOps.

3.4.1 Development (Dev)

I Developers sono tutti coloro responsabili di implementare le features richieste dal cliente garantendo alti standard di qualità. Developers quindi non sono solo gli sviluppatori veri e propri, ma anche il Product Owner, il Team Master che segue il progetto, il team di Test e quello di Quality Assurance.

I Developers hanno il compito di convertire le User Stories del cliente in componenti, funzionanti, del software finale, si concentrano principalmente sui requisiti

funzionali e vengono valutati in base al numero di features, testate e funzionanti, integrate nel prodotto.

3.4.2 Operations (Ops)

Gli Operations, per contro, si concentrano principalmente sui requisiti non funzionali per progetto, la macro-categoria include tutti coloro che si occupano di gestire deploy e release, configurare architettura e infrastruttura di rete, garantire la sicurezza, monitorare il sistema dopo il rilascio ed assicurarsi, in generale, che i clienti siano soddisfatti del prodotto sviluppato.

Il lavoro degli Ops è quindi focalizzato sul garantire sicurezza, stabilità, modularità, facile manutenzione ed strumentazione, il loro lavoro è valutato in base alla stabilità del sistema e al grado di soddisfazione del cliente.

3.4.3 Il conflitto

Gli obiettivi di Dev e Ops sono decisamente differenti, vengono valutati secondo diverse metriche e fanno riferimento a diversi responsabili, appare quindi inevitabile che nasca un conflitto tra le due categorie.

Debito Tecnico

Volendo generalizzare l'ideologia media del Developer, esso nel suo lavoro sarà focalizzato a fornire il maggior numero di features nel minor tempo possibile, spesso (fortunatamente non sempre) tralasciando alcuni requisiti non funzionali che reputa secondari o integrabili in un secondo tempo.

Si dice **Debito Tecnico** il costo, economico, temporale e fisico, imposto ad un team o organizzazione a causa delle azioni svolte da un altro team o organizzazione.

Essendo entrambi i team responsabili dello sviluppo del prodotto finale, se uno dei due non svolge il proprio lavoro adeguatamente questa mancanza si ripercuote inevitabilmente sull'altro team. Di conseguenza, è necessario che ciascuno svolga il proprio lavoro "a regola d'arte" rispettando gli standard di qualità e velocità imposti dal mercato, ma soprattutto che ogni team assuma come ottimale il lavoro degli altri. Senza fiducia non è possibile parlare di collaborazione.

3.5 DevOps in pratica

Dalle considerazioni fatte appare chiaro che, per raggiungere un livello di maturità sufficiente, sia necessario modificare radicalmente la mentalità delle persone. Serve un cambiamento, ma esso non può essere brusco ed immediato, è necessario un passaggio graduale in modo da consolidare l'ideologia e il metodo di pensare delle persone coinvolte.

3.5.1 Lewin's 3-Stage Model

Lewin [12] presenta un modello utile a guidare il cambiamento all'interno dei team di sviluppo, basandosi sulla metafora del "modellare il ghiaccio": supponendo di aver un cubo di ghiaccio, e di volerlo rendere un cono, qualcosa di radicalmente differente, il metodo più semplice è ovviamente scongelare e ricongelare nella nuova forma.

Tre sono le fasi identificate per apportare un cambiamento significativo e duraturo:

Unfreeze

I cambiamenti "imposti dall'alto" non sono utili, per fare in modo che il cambiamento sia assimilato a dovere è necessario che la motrice siano le stesse persone da cambiare. E' quindi necessario prima di ogni altra cosa identificare i problemi attuali

Figura 3.4: Lewin's 3-Stage Model

e metterli in luce, smuovere gli individui dalla loro "area di comfort", fare in modo che comprendano che non è possibile proseguire secondo la strada attuale.

Questo stadio è senza dubbio il più complesso da implementare, quando si impone alle persone di auto criticarsi, inevitabilmente si sbilancia l'equilibrio, solitamente precario, tra di esse. Per questo è necessario che ciascuno comprenda a pieno il motivo per cui si sta introducendo una crisi, per quanto controllata, all'interno del team.

In questo modo saranno gli individui stessi, una volta realizzati i problemi interni, a desiderare il cambiamento e a vederlo come l'unica strada possibile da percorrere.

Change

Una volta che le persone hanno realizzato e abbracciato la necessità del cambiamento, è necessario definire le linee guida per il futuro del team. Vi sarà prima di tutto una fase esplorativa atta a trovare la miglior soluzione per colmare i problemi messi in luce nella fase precedente.

Questo cambiamento non avviene ovviamente da un giorno all'altro, serve tempo per trovare il metodo più adatto alla determinata casistica, comprenderlo e iniettarlo nel proprio modello di lavoro. Fondamentale è la comunicazione tra le persone,

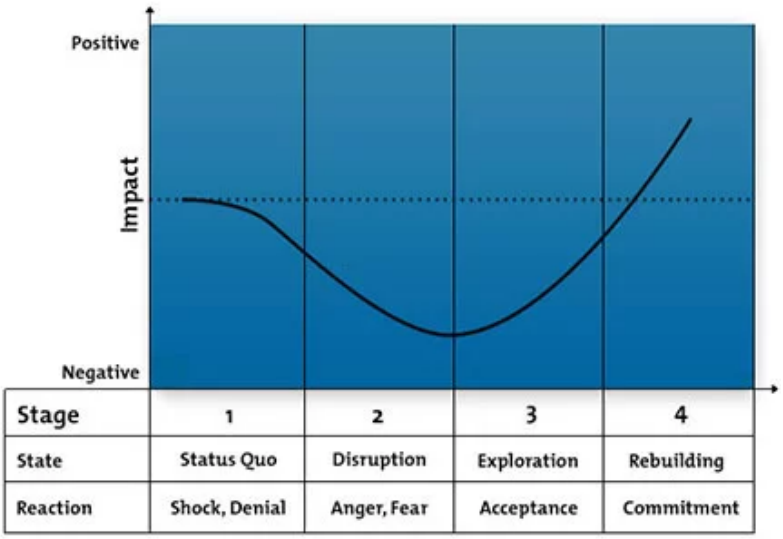


Figura 3.5: Curva di cambiamento

non devono infatti nascere ulteriori problemi, oltre a quelli già presenti, a causa di incomprensioni.

In questa fase vi saranno inevitabilmente individui contrari al cambiamento, solitamente coloro che traggono notevole vantaggio dalla loro posizione, è necessario gestire il prima possibile queste casistiche facendo loro comprendere quanto il vantaggio a lungo termine sia più significativo di quello momentaneo.

Per quanto riguarda il caso specifico del DevOps, nella prossima sezione sono presentate 7 Best Practices utili a guidare il cambiamento.

Refreeze

Terminata la fase di esplorazione ed implementazione, quando le persone hanno iniziato ad utilizzare le nuove metodologie in modo consolidato e continuativo, è bene fermare il cambiamento e consolidare quanto fatto.

Deve essere garantito che ognuno si sia lasciato alle spalle i vecchi modelli di lavoro e utilizzi in modo stabile e consapevole le nuove metodologie. Senza una fase

di consolidamento si rischia di ricadere nel pensiero del *"Cambiamento fine a se stesso"*, vanificando tutti i progressi fatti nelle fasi precedenti.

Creare un senso di stabilità è infine fondamentale per fare in modo che ognuno ricrei la propria "area di comfort" distrutta durante il processo. Per questo è altamente sconsigliato introdurre troppe novità in una sola volta, il cambiamento è un processo iterativo e incrementale che porta a migliorare se stessi e gli altri, giorno dopo giorno.

3.5.2 Il Cambiamento nel DevOps

Il cambiamento introdotto dal DevOps può essere generalizzato e riassunto in una semplice frase, adottata da November Fire [18] all'inizio del 2017.

You build it, You run it"

Il vero cambiamento necessario è far capire alle persone che ciascuno è responsabile della salute di tutto il prodotto software all'interno di ogni fase del ciclo di vita. Non si parla soltanto della fase di sviluppo, sono incluse anche la definizione e implementazione dell'infrastruttura, l'installazione, il testing, il releasing, il deployment, la configurazione, il monitoring e il bug fixing successivo al rilascio.

I Developers non sono più solo responsabili delle features da loro sviluppate, ma anche del loro funzionamento nell'ecosistema in cui verranno collocate in post-produzione, incluse sicurezza, performances e la soddisfazione del cliente.

La mentalità del *"Run what you build"* porta il Developer a sviluppare una maggiore consapevolezza delle sue azioni, dei suoi errori e del debito tecnico che, per negligenza o inconsapevolezza, potrebbe introdurre nel software. La mentalità del "*Non è un mio problema*" non trova più alcun terreno fertile dal momento che, in

caso di problematiche, è esattamente chi ha partecipato allo sviluppo a venire chiamato in causa e, non si parla del solo responsabile diretto, ma di tutto il team al completo.

Allo stesso modo gli Operations divengono responsabili non soltanto di stabilità e sicurezza in post-produzione, si devono anche prendere cura del software nella sua fase di sviluppo, ad esempio assicurandosi che l'infrastruttura e architettura utilizzata dai Dev per realizzare le features sia il più possibile simile, idealmente identica, al sistema finale di produzione.

Trasparenza e Comunicazione diventano parole chiave per una sana collaborazione all'interno di questo nuovo nato Team Multifunzionale "espanso". E' infatti necessario un continuo scambio di idee, metodologie, tecniche, paradigmi tra Dev e Ops in modo che entrambe le parti acquisiscano le capacità e la consapevolezza per prendersi cura del progetto. Se il processo di cambiamento è stato eseguito in maniera efficace, saranno le persone stesse a portarlo avanti, inserendo ciò che meglio sanno fare per migliorare continuamente e incrementalmente il lavoro proprio e dell'intero team.

Il DevOps definisce questo cambiamento con il termine **Continuous Learning**, una metodologia che consiste nello sperimentare continuamente nuove opportunità, linguaggi, metodi, tecnologie e condividerle con l'intero team ibrido in modo da crescere costantemente ed aumentare la coesione tra le persone.

3.5.3 Risultato finale

Tralasciando i vantaggi in termini di integrazione e collaborazione intra-team, lo scopo principale del DevOps resta quello di riuscire a rilasciare sul mercato un prodotto ricco di features di qualità e in continuo aggiornamento, in modo da soddisfare la richiesta degli utilizzatori. L'obiettivo finale di un'azienda che intraprende la strada del DevOps è il **Continuos Deployment**, la condizione per cui il rilascio di nuove

features, testate e funzionanti, avviene con frequenza settimanale se non giornaliera, in maniera trasparente, controllata ed automatizzata.

Il Continuous Deployment, unito a una cultura orientata alla coesione e alla collaborazione, permette di instaurare uno stream continuo di features consegnate direttamente all'utilizzatore in modo che possa dare feedback real time sulla qualità del prodotto realizzato. Il valore del software, prima generato soltanto al momento del rilascio, si converte in un valore potenziale controllabile da pure decisioni di business.

3.6 Guidare il cambiamento - 7 Best Practices

Il cambiamento necessario per abbracciare il DevOps è decisamente significativo, per rendere il tutto più semplice, lineare e guidato sono stati definiti numerosi modelli che descrivono tale processo.
A causa della mancanza di un manifesto, come già discusso, è complesso comprendere quale sia il modello più valido, molti identificano il DevOps come sola cultura, altri lo definiscono come un insieme di tool atti a velocizzare il time-to-market.

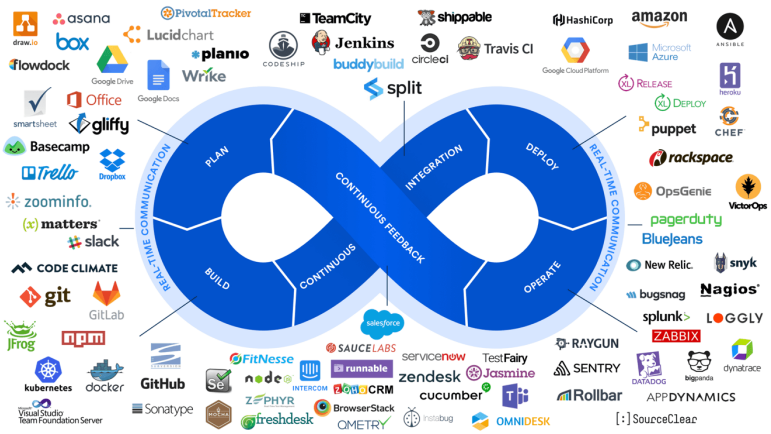


Figura 3.6: DevOps Infinity Loop Model

In questo elaborato ho seguito il modello presentato da Gene Kim, Kevin Beher e George Spafford in *"The Phoenix Project"* [15], in cui sono presentate le ideologie del movimento DevOps in forma narrativa, mettendo in luce problematiche iniziali e vantaggi successivi nella diretta esperienza di lavoro. Il modello è stato successivamente sviscerato e incrementato da Bill Ott, Jimmy Pham e Haluk Saker in "Enterprise DevOps Playbook" una serie di pillole pubblicate da O'Reilly.

Dopo aver studiato ed implementato le Best Practices proposte, sono convinto che una vera assunzione del DevOps nasca da una perfetta sinergia tra cambiamento culturale e tool di supporto. I Tools non sono altro che il mezzo da utilizzare nella fase di Change per assimilare adeguatamente i concetti del DevOps.

Il modello è composto da 7 pratiche che mirano a unificare e standardizzare strumenti e metodi di lavoro di Dev e Ops:

- **Configuration Mangement (CM)**: metodo che permette di condensare conoscenze ed esperienza altamente specifiche riguardo il Provisioning, in file di configurazione con le stesse caratteristiche di un prodotto software;

- **Infrastructure As a Code (IaC)**: allo stesso modo, anche la generazione dell'infrastruttura di rete può essere convertita in codice, combinando CM e IaC è possibile automatizzare il processo di creazione e configurazione di nuove istanze, abbattendo i tempi di attesa e velocizzando lo sviluppo;

- **Continuous Integration (CI)**: il processo di integrazione di features differenti, sviluppate da diversi individui, può essere molto complesso. Eseguendo l'operazione ad alta frequenza su porzioni minori di codice è possibile ridurre notevolmente lo sforzo di integrazione;

- **Continuous Testing (CT)**: nell'ottica di uno stream continuo di rilasci i test, continui ed automatizzati, diventano l'unica forma per garantire Qualità

e Performances del prodotto sviluppato, fornendo inoltre un valido indicatore sullo stato di sviluppo e rischio legato al rilascio;

- **Continuous Delivery (CD)**: metodo che permette di incrementare il valore del prodotto software, spostandolo dal semplice valore di rilascio finale, alla possibilità di effettuare scelte di business riguardo il come e quando rilasciare una determinata feature;

- **Continuous Deployment (CD)**: estensione del concetto di Continuous Integration a tutto il processo di sviluppo e rilascio del sofware. Si viene a creare uno stream real time di features continuamente integrate e consegnate al cliente in maniera incrementale.

- **Continuous Monitoring (CM)**: il prodotto software viene monitorato in ogni fase del ciclo di vita, sia durante lo sviluppo che dopo il rilascio, si crea quindi un feedback loop continuo che permette di migliorare il lavoro del team.

3.7 Configuration Management (CM)

Il primo passo per unificare la cultura di due gruppi così divergenti quali Dev e Ops è lo standardizzare l'ecosistema dove le due entità sussistono e lavorano. Questa mutazione include prima di ogni cosa l'ambiente di lavoro fisico, è impensabile infatti collaborare se i membri del team si trovano fisicamente collocati in stanze differenti, o anche semplicemente in un openspace con divisori.

Molto più significativa è la mutazione da introdurre in ambito di infrastruttura di lavoro, ad esempio:

Il team è composto da dieci persone, ciascuno con un notebook su cui lavorare. Cinque di essi montano Windows 10, tre hanno una Macbook Pro con macOS High

Sierra e gli ultimi due si affidano a Ubuntu 16.04. Al team viene quindi chiesto di sviluppare un sistema di previsione delle vendite, il cui deploy sarà fatto su un cluster composto da macchine Centos 7 e gestito da un team di sistemisti, dislocato in un'altro piano dell'azienda.

Il caso è estremo ma non lontano dalla realtà. Le persone coinvolte nel progetto, non solo si trovano fisicamente in luoghi differenti, ma lavorano con architetture e ambienti diversi, dipendenti dalle abitudini, positive e negative, del caso.

I potenziali problemi sono molteplici, a meno di adottare particolari accorgimenti, è altamente probabile che il codice scritto e compilato da uno sviluppatore non funzioni, o abbia comunque problemi di dipendenze e compatibilità nell'architettura di un collega, o ancora peggio nel cluster di produzione.

3.7.1 Deprecare gli script di configurazione

E' fondamentale quindi standardizzare l'ambiente di lavoro, introducendo nel team i sistemisti che mantengono il cluster. L'ideale sarebbe che proprio essi, esperti di Provisioning, si occupino di modellare, standardizzare ed implementare l'ambiente di lavoro comune per il team, in modo che rispecchi il più possibile le caratteristiche dell'ambiente di produzione.

La pratica usuale è quella di utilizzare uno o più script, scritti solitamente in linguaggio bash o similari, per portare istanze vergini allo stato desiderato. I problemi di tali script sono molteplici:

Difficoltà di comprensione a meno di possedere conoscenze specifiche del linguaggio e dei metodi di configurazione, non è sempre immediato comprendere il flusso di lavoro.

Difficoltà di condivisione e diffusione dal momento che pochi riescono a comprendere facilmente tali script, non tutti possono utilizzarli in maniera sicura ed affidabile, generando la necessità di fare affidamento sempre e comunque su una o due persone.

Dipendenza dalle abitudini del singolo standardizzare uno script è decisamente complesso, certo esistono pattern di modellazione universalmente condivisi, ma la struttura del singolo script resta fortemente dipendente da abitudini e competenze di chi l'ha sviluppato.

Complessa automazione della configurazione sarebbe comodo poter automatizzare la configurazione di un'istanza, dotandola delle sole capacità necessarie per una determinata applicazione o uno scopo molto specifico.

Configuration as a Code

Da qui nasce il vantaggio e la necessità di adottare un sistema di **Operating System Configuration Management (OSCM)**. Questo tipo di tool permette di definire le caratteristiche di un'istanza tramite un **Domain Specific Language (DSL)** machine-parsable e human-readable, facilmente comprensibile e condivisibile.

Provisioning Per Provisioning si intente il processo mediante il quale sia assegna ad un'istanza la configurazione e l'insieme di servizi che al caratterizzano all'interno della rete.

Forzare gli Operations ad utilizzare un linguaggio di programmazione più strutturato apre alla possibilità, tanto semplice quanto potente, di trattare la configurazione esattamente come se fosse un software, con gli stessi tool e tecniche utilizzati dai Developers nel loro lavoro quotidiano.

Questo significa che la configurazione può essere testata, versionata, documentata e condivisa in maniera facile ed efficace. Inoltre si elimina la barriera di eterogeneità tra i tools usati dai Dev e dagli Ops, facendo un passo avanti verso l'integrazione.

3.7.2 Idempotenza e Convergenza

Oltre all'immediato vantaggio apportato dai DSL, un tool di OSCM per essere tale deve garantire due caratteristiche fondamentali:

Idempotenza

"L'esecuzione multipla della stessa operazione, dopo la prima applicazione, non ha conseguenze sul sistema destinatario"

Questo significa che è possibile eseguire la stessa istruzione un numero infinito di volte senza temere che questa possa portare il sistema in uno stato indesiderato.

Convergenza

"Un comando viene eseguito se e soltanto se è strettamente necessario"

Di conseguenza, il tool di OSCM eseguirà i comandi di configurazione, in maniera incrementale, se e soltanto se lo stato del sistema destinatario è differente da quello specificato nella configurazione.

Per garantire tali proprietà, i tools di OSCM si avvalgono di una comunicazione bidirezionale utile alla comprensione dello stato del sistema target.

3.7.3 Modelli di Configuration Management

Il mercato attuale offre tre metodologie differenti (e relativi tools) per effettuare Configuration Management: Modello Imperativo, Modello Dichiarativo, Modello Ibrido.

CM Imperativo

Il CM Imperativo è molto simile al tradizionale concetto di Configuration Scripting, la configurazione è infatti definita sotto forma di sequenza di istruzioni da eseguire per raggiungere lo stato finale desiderato. Il vantaggio sta nella garanzia di Idempotenza e Convergenza.

Un esempio di CM Imperativo è **Chef** [4], con questo tool è possibile definire file di configurazioni, detti Recipes che indicano la sequenza di comandi da eseguire. Le recipes possono essere raggruppate in un Coockbook, pensato per contenere tutte le configurazioni riguardanti uno stesso progetto o caso d'uso. Chef opera in modalità Master-Slave, nell'istanza da configurare deve essere preventivamente installato un Client che si occupa di interagire con il Master ed eseguire comandi in sua vece.

Il modello imperativo permette di avere il massimo controllo sulla configurazione applicata ad una istanza, richiedendo però la conoscenza dei singoli step per raggiungere lo stato desiderato.

CM Dichiarativo

Il CM Dichiarativo colma esattamente questo problema, distaccandosi dal tradizionale metodo di Configuration Scripting. E' infatti molto più comune avere una specifica del tipo: *"Necessito di un'istanza con SBT in modo da generare JAR eseguibili dal nostro codice Scala"*. Non vi è alcuna definizione della versione richiesta o delle caratteristiche del sistema host.

Un tool di CM Dichiarativo permette di definire configurazioni sotto forma di descrizione dello stato finale desiderato, il metodo per raggiungere (e mantenere) tale stato è lasciato al tool.

Puppet [20] è un tool di CM Dichiarativo, permette di definire file di configurazione, in un linguaggio proprietario, che descrivono lo stato dell'istanza desiderato. Anche Puppet necessita di un Client installato sull'istanza da configurare.

Il modello dichiarativo permette di ottenere il massimo livello di astrazione possibile, non vi è più la necessità di conoscere il processo per ottenere un risultato, ma d'altra parte si perde il controllo totale di ciò che viene effettivamente eseguito.

CM Ibrido

Il Configuration Management Ibrido unisce i due modelli precedenti, cercando di estrapolarne i vantaggi. Per questo elaborato ho deciso di utilizzare **Ansible**, un tool di Configuration Management che unisce la possibilità di descrivere lo stato desiderato mantenendo il controllo del flusso di esecuzione.

3.7.4 Metodi di gestione della configurazione

Configuration Management Dinamico

Il Configuration Management Dinamico, anche detto **Fryed Model**, consiste nel partire da una istanza vergine ed applicare ad essa la configurazione desiderata tramite il tool scelto. Lo stato dell'istanza viene quindi mantenuto eseguendo periodicamente il tool che, grazie alle caratteristiche di Idempotenza e Convergenza, applicherà le sole modifiche necessarie a mantenere lo stato desiderato.

Questo modello è indubbiamente semplice ed applicabile senza necessità di particolari architetture, ha però lo svantaggio del essere sempre incrementale. Una volta configurata l'istanza non è infatti possibile risalire in maniera "safe" al suo stato

originale. Allo stesso modo, nel caso in cui una configurazione venga riapplicata a causa di un disallineamento successivo alla prima esecuzione, il tool da la sola garanzia che lo stato sia quello desiderato, non può però assicurare che la causa del disallineamento sia stata risolta.

Questo tipo di modello è adatto per quei sistemi composti da istanze stabili, in cui è improbabile che avvenga un cambiamento tale da compromettere lo stato dell'istanza nonostante il tool di CM.

Immutable Infrastructure

Il modello a Immutable Infrastructure, anche detto **Baked Model** consiste di base nella stessa sequenza di istruzioni del modello dinamico, un'istanza vergine viene configurata tramite il tool scelto per ottenere lo stato desiderato; si esegue quindi uno **Snapshot** dello stato dell'istanza e lo si utilizza come modello per le successive.

Se una delle istanza si disallinea rispetto allo stato desiderato, piuttosto che riapplicare la configurazione, si preferisce eliminarla e crearne una nuova partendo dallo snapshot memorizzato. In questo modo si ha la certezza che la nuova istanza non presenti errori di configurazione nascosti ma sia esattamente come la si è progettata.

Questo modello è diventato molto popolare con la diffusione delle piattaforme di distribuzione Container-based, come ad esempio Docker. Risulta infatti molto comodo definire una *Docker Image* che rappresenta lo stato desiderato dell'istanza e sostituirla alle macchine compromesse in caso di necessità.

Questo approccio però mostra il suo vero potenziale solo nel caso in cui si ha un gran numero di istanze, molto simili tra di loro (come può essere in un cluster composto da Data Node tutti uguali).

3.7.5 Ansible - Configuration Management Ibrido

Ansible è un Configuration Management tool che permette di unire il controllo dell'esecuzione dato dai CM Imperativi alla capacità di modellazione ed astrazione dello stato tipico dei CM Dichiarativi.

Playbook

Il file di configurazione di Ansible è detto **Playbook**, un componente auto-inclusivo che racchiude sia la configurazione sia il target da configurare.

Il Playbook descrive la configurazione secondo il modello Dichiarativo utilizzando il linguaggio YAML, questo linguaggio presenta un livello di astrazione e capacità descrittiva molto elevato, rendendolo una scelta pratica ed efficace anche in vista della diffusione e condivisione del componente.

In Appendice A, Listato A.1 è mostrato un esempio di Playbook che installa SBT (Scala Build Tool) e tutte le dipendenze necessarie, su una istanza etichettata come worker, agendo come utente root. Come si può notare, sebbene la definizione dei componenti da installare sia dichiarata sulla base dello stato desiderato, si ha comunque un notevole controllo del flusso di configurazione.

Role

La potenza espressiva di Ansible e YAML si manifesta ulteriormente aumentando il livello di astrazione della configurazione, è infatti possibile racchiudere la sequenza di task di un Playbook, in un modulo autocontenuto detto **Role**.

Ansible, tramite il suo componente ansible-galaxy, definisce in maniera automatica il ruolo, creando la struttura di directory e dipendenze necessaria. Il file che definisce i **Tasks** da eseguire viene arricchito con variabili, file, test, documentazione, e tutto il necessario per garantire che il ruolo ottenuto sia standalone. E' quindi

possibile estrapolare la sequenza di task e racchiuderli in un ruolo, il risultato finale sarà un Playbook estremamente dichiarativo e di facile comprensione (in Appendice A, Listato A.2 è mostrato un esempio di ruolo).

Supponendo di definire racchiudere in un insieme di Ruoli le varie responsabilità del sistema, il Provisioning di un'istanza diventa nient'altro che l'elencare le feature richieste per la determinata applicazione. Il Playbook e il ruolo risultano inoltre auto-documentati, per comprendere cosa viene installato è sufficiente una lettura della configurazione.

Inventory

Come si può notare dai listati, anche la definizione del target da configurare è dichiarativa, nel Playbook è infatti presente solo l'etichetta "worker". Ansible gestisce i target in un file separato detto **Inventory** nel quale è possibile assegnare delle etichette a istanze o gruppi di esse e utilizzare espressioni regolari per definire insiemi di istanze a ID incrementale.

Esecuzione Agent-less

L'esecuzione di Ansible è nominalmente Agent-less, questo significa che è possibile configurare un'istanza senza la necessità di installare preventivamente software client e relative dipendenze (come invece avviene per Chef e Puppet).

Questa definizione è vera però solo a livello teorico, in pratica Ansible è un modulo implementato in Python, di conseguenza richiede che sul target sia presente una versione sufficientemente recente del software. Inoltre l'esecuzione è fatta tramite protocollo OpenSSH, è quindi necessario un client SSH attivo e in ascolto.

Al giorno d'oggi, queste due features sono solitamente incluse nelle più comuni distribuzioni, di conseguenza, per questi casi, Ansible risulta effettivamente una soluzione Agent-less.

Modularità ed Estensibilità

Ansible è un modulo Python, il che lo rende facilmente installabile tramite `pip` o altri gestori di pacchetti. Nativamente il tool supporta più di 1200 moduli, come *yum* e *get_url* utilizzati nell'esempio in Appendice, che permettono di ottenere le funzionalità più disparate. La comunità di supporto, chiamata **Ansible-Galaxy**, offre infine una vasta gamma di ruoli Open Source ed utilizzabili out-of-the-box.

3.8 Infrastructure As a Code (IaC)

L'**Infastructure as a Code** è la diretta conseguenza ed espansione del Configuration Management nell'ottica di automatizzare e standardizzare l'ambiente di lavoro di Dev e Ops. Supponendo di aver assimilato con successo il CM, il secondo problema nella standardizzazione dell'ecosistema sta nella necessità di creare istanze virtuali "on-demand". Nel tradizionale modello di team, sono ancora una volta i sistemisti ad occuparsene, ricadendo nei problemi suddetti riguardo la complessità di condivisione.

Secondo il principio del IaC, anche l'infrastruttura di rete può essere trattata come un qualsiasi altro software, definita quindi con un linguaggio di alto livello, testata, integrata, documentata e condivisa con gli stessi tools utilizzati dai Developers.

Inoltre, come ogni altro software ben strutturato, l'IaC permette di garantire la **Replicabilità**, ogni esecuzione della configurazione porterà alla creazione della stessa infrastruttura, indipendentemente dall'utilizzatore o dal numero di esecuzioni. Questo significa, a livello di business, che il tempo di attesa necessario al Developer per ottenere una nuova istanza è drasticamente ridotto, gli Operations hanno infatti fornito a tutto il team la possibilità di creare le proprie istanze in maniera autonoma, sicura e controllata.

3.8.1 Terraform

Terraform è un tool che permette di gestire in maniera automatica e descrittiva l'**Orchestration** e il **Provisioning** di un architettura di rete, come un cluster. Il tool garantisce la compatibilità con i più comuni Provider cloud come AWS e Google Cloud Platform, sfruttando le API messe a disposizione dal servizio è in grado di configurare l'infrastruttura in maniera agent-less e idempotente.

Configurazione Descrittiva Terraform sfrutta il **HashiCorp Configuration Language (HCL)** per racchiudere esperienza e conoscenza di un singolo individuo in un componente di facile comprensione, condivisibile, portabile e autodocumentato.

Idempotenza e Convergenza Idempotenza e Convergenza sono ancora una volta garantite, inoltre Terraform, a differenza di Ansible, è un tool completamente dichiarativo, mantiene quindi l'Idempotenza sia rispetto lo stato corrente dell'infrastruttura, sia rispetto le azioni precedentemente applicate dallo stesso file di configurazione. In Appendice A (Listato A.3) è mostrato un esempio di configurazione per un'istanza Compute Engine su Google Cloud Platform.

Una modifica successiva alla variabile `server_count`, definita esternamente alla configurazione, permetterà di istanziare il numero esatto di Compute Engine in base ala necessità del caso. Terraform, valutando lo stato corrente del sistema e lo storico delle sue esecuzioni, agirà di conseguenza aggiungendo o eliminando istanze. Lo storico delle esecuzioni è salvato in un file, questo significa che è possibile mantenere un repository condiviso garantendo la trasparenza e l'allineamento all'interno del team.

Lo stesso comportamento può essere ottenuto anche con i tool di Configuration Management, trattandosi però di tool pensati per il Provisioning, offrono le

funzionalità con un approccio meno intuitivo.

Provisioning Terraform mette a disposizioni numerosi metodi per automatizzare questo processo, supponendo però di aver già assimilato tool e mindset del Configuration Management, appare naturale unire i due principi.

La fusione di Configuration Management e Infrastructure As a Code permette di automatizzare completamente la generazione, configurazione, e mantenimento del cluster utilizzato dal team, in qualsiasi ambiente del ciclo di vita del prodotto software.

Se inizialmente, per ottenere un nuovo server, erano necessarie giornate intere, con conseguente ritardo e malcontento, industrializzando il processo è possibile ottenere lo stesso risultato in tempi dell'ordine delle ore, lasciando maggior tempo allo sviluppo di features e qualità.

3.9 Continuous Integration (CI)

Per Continuous Integration si intende il processo per cui il codice relativo a differenti features dello stesso prodotto software, sviluppato da individui diversi del team, viene integrato continuamente giorno per giorno in un repository comune.

In passato, seguendo il Waterfall model, il codice di differenti sviluppatori veniva integrato tutto insieme in una giornata dedicata, il "Release Day". Integrare in una volta sola il lavoro fatto per settimane o addirittura mesi portava a molteplici problematiche quali disallineamenti di versione, scelte progettuali contrastanti, debito tecnico accumulato per mesi, necessità di modificare codice scritto da molto tempo,... . Problemi solitamente derivati da negligenza e mancanza di comunicazione. Il risultato era un processo di bug fixing iterativo e incrementale fatto di fretta, senza tenere in considerazione la qualità del prodotto finale, con conseguente perdita di valore del software.

La metodologia Agile mitiga questo problema introducendo Sprint di implementazione, integrazione e rilascio, atti ad ottenere feedback diretto dal cliente. Il problema dell'integrazione, per quanto meno evidente, resta comunque da tenere in considerazione: dovendo presentare versioni incrementali è inevitabile introdurre nel prodotto delle componenti simulate che, se non pensate saggiamente, costituiscono una criticità slittata nel tempo.

Il DevOps prende il concetto di Continuous Integration (CI) dalla metodologia Agile, nello specifico dal'**eXtreme Programming (XP)**:

"If something hurts, do it more often and bring the pain forward"

Apparentemente potrebbe sembrare una frase contrastante, in realtà mette in luce un concetto tanto semplice quanto efficace: piuttosto che attendere una giornata unica in cui unificare tutto il lavoro fatto dalle varie parti del team, è più conveniente integrare il lavoro giorno per giorno, mantenendo quindi un repository unico e condiviso con l'ultima versione costantemente aggiornata del sistema.

Il movimento DevOps spinge i Developers ad integrare il proprio lavoro quanto meno una volta al giorno, la frequenza di integrazione aumenta all'aumentare del livello di confidenza e comprensione delle metodologie.

Integrare costantemente il codice fatto da persone differenti resta comunque un'operazione critica, servono quindi una serie di accorgimenti per rendere il processo rapido, efficace ma sopratutto automatico.

3.9.1 Version Control System (VCS)

Al giorno d'oggi qualsiasi azienda dispone di un Version Control System (VCS), che esso sia basato su Git, Mercury o similari, non tutte però ne sfruttano il vero potenziale. Un VCS permette al team di gestire e mantenere il codice sviluppato all'interno di un repository condiviso ed accessibile in modo trasparente ad ogni

membro. Il VCS permette inoltre di gestire le varie versioni del codice integrato nel repository, visionarle singolarmente e, se necessario, eseguire rollback per tornare ad una precedente versione stabile del sistema.

Il VCS permette quindi di definire un flusso temporale di sviluppo, identificato da snapshot integrati nel sistema, e composto da un'unica sequenza di nodi.

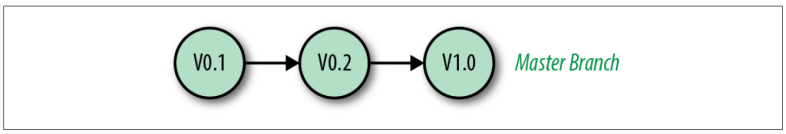


Figura 3.7: Centralized Flow

L'approccio centralizzato, contenente un solo ramo di sviluppo e produzione, è adatto solo per piccoli progetti gestiti da un unico sviluppatore. Non appena il progetto si amplia e più sviluppatori lavorano in parallelo a diverse features, diviene necessaria una migliore architettura.

Feature Branches

Come già indicato dal movimento Agile, in particolare Scrum con il Product Backlog, è funzionale suddividere il lavoro in piccole unità il più possibile autocontenute e non interdipendenti. Individuando Features o quanto meno gruppi di Features, basandosi sui requisiti del cliente, è possibile parallelizzare lo sviluppo tra i membri a disposizione del team. Maggiore è la granularità delle features, più breve sarà il tempo di lavoro e maggiore la velocità di rilascio e feedback del cliente.

- Il **Master Branch** mantiene la versione centralizzata ed aggiornata del progetto, questa deve essere modificata solo se strettamente necessario perché rappresenta anche la versione base compilata ed usata per il rilascio del prodotto;

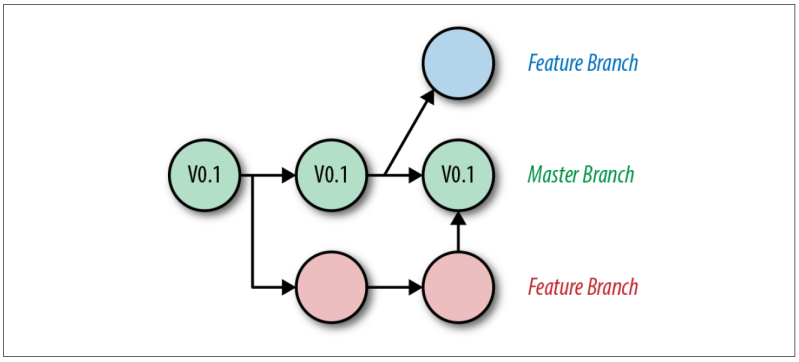


Figura 3.8: Feature Flow

- Per ogni nuova feature i Developers creano **Feature Branches**, snapshot del Master che permettono di sviluppare nuove features senza il timore di corrompere la versione principale.

Questo schema può essere complicato a piacimento introducendo rami di development, testing, releasing, hotfix, in base alle necessità dello specifico progetto.

Secondo la Continuous Integration ogni sviluppatore, non appena ha completato e accuratamente testato una feature, deve integrare il suo Feature Branch con il Master Branch. Il VCS stesso si occupa di mettere in luce eventuali problematiche di integrazione, permettendo di risolvere immediatamente bug ed incompatibilità che altrimenti sarebbero slittate avanti fino al Release Day o, ancora peggio, in produzione.

3.9.2 Test automatizzati

La decisione di integrare il proprio Feature Branch non è arbitraria, per assicurare la totale assenza di criticità nel Master Branch è necessario che ogni feature, prima della sua integrazione, sia testata a fondo. Seguendo sempre i principi del eXtreme Programming:

- La batteria di test deve coprire tutti gli aspetti funzionali e non funzionali riguardo la feature;
- Deve inoltre essere eseguibile in tempi brevi, idealmente in meno di 10 minuti. Tempi maggiori comporterebbero un'eccessiva attesa per validare il proprio lavoro, disincentivando la frequente integrazione;
- I test e relativi messaggi di errore devono essere semanticamente significativi in modo da portare alla correzione in tempi brevi, ancora una volta il tempo ideale è 10 minuti.

La batteria di Test deve inoltre coprire ogni aspetto necessario a garantire una sicura integrazione, questo include **Unit Tests** per verificare che i requisiti funzionali della nuova features siano rispettati, **Smoke Tests** atti a verificare i requisiti di contesto, **Integration Tests** che verificano il corretto funzionamento dell'intero sistema, supponendo che la nuova feature sia stata integrata con successo.

Test Driven Development (TDD)

Il Test Driven Development (TDD) è una tecnica anch'essa derivante del eXtreme Programming, secondo la quale lo sviluppo di una features si struttura in tre fasi ripetute iterativamente:

- Stesura di Test automatizzati per la determinata features, tali test falliranno non essendo implementati;
- Scrittura del codice che permette di passare il test secondo i requisiti del cliente
- Ottimizzazione del codice scritto

I Test, pur mantenendo la funzione di controllo del funzionamento, assumono un significato totalmente innovativo all'interno del processo di sviluppo:

- Sono la naturale conversione dei requisiti del cliente in elementi modulari, human-readable e machine-parsable;

- Permettono di rappresentare la semantica di ogni componente del sistema prima ancora che esso sia stato implementato;

- Rappresentano una metrica significativa riguardo la progressione di sviluppo del prodotto finale (**Copertura**).

3.9.3 Pipeline di Continuous Integration

L'unione dei principi suddetti porta alla necessità di automatizzare il processo di test e integrazione, si viene quindi a creare una Pipeline composta da 5 fasi:

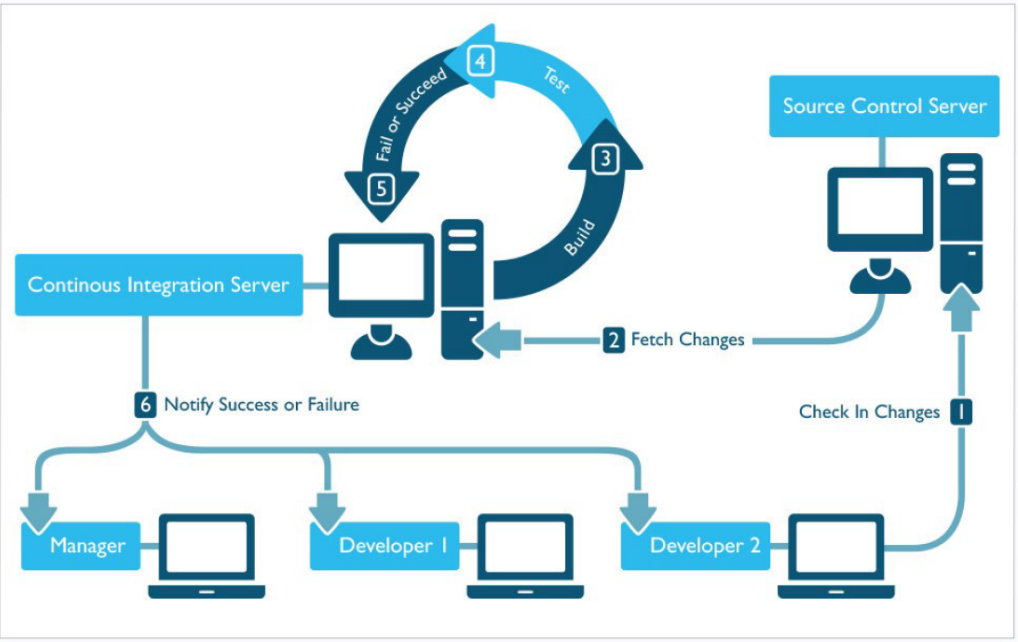


Figura 3.9: Continuous Integration Pipeline

1. **Unit Tests Locali:** Il Developer implementa la feature ed esegue gli Unit Test, definiti secondo il TDD, nel proprio ambiente assicurandosi che il componente sia compilabile e funzionante. Esegue infine l'integrazione verso il Version Control System.

2. **Continuous Integration Server:** All'interno dell'infrastruttura di rete del team deve essere presente una particolare istanza detta **Continuous Integration Server**. Esso è collegato al VCS e reagisce in modo automatico a ogni nuovo cambiamento all'interno dei vari Branch dell'architettura di sviluppo.

3. **Compilazione standardizzata:** Il CI Server recupera quindi la feature appena integrata nel VCS e ne esegue la compilazione. L'ideale è che questa sia la **sola ed unica compilazione** eseguita all'interno del processo di sviluppo orientato al deployment.

4. **Test Automatizzati:** Il componente compilato viene quindi testato secondo la stessa batteria di Unit Tests eseguiti in locale dallo sviluppatore. Ammettendo che la prima batteria sia completata con successo, il server si occupa di integrare il componente nel resto del sistema (ottenendo la versione costantemente aggiornata dal Master Branch) ed eseguire una seconda batteria di Integration, Smoke e Performance Tests atti a verificare l'effettiva possibilità di integrazione del nuovo componente.

5. **Feedback Loop:** Sia in caso di successo che di fallimento, il Continuous Integration Server si occupa di notificare lo stato a tutti coloro che possono trarre vantaggio da questa informazione.

   Successo In caso di successo la nuova feature può essere integrata in sicurezza nel Master Branch, il sistema si occupa quindi di notificare l'evento al Developer oppure, ancora meglio, di aprire una richiesta di merge (**Pull**

Request) tra il ramo della Feature e quello principale. Il VCS valuterà in automatico la corretta sequenza di azioni utili ad integrare le due versioni disallineate.

Fallimento In caso di fallimento le criticità devono essere risolte nel minor tempo possibile, il sistema si occuperà di fornire al Developer tutte le informazioni necessarie ad ottimizzare il processo di correzione, questo include ad esempio lo Stacktrace di errore, grafici e metriche di performance.

Si noti come la compilazione e l'esecuzione dei test avvenga in un ambiente esterno e controllato, in questo modo è possibile standardizzare il processo e replicare eventuali condizioni critiche in maniera deterministica.

3.9.4 Jenkins Blue Ocean - Continuous Integration Server

Jenkins [13] è un Continuous Integration Server open source che permette di creare pipeline di integrazione tramite un apposito linguaggio descrittivo. L'installazione e configurazione del server è semplice ed intuitiva grazie ad un'apposita interfaccia web, la community di supporto mette inoltre a disposizione una vasta gamma di plugin che permettono di espandere le funzionalità di Jenkins oltre ogni limite.

Uno tra i plugin più utilizzati al momento è **Blue Ocean** [2], un restile della interfaccia grafica di Jenkins orientato alla realizzazione e gestione di Pipeline di Continuous Integration su progetti Multibranch tramite un editor grafico.

Definizione della Pipeline

All'interno del progetto da integrare con Jenkins deve trovarsi il **Jenkinsfile**, tramite il linguaggio Groovy, direttamente esteso dal Python, è possibile definire gli step di esecuzione della pipeline. Definire la pipeline in un file di configurazione esterno, piuttosto che cablarla staticamente all'interno del server, offre numerosi vantaggi:

Definizione parallela ed incrementale La pipeline viene definita dallo stesso Developer che si è occupato di sviluppare la feature, senza alcuna necessità di interazione con il server Jenkins. La pipeline può inoltre essere definita incrementalmente seguendo le necessità di configurazione e testing che emergono durante lo sviluppo.

Pipeline Branch-specific Il Jenkinsfile è parte integrante dello snapshot integrato nel VCS, di conseguenza è possibile definire pipeline diverse per ogni Branch.

Condivisione e Riutilizzo Il Jenkinsfile stesso è parte del codice Continuamente Integrato, esso è accessibile a tutti i componenti del team in maniera trasparente. Lo stesso Jenkinsfile può inoltre essere utilizzato per differenti progetti che posseggono comuni caratteristiche, senza la necessità di modificare gli step della pipeline. Ad esempio, due progetti basati su SBT, per quanto relativi a differenti applicazioni, avranno una sequenza di step identica.

In Appendice A, Listato A.4 è riportato un esempio di Jenkinsfile.

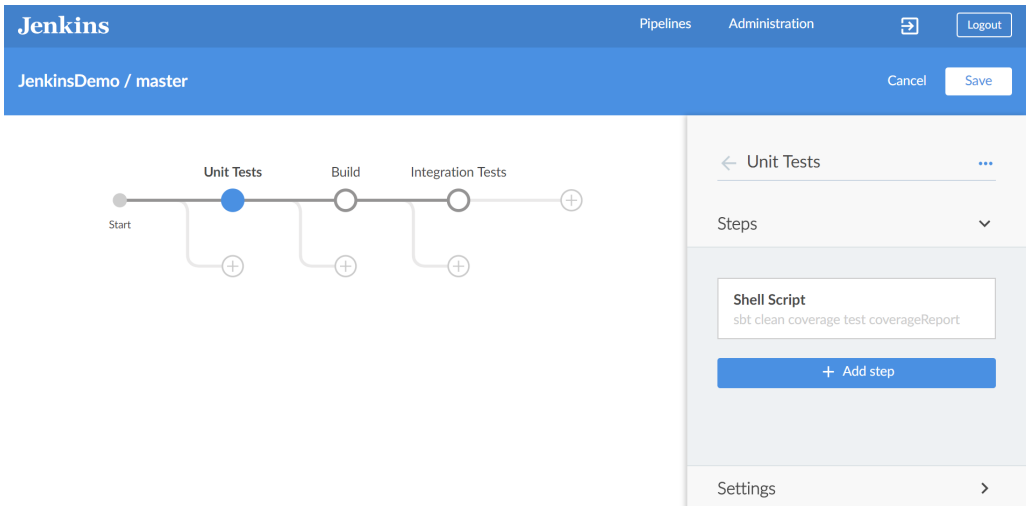


Figura 3.10: Blue Ocean - Pipeline Editor

3.9.5 Vantaggi del Continuous Integration

Adottare il Continuous Integration richiede indubbiamente un cambiamento culturale non trascurabile del modello di lavoro del team, ogni membro deve essere in grado di implementare i propri test, è necessario creare e configurare il Continuous Integration Server (facilmente risolvibile con IaC e CM), ma soprattutto è necessario che ogni individuo del team si impegni ad integrare le proprie modifiche in modo costante, idealmente ad ogni modifica funzionate, in pratica almeno una volta al giorno.

I vantaggi ottenuti da questo sforzo sono molteplici:

- La sequenza di test automatizzati permette di garantire la qualità del software con minor sforzo, diminuendo notevolmente la quantità di bug portati in produzione;

- La compilazione e rilascio del master Branch non presenta più alcun problema di integrazione e disallineamento dal momento che rappresenta la versione più aggiornata e, teoricamente, pronta per il rilascio in ogni momento;

- Ogni feature è sviluppata nella sua completezza da subito, test non automatizzati o peggio svolti da individui esterni al team portano a lunghi tempi di attesa nel quale inizia lo sviluppo della feature successiva. In caso di problemi nella prima, lo sviluppatore è costretto a tornare sui suoi passi con possibilità di confusione ed ulteriori errori.

- Il costo dei test si riduce drasticamente, il lavoro che prima veniva fatto manualmente da un apposito team, ora può essere fatto in tempi nettamente minori dal CI Server. Di conseguenza il team di QA può dedicarsi a migliorare ulteriormente la qualità del prodotto software.

- Il sistema finale è più stabile, dato che ogni componente è stato testato nel dettaglio, sia singolarmente che nel complesso del sistema;
- Avere un feedback immediato, visivo e condiviso tra tutto il team aumenta il grado di trasparenza e responsabilizzazione, diminuendo il Debito Tecnico accumulato durante lo sviluppo.
- La frequenza di rilascio aumenta esponenzialmente.
- Le condizioni generali di lavoro migliorano.

3.10 Continuous Testing (CT)

Il concetto di Continuous Testing definito dal movimento DevOps si differenza da quello di **Automated Testing** esaminato nella sezione precedente.

3.10.1 Automated Testing

L'Automated Testing si riferisce alla pratica di automatizzare le batterie di test in modo da ottimizzare il lavoro di Testers e QAs, mettendo in luce e risolvendo il maggior numero di bug possibile.

3.10.2 Continuous Testing

Il concetto di CT è al tempo stesso simile ma completamente differente da quello di AT. Nel CT vengono comunque eseguite batterie di test automatizzati, essi però non sono soltanto utili ad identificare e risolvere bug. In un'ottica di Continuous Deployment, dove il prodotto viene rilasciato in maniera automatica ad ogni nuova versione stabile e funzionante, i Test devono essere definiti nel modo più preciso e completo possibile dal momento che rappresentano l'unica garanzia di qualità e difesa dal rilascio di bug in produzione.

Indicatore di Rischio

Ogni Test non è solo un metodo per eliminare bug, diventa un potente indicatore del rischio commerciale darivato dal rilascio di un possibile candidato che ha passato tutte le batterie. Ad esempio, la **Coverage** è un indicatore che descrive il numero di righe di codice testate rispetto al totale di righe scritte. Una Coverage del 10% indica quindi che solo per una piccola parte del codice scritto è garantito il corretto funzionamento, di conseguenza il relativo candidato rappresenta un elevato rischio sul mercato.

Indicatore di Avanzamento

Supponendo che i Developers abbiano adottato il Test Driven Development, le funzionalità già implementate restituiranno un esito positivo mentre quelle ancora in fase di sviluppo risulteranno "pending". La Coverage restituirà quindi un valore proporzionale al grado di avanzamento del progetto, un prodotto con Coverage 10%, cioè nelle sue prime fasi di vita, rappresenterà un elevato rischio sul mercato.

Il punto focale nell'analizzare il Master Branch, costantemente candidato per il rilascio, passa quindi dal mero funzionamento a un molto più significativo livello di rischio accettabile.

3.11 Continuous Delivery

Il Continuous Delivery consiste nell'espansione del concetto di Continuous Integration a tutta la fase di sviluppo del prodotto software. Così come Developers e Operations integrano giorno per giorno le modifiche fatte sul proprio codice, anche il prodotto software deve poter essere integrato in maniera continua e costante. Si viene così a creare uno stream continuo di aggiornamenti, automatizzati e garantiti

dai maggiori standard qualitativi, che permette di soddisfare l'incessante richiesta di features del mercato odierno.

Il movimento DevOps identifica con la sigla **CD** sia il Continuous Delivery che il Continuous Deployment, facendo riferimento alle considerazioni fatte quanto la mancanza di un manifesto, la scelta di questa nomenclatura non può fare altro che introdurre ulteriore confusione in coloro che desiderano abbracciare la metodologia.

Il **Continuous Delivery** consiste nella **possibilità** di rilasciare in produzione ogni nuova feature integrata nel Master Branch.

Il termine "rilascio" all'interno della frase è da intendersi con una semantica di "Release" piuttosto che di "Deploy":

Per **Deploy** si intende l'inserimento di un componente software all'interno di un determinato ambiente. Nel contesto di una linea di produzione, l'ambiente target sarà l'architettura dove il componente software verrà utilizzato al termine del processo di sviluppo. Il Deployment è quindi un processo tecnico e pratico che coinvolge una serie di problematiche quali il corretto funzionamento del software e l'architettura necessaria ad una transizione di versione sicura ed affidabile.

Per **Release** invece, si intende il rendere disponibile una determinata feature del prodotto software ai suoi utilizzatori finali. Il Release è il punto focale del Continuous Delivery:

- Il **Waterfall Model** impediva il Release fino a quando tutte le features non fossero state sviluppate. Il valore di business del prodotto era quindi nullo per tutta la fase di sviluppo, manifestandosi solo dopo il suo rilascio.

- Il movimento **Agile** impone il Release costante per mostrare nuove funzionalità al cliente, parte integrante del processo di sviluppo. Questa forzatura lasciava quindi ben poco spazio alla scelta di business e alla strategia.

- Il **Continuous Delivery** lascia totale libero arbitrio sulla strategia di Release. Il valore del prodotto si manifesta quindi fin da subito sotto forma di valore potenziale di Release.

3.11.1 Disaccoppiare Deploy e Release

Il movimento DevOps presenta numerose tecniche per disaccoppiare la necessità di aggiornamento continuo e la scelta di rendere fruibile una determinata feature al consumatore; seguendo l'ideologia "Keep It Simple, Stupid" (KISS), il metodo più semplice ed efficace è dato dalle **Toggle Features**.

Supponendo di aver disaccoppiato tra loro le features, seguendo i principi del Continuous Integration, si racchiude ogni nuova feature all'interno di un componente che ne permette l'abilitazione "by-need". Il software presenta inoltre un metodo per modificare lo stato della feature dall'esterno, senza la necessità di compilare nuovamente il tutto.

In questo modo la pipeline può essere estesa anche alla fase di rilascio, le nuove features saranno testate in ogni dettaglio dalla pipeline di CI, il nuovo aggiornamento verrà però rilasciato con tali features disabilitate in modo da non comportare un cambiamento drastico per l'infrastruttura e per l'utilizzatore. Il Release di una feature diventa quindi una pura scelta di business, permettendo alle figure solitamente distanti dal processo di sviluppo, come il reparto marketing, di trarre enorme vantaggio dal valore potenziale del prodotto.

3.11.2 Dark Launching

Il Dark Launching è una tecnica, utilizzata da Facebook, AirBnB e dai maggiori distributori di servizi software, per la gestione delle Release in maniera graduale e controllata. Secondo il Dark Launching è possibile iniziare a convertire parte del valore potenziale di una feature anche prima del momento strategico ottimale, la nuova

feature viene quindi messa a disposizione di un piccolo sottoinsieme di utilizzatori, consci o inconsci.

Il piccolo subset di istanze complete viene quindi monitorato nel dettaglio per mettere in luce le performance in produzione, in vista dello scaling, il grado di soddisfazione dei clienti e l'eventuale presenza di bug e anomalie sfuggite al Continuous Testing.

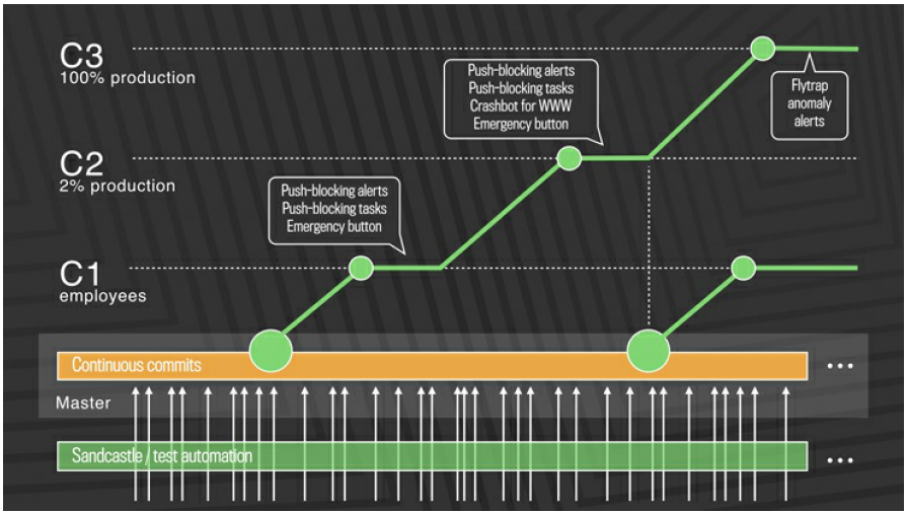


Figura 3.11: Facebook Dark Launching secondo Facebook

Facebook adotta un rilascio in tre fasi per la gestione degli aggiornamenti più significativi:

1. La nuova feature viene rilasciata tra i dipendenti dell'azienda, monitorando il prodotto e garantendo la presenza di un "arresto forzato" prima dell'effettivo rilascio.

2. Si passa quindi al rilascio del 2% in produzione, in questa fase si collezionano segnali di alert, ponendo grande attenzione ad eventuali sovraccarichi del sistema o ai punti più critici della feature.

3. La feature viene infine rilasciata in produzione ed inserita nel processo di Continuous Monitoring.

3.12 Continuous Deployment

Il Continuous Deployment consiste nella completa industrializzazione del processo di sviluppo e distribuzione del prodotto software. Anche il Deploy viene automatizzato creando un effettivo flusso di rilasci generato dalla semplice pressione di un pulsante.

Il Continuos Deployment non è però una tecnica adatta a tutte le aziende, raggiungere un tale livello di automazione presuppone infatti la completa comprensione e adozione delle pratiche precedenti e un alto livello di fiducia nella **Deployment Pipeline**.

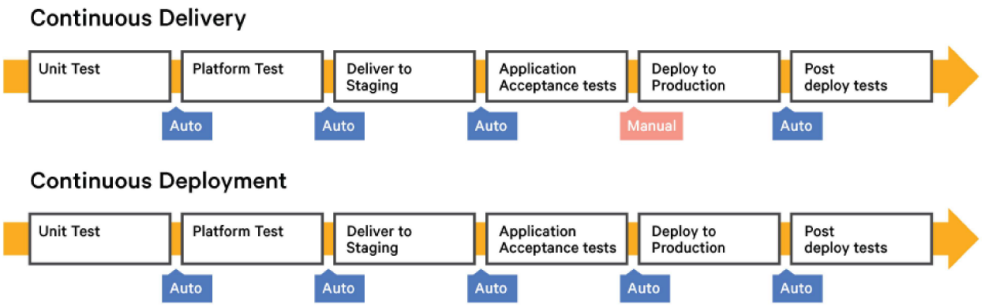


Figura 3.12: Delivery e Deployment

Eliminare ogni intervento umano implica che ogni fase del processo sia implementata al meglio:

Una sola compilazione

Il software deve essere compilato una e una sola volta, in un ambiente identico a quello di produzione, isolato e monitorato. La Deployment Pipeline deve poter identificare in maniera univoca ogni singolo prodotto compilato e testato in modo da poter selezionare il miglior candidato o eliminare quelli corrotti.

Test come unica garanzia di qualità

La Deployment pipeline deve contenere test perfetti, in grado di garantire la massima copertura e il massimo livello qualitativo, come richiesto dagli standard di produzione. E' inoltre necessario introdurre **Smoke Tests** in grado di verificare che l'ambiente di esecuzione sia consistente con le specifiche e che il prodotto sia effettivamente attivo e funzionante.

Processo di Deployment industrializzato

Anche il processo di Deployment deve essere automatizzato e gestito con componenti software al pari del prodotto in sviluppo. Il processo deve essere testabile, documentato, convergente e idempotente, in modo da poter essere ripetuto in qualsiasi ambiente della fase di produzione. Utilizzando lo stesso metodo per ogni stage si ha la possibilità di verificare l'efficacia dello stesso ben prima del suo effettivo utilizzo in produzione.

Ambienti standardizzati

Per poter standardizzare il deploy è necessario prima di tutto che gli ambienti siano standardizzati, secondo i principi del Configuration Management.

3.12.1 Blue/Green Deployment

Il Blue/Green deployment è una tecnica che permette di gestire in maniera graduale e controllata il Continuous Deployment di un software, solitamente di un servizio web.

L'ambiente di produzione viene replicato, ottenendo due ambienti gemelli chiamati Green Environment e Blue Environment, il primo conterrà l'attuale versione

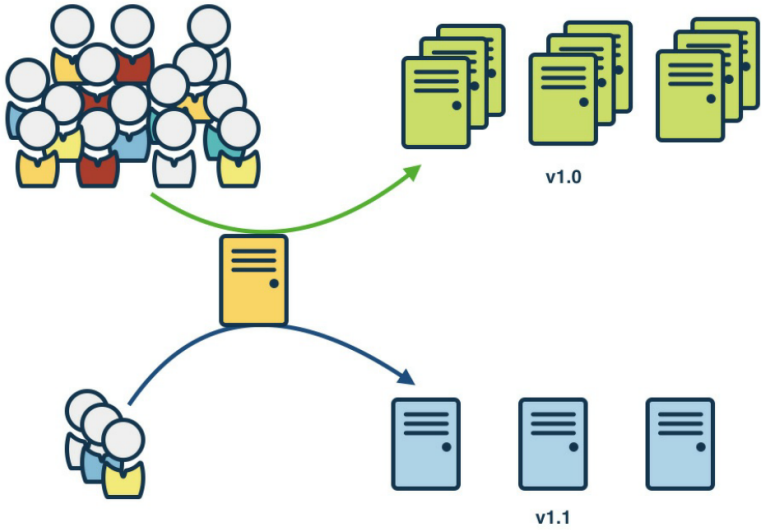


Figura 3.13

del software, utilizzato dal 100% dell'utenza, il secondo sarà dedicato al rilascio della nuova versione. Un terzo elemento dell'infrastruttura si occupa di dirottare una percentuale definita di utenti dalla vecchia alla nuova versione, completa di tutte le nuove funzionalità. La transizione tra Blue e Green è graduale e incrementale, l'utenza viene mano a mano dirottata fino al completo abbandono del Green Environment, durante questa fase si monitorano le performances della nuova istanza allo scopo di rilevare criticità e problemi di scaling.

Al termine della transizione la vecchia istanza può essere mantenuta, al fine di poter effettuare un rollback in caso di necessità, oppure disabilitata. In ogni caso assumerà il ruolo di pilota al successivo rilascio.

Il B/G Deployment si differenzia dal Dark Launching per la necessità di avere due ambienti di produzioni differenti, in ogni caso la configurazione dell'istanza aggiornata può essere gestita con le Toggle Features, inoltre le features sono rilasciate a un subset in continua transizione fino alla totale copertura.

3.12.2 Oltre il prodotto

Nelle sezioni 3.7 e 3.8 sono presentate le metodologie per convertire gli script di Provisioning e Orchestration in componenti software, in tutto e per tutto gestibili allo stesso modo del prodotto sviluppato.
Questo significa che è possibile creare una Continuous Deployment Pipeline anche per l'infrastruttura sul quale è rilasciato il software: ogni nuova modifica alla configurazione delle istanze o all'architettura stessa viene quindi inserita in un VCS, testata, integrata ed applicata in maniera continua e automatizzata.

3.13 Continuous Monitoring (CM)

Il Continuous Monitoring rappresenta la chiusura del feedback loop, fondamentale per il miglioramento incrementale del processo di produzione. Il lavoro del team non termina con il rilascio in produzione, il prodotto deve essere infatti costantemente monitorato per assicurare che le specifiche funzionali e non funzionali siano garantite sul lungo termine. Allo stesso modo, anche la Deployment Pipeline e il lavoro del team deve essere monitorato per garantire che tutto funzioni secondo i più elevati standard di efficienza.

3.13.1 Monitoring post-produzione

Il prodotto deve esser progettato fin dalle prime fasi del processo di sviluppo per integrare strumenti di controllo atti a valutarne il funzionamento:

Test in produzione

Gli stessi test eseguiti nella Deployment Pipeline devono poter essere eseguiti periodicamente sul software in produzione, in modo da verificare costantemente che tutto funzioni senza problemi.

Metriche

Il Software deve esporre una serie di Metriche (KPI), valori atti a valutare il funzionamento del prodotto in modo significativo e duraturo. La definizione dei KPI è strettamente legata all'applicativo del caso. Per un servizio web è significativo monitorare il tempo di risposta alla singola richiesta, oppure il numero di richieste giornaliere; per un modello di previsione è interessante verificare che l'errore fatto sulla predizione converga allo zero.

Due sono i metodi solitamente utilizzati per esporre metriche:

White Box La generazione ed esposizione delle metriche è parte integrante del codice dell'applicativo, KPI di questo tipo sono molto significativi perchè danno un valore costantemente aggiornato sul corretto funzionamento. Un esempio di KPI White-box è l'errore sulla singola previsione.

Black Box Le metriche sono periodicamente collezionate da un software esterno che valuta gli effetti del prodotto sull'ambiente di esecuzione. Ad esempio, una metrica black box è lo stato di funzionamento di un determinato servizio, o il numero di elementi salvati in un database al termine dell'ultima esecuzione di un processo Batch.

3.13.2 Monitoring del processo

Anche la Deployment Pipeline deve essere progettata per esporre KPI utili a valutarne l'efficacia. Ad esempio:

- Tempo necessario a rilasciare un commit in produzione;
- Tempo necessario ad eseguire la fase della pipeline;
- Numero di features sviluppate in un arco di tempo definito;

- Numero di bug evidenziati dalle varie fasi di test della pipeline;
- ...

3.13.3 Prometheus - Collezionare ed Esporre KPI

Prometheus [19] è un tool di monitoring ed analisi di serie temporali sviluppato da SoundCloud [24] nel 2012, dopo un periodo di sviluppo interno è stato rilasciato in licenza open source in modo da renderlo disponibile ad ogni organizzazione. Prometheus permette di collezionare KPI provenienti da differenti sorgenti, sia Black-box che White-box, mettendoli a disposizione per future analisi.

Il tool è strutturando mantenendo la modularità e la customizzazione come punto focale, dando la possibilità di creare un sistema replicato, distribuito e scalabile secondo la necessità del caso.

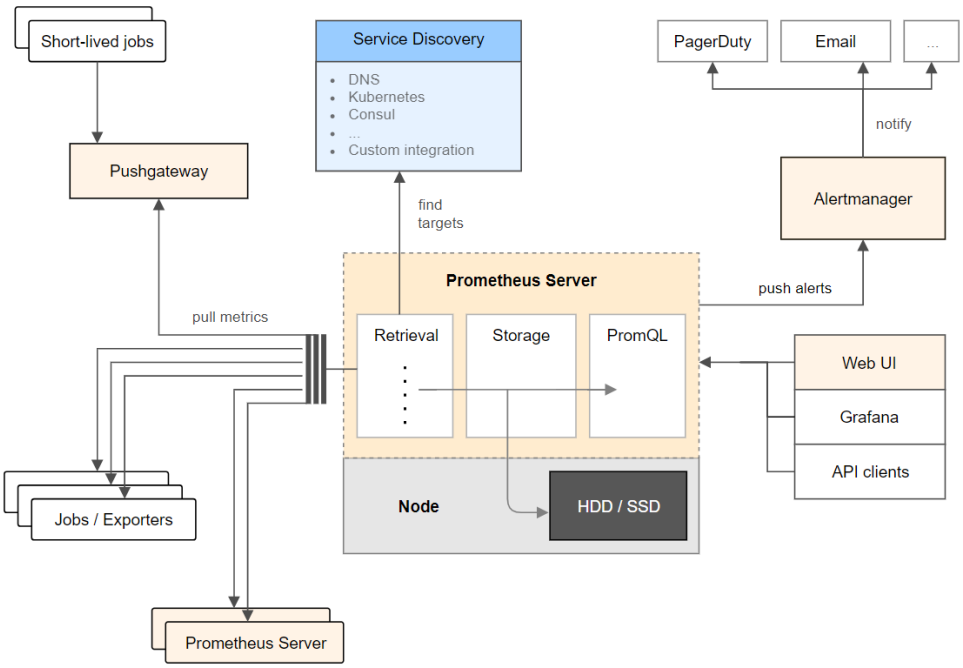


Figura 3.14: Architettura di Prometheus

Prometheus Server

Il core del sistema è dato dal Prometheus Server, esso si occupa di collezionare le metriche provenienti dalle differenti sorgenti, mantenerle in maniera sicura ed efficiente, ed esporle per le future analisi. Un sistema di Service Discovery permette di trovare le varie istanze monitorate dal Server, la configurazione dei target è fatta tramite un file esterno scritto in YAML.

Scraping Il Prometheus Server è in grado di collezionare metriche da qualsiasi fonte, a patto che queste siano rappresentate nel formato corretto, tutto ciò di cui Prometheus ha bisogno è un endpoint a cui puntare:

- **Pushgateway**: Il pushgateway è un server installabile in qualsiasi nodo della rete verso il quale i software batch, la cui esecuzione è saltuaria, mandano le metriche in modo che esse possano essere mantenute e rese disponibili al Prometheus Server. Una serie di librerie, accessibili in numerosi linguaggi, permettono di esporre le metriche necessarie verso il pushgateway, secondo i pattern White box.

- **Exporters**: servizi che permettono di accedere ad informazioni di basso livello tramite un endpoint di rete. Il più comune è il **Node Exporter** che permette di collezionare informazioni riguardo le risorse hardware del nodo in analisi.

- **Jobs**: i servizi real time possono essere monitorati esponendo i propri KPI in un endpoint dedicato, il Prometheus Server penserà ad eseguire periodicamente lo scraping dei dati,

- **Prometheus Servers**: Prometheus può inoltre monitorare altre istanze del server permettendo di rendere modulare e facilmente espandibile l'architettura di rete del cluster.

Exposing Le metriche collezionate sono quindi esposte verso due interfacce:

- **Alermanager**: L'Alertmanager permette di definire regole sulle metriche estratte dal Server e scatenare allarmi in caso la situazione si faccia critica. Solitamente l'analisi è fatta non sul singolo evento ma su una variazione temporale.

- **Dashboard**: I dati vengono quindi visualizzati ed analizzati tramite interfacce grafiche e API client.

3.13.4 Grafana - Dashboard di analisi OLAP

Grafana [7] è una piattaforma open source che permette di collezionare metriche provenienti da una miriade di fonti differenti, numerosi plug-in di input e output permettono di customizzare nei minimi dettagli le funzionalità offerte dalla piattaforma. Inoltre una stabile comunità di sviluppatori supporta costantemente il progetto, condividendo Dashboard e plug-in, oltre alla propria esperienza.

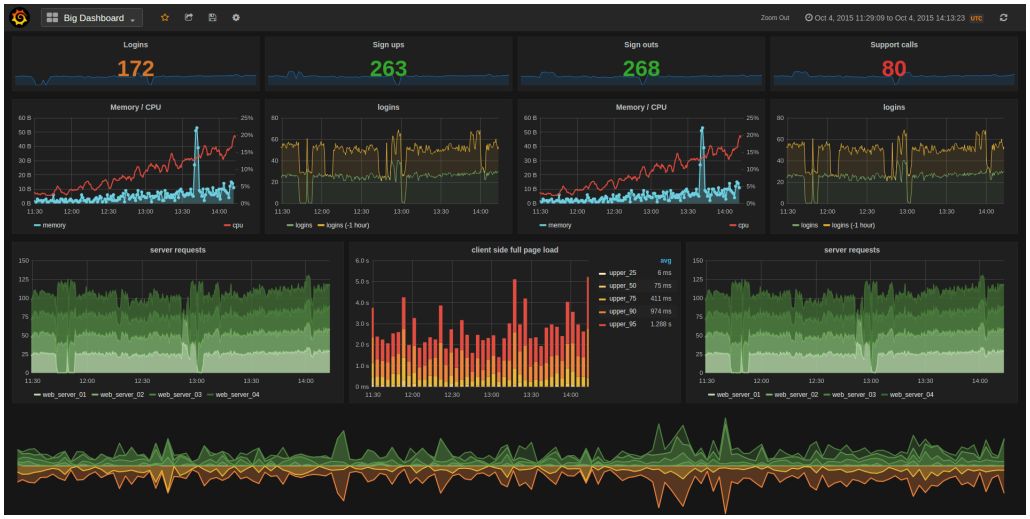


Figura 3.15: Grafana Dashboard

Molteplici Sorgenti Essendo una piattaforma esterna, Grafana permette di unificare gli stream di KPI provenienti da molteplici sorgenti, non solo Prometheus, mostrando una vista unificata ed aggregata.

Dashboard Efficaci Grafana permette di creare dashboard Efficaci, in grado di fornire tutte le informazioni necessarie alla valutazione real time dello stato del sistema con un semplice colpo d'occhio.

Query OLAP Vi è inoltre la possibilità di eseguire query real time sulle metriche collezionate, modificando il livello di granularità della visualizzazione, aggregando fonti differenti e unendo dashboard in base alle necessità del momento.

Sistema di Alert Infine possono essere definiti Alerts direttamente sui grafici mostrati. Tali eventi possono inoltre essere notificati a molteplici destinatari come email, dispositivi hardware, chatops,... .

3.13.5 ChatOps

Il ChatOps [9] consiste in un cambiamento del metodo di comunicazione del team spostandolo da scambio di mail o messaggi 1-1 alla comunicazione totale e trasparente all'interno di gruppi di chat. Tool come **Slack** o **HipChat** permettono di gestire le comunicazioni in un ambiente strutturato e altamente customizzabile, in cui è possibile creare canali dedicati ai singoli progetti, integrare applicazioni esterne e documentazione di vario tipo.

La vera natura dei ChatOps si manifesta quando tali canali di comunicazione trasparente non sono utilizzati solo per organizzare il lavoro ma anche per interagire con il sistema stesso. E' infatti possibile creare speciali utenti, chiamati **BotUsers** che nascondono al loro interno software in grado di fare da bridge tra la chat e il sistema esterno, in entrambe le direzioni.

Dashboard e metriche Il Bot può interagire con Prometheus e Grafana al fine di interrogare il sistema di Scraping, visualizzando la singola metrica o una più significativa dashboard, direttamente dove e quando serve.

Stato della Pipeline Un test è fallito, una Build è stata completata, la nuova feature ha passato i test ed è pronta per la pull request vero il Master Branch, la Deployment Pipeline può notificare la chat in un apposito canale ad ogni esecuzione, permettendo ai responsabili di intervenire tempestivamente in caso di problemi. In un caso di Continuous Delivery il BotUser può inoltre essere usato per scegliere se rilasciare o meno una feature in produzione.

Alerts Gli Alert generati da Prometheus e Grafana possono essere consegnati per mail, o ancora meglio all'interno della chat, mandando notifiche a coloro che hanno partecipato allo sviluppo del componente compromesso, creando un canale dedicato alla risoluzione della criticità e arricchendolo con tutte le informazioni utili alla rapida risoluzione del problema. La fase di discovery successiva a un alert diventa quindi totalmente automatizzata e indipendente del caso specifico, dando la possibilità a Dev e Ops di focalizzarsi completamente sulla rapida risoluzione del problema.

Capitolo 4

Caso d'Uso: dallo sviluppo al deployment

4.1 Motivazioni

Alla luce di quanto analizzato nei due capitoli precedenti, appare chiaro il vantaggio che il mondo dei Big Data può trarre dall'adozione delle metodologie del Movimento DevOps.

Il Data Warehouse, strutturato sotto forma di cluster di nodi entry-level, richiede abilità di Provisioning e Orchestration non ottenibile con le tradizionali tecniche di gestione. La combinazione di Configuration Management e Infrastructure As a Code trova terreno fertile, dovendo configurare grandi numeri di istanze più o meno simili, in maniera automatica e by-need secondo le necessità del caso, .

Dovendo processare grandi quantità di dati, i processi di elaborazione, sia batch che real time, richiedono un livello di affidabilità e robustezza non trascurabile, per

garantire la Veracità dei Big Data è necessario che ogni informazione sia processata al meglio in modo da non compromettere il Decision Making. Così come i dati sono altamente eterogenei e veloci, anche i processi che li elaborano devono essere progettati ed aggiornati alla stessa velocità, una Pipeline di Continuous Integration, integrando il Continuous Testing, può senza dubbio garantire tali livelli di affidabilità.

Per quanto riguarda il caso specifico dei processi real time, come possono essere i software di Analysis e Decision Making, è necessario garantire un funzionamento del servizio 24/7. Diventano quindi indispensabili le metodologie del Continuous Delivery e Deployment, in modo da garantire un servizio costantemente disponibile.

Il Monitoring e la comunicazione real time sono, infine, requisiti indispensabili per amministrare in maniera controllata ed efficace un sistema così complesso come quello Big Data; il team deve essere in grado di differenziare le condizioni di Fast Fail, gestite dall'infrastruttura Big Data, dai casi critici in modo da garantire la qualità del servizio.

Ritengo quindi che il mondo Big Data presenti il giusto grado di varietà, utile a mettere in luce i vantaggi apportati dal movimento DevOps.

4.2 Caso d'uso

In questo elaborato si vogliono mostrare i benefici del movimento DevOps, applicati a un caso d'uso pratico quale l'implementazione di un **Raccomandatore di Film** basato sul dataset **Movielens**.

Verranno messe in luce problematiche e vantaggi delle 4 principali tipologie di processi in ambito Big Data: Batch ETL, Batch Machine Learning, Real time ETL, Real time Machine Learning; evidenziando come le metodologie viste nel capitolo precedente possano portare a un prodotto software migliore, realizzato col minimo sforzo. Il caso d'uso è stato volutamente mantenuto semplice, in questo modo si è voluto dare più spazio alle tecnologie emergenti utilizzate e alla definizione di un caso generale di applicazione del DevOps.

4.3 Struttura

Nei capitoli successivi verrà illustrato il caso d'uso nel dettaglio, l'analisi seguirà la seguente struttura:

Architettura Analisi dell'architettura cloud che ospita il progetto, la configurazione delle istanze e la distribuzione di processi e servizi.

Processi di Elaborazione Analisi delle 4 tipologie di processi Big Data, evidenziando le caratteristiche che li contraddistinguono e l'implementazione adottata.

DevOps Applicazione delle tecniche DevOps al caso d'uso.

Capitolo 5

Caso d'Uso: Architettura

5.1 Architettura Cloud

Nel progettare l'architettura, illustrata in Figura 5.1, si è cercato di ricostruire il più fedelmente possibile, l'architettura di un sistema Big Data. Allo stesso tempo si è cercato di disaccoppiare servizi e responsabilità in modo da rendere il sistema modulare e facilmente espandibile, sia verticalmente che orizzontalmente.

Questo capitolo vuole semplicemente introdurre i tre nodi del cluster con i relativi ruoli all'interno del sistema, i capitoli successivi si occuperanno di analizzare nel dettaglio i servizi strettamente legati al caso d'uso e all'applicazione delle metodologie DevOps.

Il Cluster è implementato su **Google Cloud Platform**, grazie al servizio di **Compute Engine (CE)**.

5.1.1 Cloudera VM

Configurazione Hardware CE dotata di 4 vCPU, 18 GB RAM, 35GB HDD permanente standard, Centos-7-v20171129, zona us-east1-b. La macchina è la più

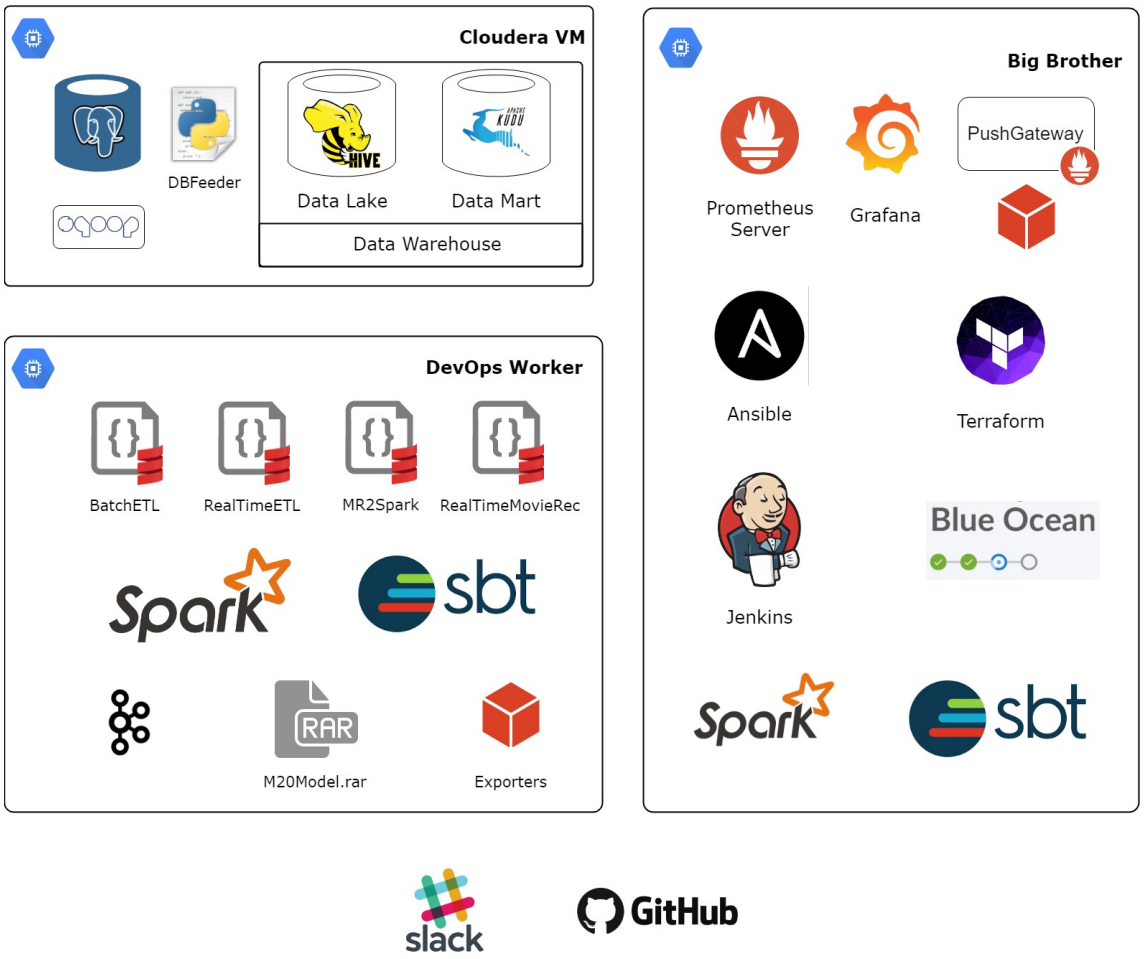


Figura 5.1: Architettura del sistema

potente del cluster, essendo il nodo master del sistema è stato dotato di maggiori capacità in modo da poter gestire comodamente i processi di Cloudera CDH.

Ruolo

Cloudera VM è pensata per ospitare il Data Warehouse del sistema implementato grazie alla piattaforma **Cloudera**. Il Data Warehouse è fisicamente e logicamente composto da un Data Lake e un Data Mart, implementati con tecnologie differenti. Il CDH offre inoltre servizi accessori, quali un database PostgreSQL e Sqoop1 utilizzati nelle fasi di Ingestion e Estraction Batch.

Hive Data Lake

Nel Data Lake, realizzato come un insieme di tabelle Hive, vengono memorizzati i dati nella loro forma originale, senza preoccuparsi di pulizia o trasformazioni. Questo datastore è pensato per accogliere tutti i dati relativi all'organizzazione senza fare distinzioni riguardo il singolo applicativo. Seppur non effettivamente necessario per il funzionamento del caso d'uso, si è deciso di mantenerlo nell'ottica di definire un'architettura il più possibile generale. L'architettura di Hive, come già discusso, permette di memorizzare ed accedere ai dati tramite un'interfaccia SQL-like; essendo un metodo prossimo al formato di memorizzazione originale dei dati, ho valutato che potesse garantire il minimo intervento in termini di gestione.

Kudu Data Mart

Il Data Mart invece conterrà i dati relativi ad una singola applicazione, in questo caso tutto il necessario ad implementare ed utilizzare il Raccomandato di Film. Tali dati sono più raffinati di quelli contenuti nel Data Lake, due processi ETL si occupano infatti della pulizia e trasformazione delle informazioni in modo da renderle adatte alla successiva analisi.

La scelta di utilizzare un Data Mart Kudu è derivata dalla necessità di memorizzazione ed accesso, rapido ed efficiente, dei campi di tipo Double dei `Ratings`. Grazie alla memorizzazione in formato colonnare di Kudu è possibile ottimizzare l'elaborazione minimizzando il numero di accessi al datastore.

Database PostgreSQL

Cloudera CDH include, nella sua configurazione standard, un database PostgreSQL versione 9.2. Questo tipo di RDBMS, seppure datato rispetto agli altri sistemi di

gestione di database, offre una maggior robustezza e una ricca documentazione, rendendolo uno strumento comodo da utilizzare e facile da interfacciare. Lo script Python `DBFeeder.py` si interfaccia con tale database per simulare l'interazione utente con un sistema a monte.

5.1.2 DevOps Worker

Configurazione Hardware CE n1-standard-2 dotata di 2 vCPU e 7.5 GB RAM, 40GB HDD permanente standard, Centos-7-v20171129, zona us-east1-b. Si è voluto utilizzare una macchina dalle caratteristiche standard per mettere in luce il vantaggio dello scaling orizzontale.

Ruolo

Il DevOps Worker rappresenta un Edge Node, dedicato alla computazione all'interno del cluster. Su di esso vengono eseguiti i quattro processi principali di elaborazione e tutto il necessario alla loro esecuzione.

Processi

I processi, analizzati nel prossimo capitolo, vogliono mostrare le quattro principali combinazioni di elaborazione all'interno di un sistema Big Data. Per garantirne l'esecuzione, il nodo è dotato di Spark 2.2.0 ed SBT 1.0.3, utilizzato per la gestione delle dipendenze.

Servizi

- **Confluent Kafka** è utilizzato dal processo di ETL Real Time per estrarre i dati memorizzati nel database PostgreSQL, collocato in remoto sul nodo Cloudera VM.

- **Node_Exporter** e **DevOpsMetrixExposer** sono due daemon che espongono metriche di interesse per sistema di Continuous Monitoring. Il primo espone metriche relative allo stato hardware del nodo, il secondo mette a disposizione informazioni riguardo l'esecuzione dei processi.

5.1.3 Big Brother

Configurazione Hardware CE n1-highmem-2 dotata di 2 vCPU e 13 GB RAM, 40GB HDD permanente standard, Centos-7-v20171129, zona us-east1-b. Pur non richiedendo grosse capacità computazionali, la macchina necessità di più memoria di un semplice edge-node a causa del grande numero di servizi installati.

Ruolo

Il nodo Big Brother è pensato per essere un nodo di supporto e controllo del cluster, in esso sono collocati tutti i servizi propri del processo di sviluppo devops-based, oltre ai tool necessari al Provisioning, Orchestration e Instrumentation del cluster.

Servizi

- Il **Pushgateway** permette ai processi batch di trasmettere metriche in modalità push e white-box;

- Il **Prometheus Server** effettua lo scraping delle metriche esposte dagli Exposer, presenti sia sul DevOps Worker sia sul Big Brother stesso, dal Pushgateway e dal servizio Machine Learning Real Time di raccomandazione;

- **Grafana** offre un'interfaccia a dashboard utile per monitorare le metriche raccolte ed attivare allarmi in caso di necessità;

- **Ansible** permette di gestire il Provisioning dei nodi del cluster, non solo in termini di espansione ma anche di monitoring dello stato, oltre al deploy sicuro dei processi in produzione;
- **Terraform**, associato ad Ansible, gestisce l'espansione orizzontale del cluster secondo la necessità del caso;
- **Jenkins**, sfruttando l'interfaccia **Blue Ocean**, mette a disposizione il server di Continuous Integration e Continuous Deployment;
- **Spark 2.2.0** e **SBT 1.0.3** contribuiscono a creare un ambiente di sviluppo identico a quello di produzione, sono utilizzati all'interno della pipeline di CD per effettuare i test e la compilazione dei processi.

5.1.4 Servizi Esterni

Per completezza sono mostrati anche **Slack**, utilizzato per la gestione delle comunicazioni intra-team e la notifica di eventi, e **GitHub**, Version Control System utilizzato per il Continuous Integration del codice implementato.

5.2 Google Cloud Platform (GCP)

Google Cloud Platform offre un servizio di cloud computing hostato ed offerto da Google, è stato utilizzato in questo PoC sfruttando i 300 $ messi a disposizione per ogni account gmail registrato.

GCP offre numerosi servizi a supporto del calcolo e dello storage Big Data, in particolare sono state utilizzate le virtual machine Compute Engine, pratiche per la completa personalizzazione e creazione by-need. Grazie alla gestione basata su progetti, è possibile creare cluster di macchine automaticamente identificate tra loro senza doversi preoccupare di DNS o IP dinamici.

Infine il sistema di connessioni a chiavi SSH e la gestione dei ruoli IAM ha permesso l'Orchestration del cluster in totale sicurezza.

Capitolo 6

Caso d'Uso: Processi di Elaborazione

6.1 Movie Recommender

Il sistema implementato, mostrato in Figura 6.1, sfrutta i dati del dataset Movielens M20 [17] per generare un modello di raccomandazione. Come si può notare, il sistema si basa sula **Two Layer Architecture** illustrata in Figura 2.1.

I dati del dataset sono memorizzati nel sistema sotto forma di file `.csv`, al fine di simulare l'interazione utente con un servizio esterno tali dati sono caricati incrementalmente dallo script Python `DBFeeder.py` e memorizzati nel Database PostgreSQL.

In base alla natura dei dati, questi sono successivamente ingeriti da due diverse pipeline:

- I dati relativi ai `Movies` sono tendenzialmente statici, quindi caricati un'unica volta tramite un processo Batch;

- I `Ratings` sono trattati come uno stream real time, supponendo che questi derivino dall'interazione di utenti con un servizio a monte.

Due processi si occupano quindi di Estrarre, Pulire e Caricare infine i dati su Data Lake e Data Mart.

Un Processo Batch, periodicamente, estrae le valutazioni ricevute fino a quel momento e, sfruttando l'algoritmo **Alternating Least Squares**, genera un modello di raccomandazione basato sul **Collaborative Filtering**. Tale modello è quindi caricato da un processo Real time che, attraverso un'interfaccia grafica, permette a utenti o altre entità del sistema di richiedere raccomandazioni mirate.

6.1.1 Implementazione

Per quanto riguarda l'implementazione pratica, tralasciando lo script di Ingestion realizzato in Python, i processi di effettiva elaborazione sono stati implementati in **Scala 2.11.8**, sfruttando l'engine **Spark 2.2.0** per processare le strutture dati in gioco. La gestione delle dipendenze è stata lasciata al framework **Scala Build Tool (SBT)** che ha permesso inoltre la creazione di componenti eseguibili standalone, utili alla distribuzione.

Le prossime sezioni analizzeranno nel dettaglio le fasi di Ingestion, ETL e Analysis.

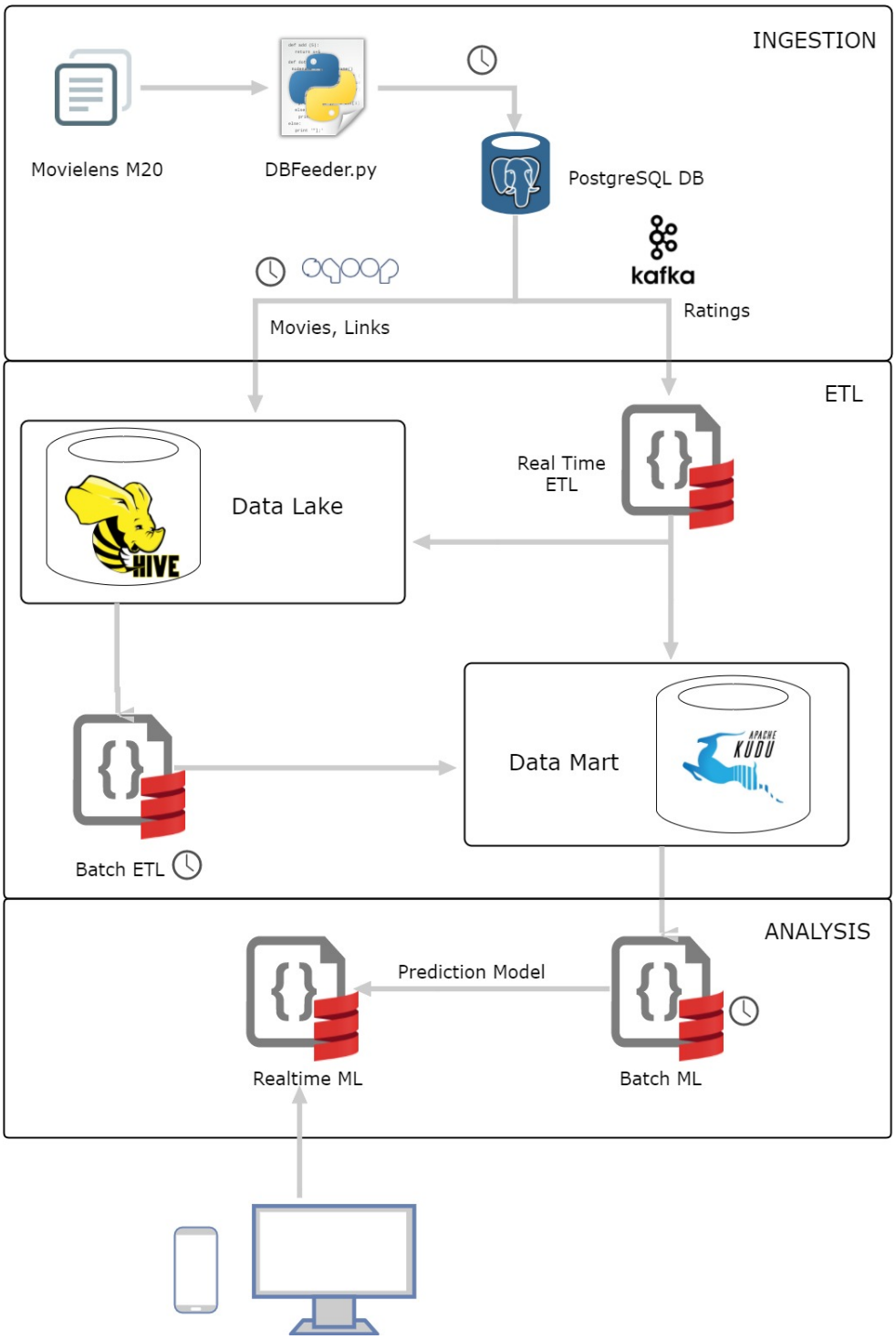


Figura 6.1: Processi di Elaborazione

6.2 Ingestion

La fase di Ingestion consiste nella simulazione del caricamento dei dati riguardanti le valutazioni nel database PostgreSQL, gestendoli secondo la loro natura.

6.2.1 Dataset Movielens M20

Movielens è un progetto di GroupLens [8], il Social Computing Research Group del dipartimento di Computer Science and Engineering dell'università del Minnesota. Il progetto mira ad analizzare il comportamento di un gruppo di utenti all'interno di un sito web, studiando le tecniche di raccomandazione, e di design di interfacce utente.

Il gruppo mette inoltre a disposizione il dataset utilizzato nelle proprie ricerche, in modo che chiunque possa trarre vantaggio dalla grande quantità di utenti registrati sul portale. Il Dataset è disponibile in differenti versioni, raccolte in epoche diverse e con una diversa struttura interna; per questo elaborato ho sfruttato il dataset Movielens M20, contenente **20 milioni di valutazioni**, generate da **138.000 utenti** su **27.000 film**.

Struttura

Movielens M20 viene fornito sotto forma di 6 file `.csv` relativi a 6 differenti tabelle: movies, links, ratings, tags, genome_tags e genome_scores. La Figura 6.2 mostra la struttura dei campi.

Users Gli utenti contenuti nel dataset rappresentano coloro che si sono registrati sul portale Movielens, di conseguenza non è scontato che i `movieId siano`

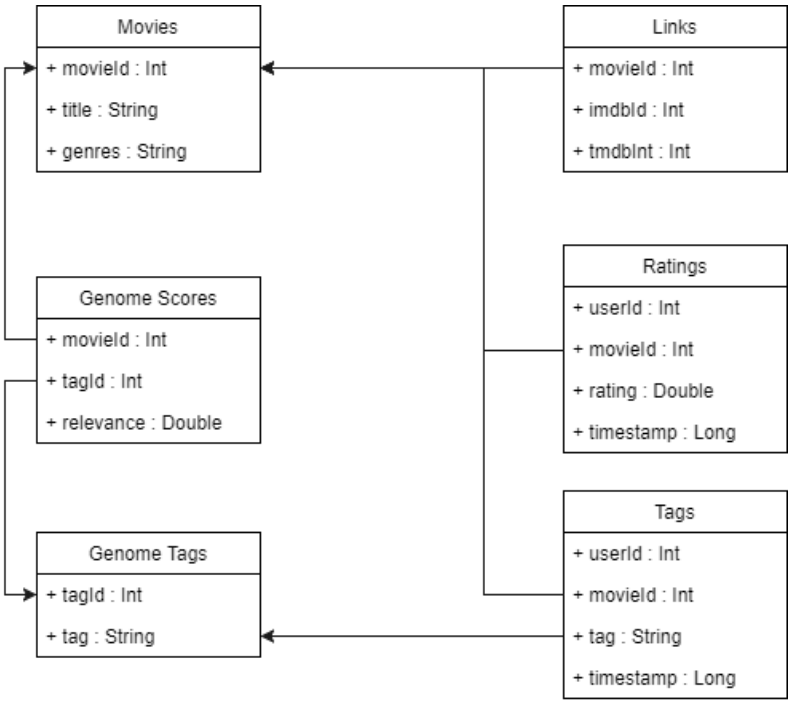


Figura 6.2: Movielens M20

`continui`. I loro identificativi sono stati resi anonimi e consistenti tra `Ratings` e `Tags`.

Movies Un movie è inserito nel dataset se e soltanto se ha ricevuto almeno un tag o una valutazione, gli id sono consistenti tra `Movies`, `Links`, `Ratings` e `Tags`.

Ratings La valutazione è espressa secondo una scala a 5 stelle, con incrementi di 0.5 (0.5 - 5.0 stelle). I Timestamp sono rappresentati in formato UTC.

Tags I Tag rappresentano metadati assegnati dagli utenti ad un film, possono essere singole parole o piccole frasi, in ogni caso la loro semantica è soggettiva.

Genome Scores e Genome Tags Matrice densa dove ogni movie ha un valore di rilevanza per ogni tag, rappresentano quanto un film rispecchia le caratteristiche

specifiche di un tag [14].

6.2.2 Popolare il Database

Si presuppone che, a monte del sistema, vi sia una qualche piattaforma che permetta agli utenti di assegnare tag e valutazioni ai movies del dataset, ogni nuovo Rating e Tag viene memorizzato nel database per poi essere processato.

Allo scopo di simulare tale interazione, è stato sviluppato un semplice script, in linguaggio Python, che a intervalli regolari inserisce nuovi dati all'interno del database. L'inserimento varia in base alla natura dei dati:

- **Movies** e **Links** sono tendenzialmente statici, vengono quindi caricati una sola volta, mantenendo comunque la possibilità di aggiunte future;
- **Ratings** e **Tags** sono idealmente generati dall'interazione real time degli utenti con il portale a monte del sistema, lo script si occupa di estrarre giorno per giorno un subset di dati ed inserirli nel database PostgreSQL.

In questo progetto ho deciso di non elaborare i dati di Genome Tag e Genome Score, dal momento che si tratta di informazioni già preprocessate e relativi all'intera collezione di tag.

Appare chiara la presenza di due flussi di informazione differenti, il primo saltuario di tipo Batch, il secondo Real Time. Il sistema gestisce tali flussi grazie a Sqoop e Kafka, tools illustrati nel Capitolo 2.

Per facilitare il lavoro dell'estrazione è stato aggiunto un campo di identificazione, univoco e continuo, abilitando la possibilità di gestire automaticamente l'offset di lettura, per entrambi i flussi di estrazione.

6.3 Estraction Transformation Loading (ETL)

La fase di ETL consiste nell'estrazione dei dati dalle sorgenti originali, in questo caso il solo Database PostgreSQL, la loro elaborazione ed infine il caricamento all'interno del Data Warehouse.

6.3.1 Batch ETL

Estraction - Sqoop

Apache Sqoop, analizzato nel Capitolo 2, è utilizzato per estrarre `Movies` e `Links` dal database PostgreSQL, trasferendoli nel Data Lake (modalità bulk).

In Appendice B, Listato B.1 è mostrato un esempio di estrazione tramite Sqoop. Il Tool si occupa di caricare i `Movies` dal database PostgreSQL verso la tabella Hive `datalake.movies`, l'offset è valutato sul campo `ID`, aggiunto automaticamente dallo script Python all'inserimento nel database. La gestione dell'offset di lettura è automatica, da qui la necessità di avere ID incrementali e continui.

Transformation

I dati relativi a `Movies` e `Links`, automaticamente inseriti nel Data Lake da Sqoop, devono comunque essere trasferiti nel Data Mart in modo da renderli accessibili al Raccomandatore.

Per come è strutturato il Dataset Movielens M20, `Movies` e `Links` non sono strettamente necessari allo scopo del caso d'uso, i `Ratings` infatti contengono già tutto il necessario all'identificazione di un movie. Il vantaggio apportato dalla gestione di queste due entità sta nell'arricchimento di un'eventuale interfaccia utente con titolo, generi e link di riferimento. Ho dunque ritenuto superfluo allocare risorse hardware e software alla memorizzazione di due strutture dati distinte e parzialmente ridondanti.

Il processo **BatchETL** si occupa di estrarre on-demand le due strutture dati, effettuarne il merge e memorizzare i dati nel Data Mart. In particolare:

- Solo il link relativo al **The Movie Database** (tmbdID) viene mantenuto, questo perchè il portale offre un insieme di API open source utili allo Scraping di metadati riguardo il movie selezionato. In vista di un futuro arricchimento il Data Mart contiene il riferimento completo.

- Viene eliminato l'ID univoco inserito in fase di salvataggio sul Database PostgreSQL. Avendo una struttura del tutto identica a un database relazionale, Kudu permette di identificare in maniera univoca i movies tramite il loro `movieId`. Inoltre non vi è più alcun requisito di continuità tra gli identificativi;

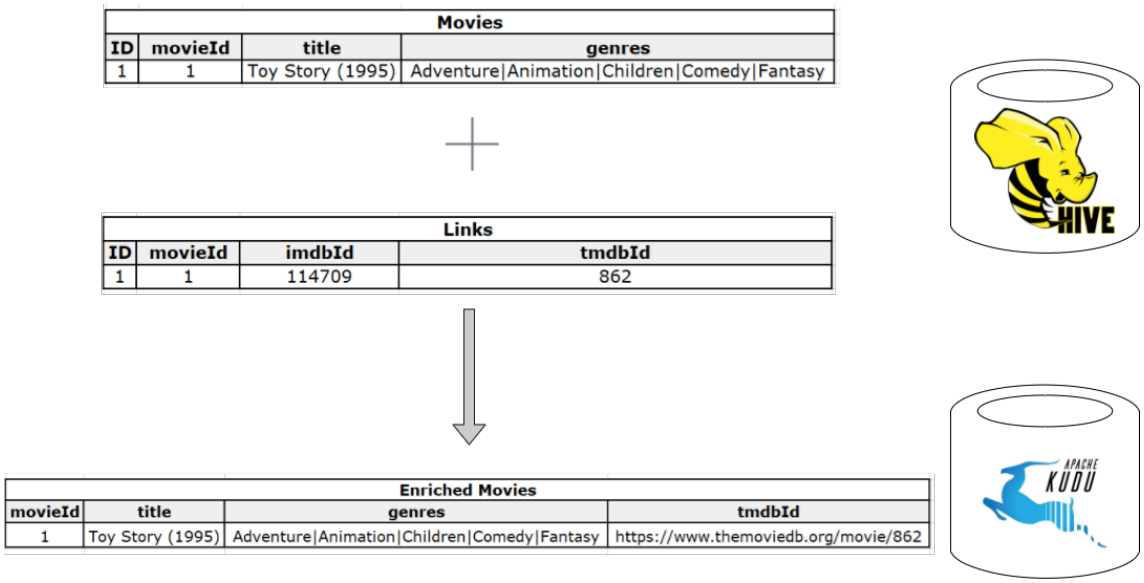


Figura 6.3: Processo Batch ETL

In Appendice B, listato B.2 è mostrato il codice Scala che unisce i due Dataframe.

Loading

Il Main (Appendice B listato B.3) si occupa della comunicazione con il Data Warehouse, estrae `Movies` e `Ratings`, applica la trasformazione tramite la classe `ETL` ed infine carica la nuova struttura dati sul Data Mart. Si noti come per il Data Mart, a differenza del Data Lake, sia necessaria l'esistenza preventiva delle tabelle Kudu. Il motivo è legato al diretto accesso di Kudu ai file su HDFS (bypassando Map Reduce): dovendo organizzare la distribuzione e replicazione dei file è necessario definire preventivamente il metodo di partizionamento e il numero di partizioni adottati.

Nel Listato B.4 è mostrato un esempio di creazione della tabella `ratings`.

Upsert Kudu

Kudu fornisce la possibilità nativa di effettuare inserimenti **idempotenti** tramite l'`UPSERT`. Questo metodo è l'unione dei processi di Insert e Update, un record verrà infatti memorizzato se e soltanto se non presente in tabella. La ricerca dell'elemento è effettuata sul singolo identificativo, combinazione efficiente se affiancata alla struttura colonnare di Kudu.

6.3.2 Real Time ETL

Per quanto riguarda `Ratings` e `Tags`, di natura strettamente dinamica e Real Time, è stata implementata una pipeline basata su Confluent Kafka per gestire lo stream continuo di informazioni.

Stream di Estrazione - Confluent Kafka

Il processo di estrazione real time da un Database era in passato piuttosto complesso, veniva utilizzata la tecnica del **Dual Writes** per la quale ogni inserimento scatenava, tramite trigger, l'invalidazione della cache e la successiva reindicizzazione

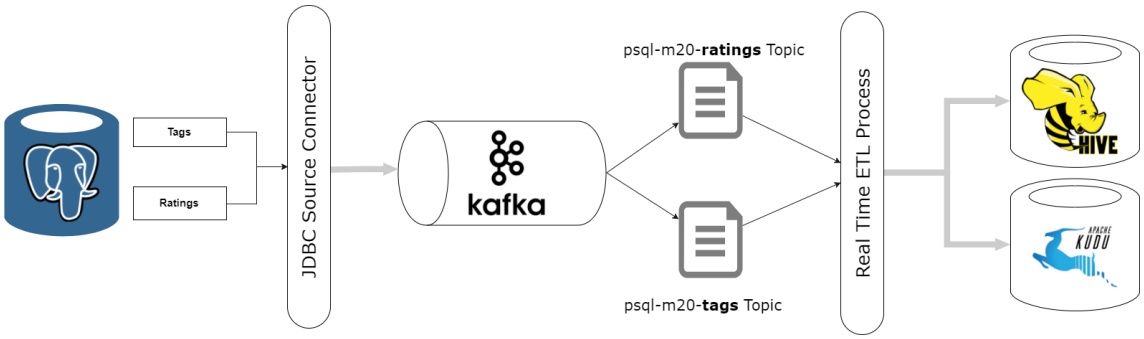


Figura 6.4: Pipeline di Ingestion Kafka

delle strutture dati a valle. La necessità di reindicizzare un datastore ad ogni inserimento precludeva un effettivo stream real-time di informazioni, inoltre erano usuali casi di disallineamento tra differenti strutture dati.

Kafka, tramite il **JDBC Source Connector**, permette di invertire la gestione dello stream real time. Invece di lasciare al database l'onere di notificare le strutture a valle, secondo pattern che si allontanano dalle effettive responsabilità di un RDMBS, è il connettore stesso che si preoccupa di verificare se nuovi elementi sono stati inseriti. Ogni tabella viene convertita in un **Topic** kafka, ogni elemento in un messaggio del Topic.

Il Connettore JDBC è pensato per eseguire periodicamente query SQL alla ricerca di elementi aggiunti e/o modificati nelle tabelle target; l'utente ha la possibilità di customizzare le tabelle monitorate e la query di estrazione, è comunque consigliabile l'utilizzo di query semplici per non rallentare lo stream real-time.

Il Flusso di informazioni così generato non è ancora del tutto real-time, si parla di **nealry real time (NRT)**, i dati infatti sono processati a blocchi di dimensione fissa, specifica in accordo con la reattività del connettore Sink implementato.

Configurazione della Pipeline Anche la pipeline stessa deve essere opportunamente configurata, Kafka permette di disaccoppiare totalmente sorgente e destinazione, creando una pipeline real-time indipendente dall'applicazione in cui viene utilizzata. Facendo riferimento al Listato B.6 in Appendice B, i parametri importanti sono:

- `bootstrap.server`: indica indirizzo e porta usata da Producers e Consumers per interagire con Kafka.
- `offset.storage.file.filename`: file sul nodo di installazione di Kafka che memorizza gli offset di lettura per ogni Topic. Grazie a questo file, Kafka riesce a fornire sempre l'ultimo messaggio non consumato anche in presenza di consumatori multipli sullo stesso Topic, appartenenti o meno allo stesso gruppo;
- `key.converer` e `value.converter`: rappresentano il formato in cui si presentano i dati ai Consumers;
- `internal.key.converter` e `internal.value.converter`: rappresentano il formato di salvataggio interno alla pipeline Kafka.

Transformation

Il processo di ETL real time elabora i `Ratings` e `Tags`, caricandoli su Data Lake e Data Mart. Questo doppio caricamento è pensato per mantenere l'uniformità all'interno del Data Warehouse.

Il processo RealTimeETL agisce come un daemon, ascolta continuamente i topics Kafka in attesa di nuove pubblicazioni. Le trasformazioni applicate dal processo si riducono alla rimozione dell'ID sequenziale e il rename del campo `timestamp` in `time`; quest'ultima trasformazione è dovuta all'utilizzo da parte di Kudu della parola "timestamp" come tipo di dato e quindi keyword inutilizzabile per i campi. Si

è preferito semplificare tali trasformazioni, al fine di minimizzare il tempo di elaborazione e introdurre il minor ritardo possibile nello stream di dati.

In Appendice B, Listato B.7 è mostrato il codice del connettore Sink responsabile della trasformazione introdotta.

6.3.3 Kafka Sink Connector - Spark Streaming

Il Kafka Consumer è implementato grazie alla libreria Apache Spark Streaming, già discussa nel Capitolo 2. Il Flusso continuo di messaggi Kafka è discretizzato e reso accessibile tramite il `DStream`. Il connettore applica la Trasformazione ai singoli RDD dello stream, salvandoli poi nel Data Warehouse.

Kafka DirectStream

Come si può notare dal Listato B.8 in Appendice B, Spark Streaming non si interfaccia direttamente con i topics Kafka. La creazione e configurazione dello Spark Streaming Context è gestita usando come punto di partenza la SparkSession creata da `Storage`. Per quanto riguarda l'interazione con i Topics, la libreria **Spark Kafka 0.8** utilizzata in questo elaborato, mette a disposizione due modalità [25]:

Approccio Receiver-based Questo tipo di approccio sfrutta un Sink Connector automaticamente istanziato dalla libreria. I dati ottenuti dal Receiver sono processati da uno dei nodi esecutori, sul quale uno Spark Job si occupa di estrarre e processare gli elementi dello stream. A causa di questa sequenza di memorizzazioni, non si ha la certezza di non perdere dati dello stream, la soluzione è quella di abilitare il Write-ahead Log, un sistema per cui Spark memorizza in maniera sincrona ogni singola operazione svolta; esaminando il WAL a ritroso è possibile recuperare i dati persi in

caso di necessità, ma questo equivale in pratica alla memorizzazione duplice di ogni messaggio estratto da Kafka.

La gestione dell'offset è lasciata alle API di alto livello del Receiver, che solitamente si appoggia su un gestore di risorse come Zookeeper. La soluzione è senza dubbio complessa e non del tutto affidabile, ho dunque deciso di sfruttare la seconda modalità, l'Approccio Diretto.

Approccio Diretto (Receiver-less) L'approccio Receiver-less, sfruttato in questo elaborato, consiste nel interfacciarsi in maniera diretta al Topic Kafka selezionato, senza sfruttare un Receiver Sink come tramite. Il Job Spark si occuperà di eseguire query periodiche sul Topic, tramite le stesse API di un qualsiasi consumatore, alla ricerca di nuovi messaggi non processati. I messaggi recepiti sono elaborati in maniera NRT dallo Spark Engine.

Tramite il metodo:

```
val messages = KafkaUtils.createDirectStream[String, String,
    StringDecoder, StringDecoder](ssc, kafkaParams, topics)
```

Listing 6.1: Creazione di DirectStream

si crea il **DirectStream** di DStream (Discretized) rappresentante un mapping 1-1 tra il topic in aggiornamento e la sequenza di RDD. Nella definizione dello stream è semplicemente necessario inserire la configurazione di Kafka e i **Decoder** necessari ad interpretare correttamente il formato di chiavi e valori. A Default è fornito lo **StringDecoder**, l'utilizzatore è comunque libero di implementare il proprio Decoder, o utilizzare un metodo handler, per gestire conversioni di tipo particolari.

Questo tipo di approccio porta numerosi vantaggi:

- **Parallelismo semplificato tra partizioni**: il DirectStream rappresenta un mapping 1-1 con il topic, questo include anche le partizioni in esso create.

La lettura diventa quindi automatizzata e trasparente all'utilizzatore, senza la necessità di creare stream differenti per diverse partizioni.

- **Zero-loss in maniera efficiente**: Non essendoci Receivers non sussiste il rischio di perdere il messagio tra ricevitore e consumatore, la gestione del recovery in caso di criticità è lasciata alla retention del database di Kafka;

- **Semantica Exactly-Once**: In questo approccio l'offset è gestito tramite API di basso livello direttamente dallo Spark Streaming Context, non vi è più quindi il rischio di disallineamento tra l'offset memorizzato da Kafka e quello su zookeeper. La stessa semantica deve essere mantenuta anche a valle del connettore, per questo motivo il salvataggio in Data Lake e Data Mart è idempotente (Upsert).

6.4 Analysis

Una volta che il Data Mart contiene dati relativi a `Movies` (arricchiti) e `Ratings` è possibile utilizzare i dati per gli scopi specifici del team di analisi. In questo progetto è stato realizzato un raccomandatore **collaborative-filtering** basato sull'algoritmo **Alternating Least Squares**.

Due sono i processi che permettono l'analisi dei dati, ancora una volta il primo è di tipo Batch, mentre il secondo ha una natura Real Time. Il processo MRSpark2 computa periodicamente un modello di raccomandazione, basato sui Ratings giunti fino a quel momento nel Data Mart, il servizio RealTimeMovieRec sfrutta tale modello per effettuare raccomandazioni Real Time tramite un'interfaccia web.

6.4.1 Sistema di Raccomandazione

Al giorno d'oggi, i sistemi di raccomandazione sono uno dei più diffusi Decision Support System, sono ad esempio utilizzati negli e-commerce per consigliare prodotti sulla base dei precedenti acquisti, o ancora sono utili per analizzare le preferenze di una fetta di mercato osservandone il comportamento in una vista più ampia.

Tralasciando le metodologie più banali presenti nella letteratura, come il consigliare il prodotto più venduto indipendentemente dall'individuo target, due sono i metodi più comuni: Content-based Filtering e Collaborative Filtering.

Content-based Filtering

Se in passato l'utente ha acquistato o espresso una preferenza per un determinato tipo di prodotto, allora è molto probabile che gradisca prodotti simili. Se, ad esempio, l'utente ha guardato "Avengers" e "Avengers 2", probabilmente apprezzerà anche "Guardiani della Galassia" dal momento che tali film hanno caratteristiche in comune. A livello pratico la decisione si converte in un clustering dei prodotti, ricondotti a vettori di features. Prodotti appartenenti allo stesso cluster saranno quindi simili e potenzialmente graditi dall'utente. Un approccio di questo tipo presuppone la conoscenza di tutte le caratteristiche dei prodotti in questione, dà inoltre i migliori risultati solo in contesti contenenti solo elementi dello stesso tipo o comunque classi omogenee di elementi.

Collaborative Filtering (CF)

Il Collaborative Filtering si basa solo e soltanto sullo storico delle azioni fatte da un gruppo di utenti su un insieme di prodotti. Le raccomandazioni sono generate sulla base delle similitudini di comportamento tra utenti. Ad esempio:

- L'utente A ha guardato "Avengers", "Avengers 2" e "Guardiani della Galassia";

- L'utente B ha guardato "My Little Pony" e "Stardust";
- L'utente C ha guardato "Avengers", "Avengers 2" e "Alla ricerca di Nemo".

Appare chiaro come gli utenti A e C abbiano gusti piuttosto simili, risulta quindi sensato consigliare "Alla ricerca di Nemo" ad A e "Guardiani della Galassia" a C, per quanto questi due film siano completamente differenti. Per l'utente B non è invece possibile generare alcun tipo di raccomandazione con buona certezza di successo.

Questo tipo di approccio, chiamato nello specifico **Collaborative Filtering User-User**, permette di generare raccomandazioni sulla base della similarità tra gli utenti. Due utenti che hanno fatto più o meno le stesse azioni avranno probabilmente gli stessi gusti.

In questo modo è possibile disaccoppiare totalmente la generazione delle raccomandazioni dalle caratteristiche dei prodotti raccomandati. Il problema principale, detto **Cold Start**, sta nella dipendenza da uno storico, risulta complesso infatti generare raccomandazioni per nuovi prodotti o per utenti che hanno eseguito nessuna o poche azioni.

Esprimere il Gradimento

L'espressione del gradimento di un utente nei confronti di un prodotto può avvenire in due modalità:

Rating Esplicito All'utente viene chiesto di esprimere esplicitamente un gradimento su un determinato prodotto, esso può essere una valutazione da 0 a 5 stelle (come in questo caso d'uso), o anche solo un *"mi piace"/"non mi piace"*;

Rating Implicito Si analizza il comportamento dell'utente per identificare pattern di gradimento. Il metodo è decisamente più complesso ma permette di estrarre

informazioni più nascoste e significative. Ad esempio, si potrebbe analizzare la sequenza di pagine visitate dell'utente in un sito di e-commerce, oppure il movimento del mouse sullo schermo.

6.4.2 Matrix Factorization (MF)

Il primo approccio implementato è stato un Raccomandatore basato sulla similarità tra gli utenti, calcolata secondo il metodo di Jaccard. A causa del grande numero di `Ratings` è apparso immediatamente che un modello di questo tipo non fosse adatto alla generazione di raccomandazioni real time, i tempi di risposta erano infatti dell'ordine dei minuti.

Allo scopo di gestire in maniera efficiente la grande quantità di elementi (20 milioni di valutazioni) del dataset Movielens, in modo da minimizzare i tempi di computazione del modello e di risposta a fronte di interrogazioni real time, ho fatto affidamento alla libreria **Apache Spark MLlib**, nello specifico l'algoritmo di **Matrix Factorization (MF)**, di tipo CF e ottimizzato con **Alternating Least Square (ALS)**.

Il problema di clusterizzare utenti e prodotti per definire raccomandazioni è, in realtà, un problema di **Co-Clustering**, la valutazione di un utente verso un prodotto dipende infatti sia dalle *Preferenze dell'Utente* che dalle *Caratteristiche del Prodotto*. Mentre nel Clustering tradizionale si cerca di classificare gli elementi (righe) di un insieme sulla base delle relative caratteristiche(colonne), nel Co-Clustering si suddividono in maniera simultanea sia le righe che le colonne della popolazione.

Il modello di valutazione viene rappresentato come una tabella in cui:

- Le righe sono gli utenti;
- Le colonne sono i film;
- Le celle interne rappresentano la valutazione che l'utente ha assegnato al film.

Ovviamente non tutti gli utenti avranno assegnato valutazioni ad ogni film, la Raccomandazione corrisponde infatti alla previsione del valore di un determinata cella della matrice sparsa. L'algoritmo MF permette di stimare questi valori, approssimando iterativamente il valore delle celle mancanti sulla base del modello d'insieme definito da utenti e prodotti.

Modello di Co-Clustering

Sia $R \in R^{m \times n}$ matrice di valutazioni applicate da m utenti su n prodotti. Lo scopo della Matrix Factorization è fattorizzare R in due sotto-matrici, $U \in R^{m \times k}$ e $P \in R^{n \times k}$, tali che $R \approx UxP$.

K rappresenta il *rank*, valore utilizzato per specificare la dimensione, cioè il numero di features, che descrivono le matrici di Utenti e Prodotti.

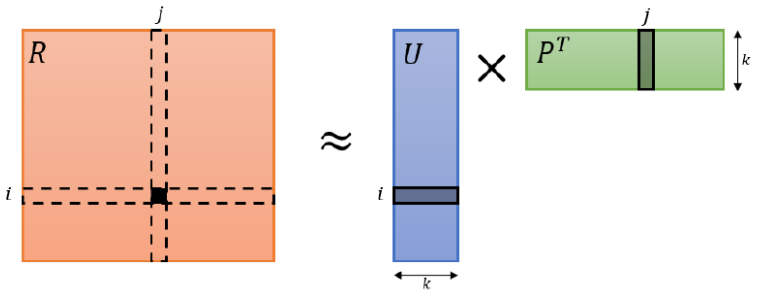


Figura 6.5: Modello di Matrix Factorization

In pratica, ogni cella $R_{i,j}$ viene fattorizzata nel prodotto scalare $u_i \cdot p_j$ con $u_i, p_j \in R^k$.

Ogni valutazione di R dipende direttamente da k features definite dalle matrici U e P. Dal momento che esistono, per natura, delle categorie di prodotti (film), ogni utente con le sue valutazioni, crea una connessione con una determinata "forza" nei confronti di una categoria. Il problema sta quindi nel definire le categorie di prodotti, conoscendo solo le valutazioni degli utenti sul singolo elemento.

Il Matrix Factorization approssima il valore della matrice originale R (di ogni sua cella, comprese quelle mancanti) conoscendo le matrici U e P, sulla base di una funzione di costo:

$$J = ||R - U \times P^T||_2 + \lambda(||U||_2 + ||P||_2)$$

Dove:

- Il primo Termine corrisponde al **Mean Square Error (MSE)**, cioè la distanza media tra i Ratings originali di R e quelli approssimati da $U \times P^T$;
- Il secondo Termine è di Regolarizzazione e serve ad evitare che il modello vada in overfitting su un determinato subset di elementi.

Si noti come k **e** λ siano due parametri fondamentali, sui quali effettuare il tunining, al fine di ottimizzare il modello generato.

Alternating Least Square (ALS)

Osservando il modello di MF, si nota come in realtà lo scopo sia l'apprendimento di due tipi di variabili, quelle appartenenti a U e quelle in P, correlate dal prodotto vettoriale. L'ALS mira ad approssimare R considerando una sola categoria di incognite per volta, in un processo iterativo che tende alla convergenza secondo una funzione di costo convessa.

Fissando, ad esempio, P e calcolando R solo sulla base dell'ottimizzazione data da U, il problema si riduce semplicemente a una **regressione lineare**, cioè la stima di un modello per un insieme di elementi identificabili nello spazio secondo un set di features.

La regressione lineare consiste nella stima dei parametri della funzione che meglio approssima i valori di un insieme di elementi, la stima è fatta minimizzando

l'errore quadratico medio tra i punti dell'insieme e quelli della funzione candidato. Modulando il parametro β è possibile trovare la funzione che meglio approssima i valori in esame.

$$||y - X\beta||_2$$

La soluzione è quindi data da $\beta = (X^T X)^{-1} X^T y$ (Ordinary Least Square). La soluzione ottenuta è unica e garantisce che β diminuisca o resti costante.

L'ALS consiste in un procedimento iterativo basato su due step:

- Nel primo Step si fissa P e si ottimizza la stima per U;
- Nel secondo Step si fissa U e si ottimizza la stima per P.

Grazie alla proprietà del OLS, la funzione di costo J può soltanto decrescere o rimanere invariata, di conseguenza il modello tende alla convergenza in un numero finito di iterazioni. In ogni caso, non vi sono garanzie che il risultato finale ottenuto sia un massimo globale.

Includendo il termine di regolarizzazione e scomponendo sulla base di quanto detto, è possibile rappresentare la funzione di costo come segue:

$$\forall u_i : J(u_i) = ||R_i - u_i \times P^T||_2 + \lambda \cdot ||u_i||_2$$
$$\forall p_j : J(p_j) = ||R_i - U \times p_j^T||_2 + \lambda \cdot ||p_j||_2$$

con:

$$u_i = (P^T \times P + \lambda I)^{-1} \times P^T \times R_i$$
$$p_j = (U^T \times U + \lambda I)^{-1} \times U^T \times R_j$$

Dal momento che gli u_i sono indipendenti, il calcolo delle features può essere altamente parallelizzato. **Apache Spark MLlib**, implementando l'algoritmo, sfrutta esattamente questa caratteristica per distribuire e parallelizzare in maniera efficiente il calcolo del modello.

Ratings Mancanti

I Ratings mancanti non possono essere considerati come valori nulli nel calcolo di U e P essendo essi stessi lo scopo della generazione del modello. Si introducono quindi dei pesi per selezionare su quali elementi effettuare il training.

$$w_{i,j} = \begin{Bmatrix} 1 & R_{i,j} & valorizzato \\ 0 & R_{i,j} & sconosciuto \end{Bmatrix}$$

$$u_i = (P^T \times w_i \times P + \lambda I)^{-1} \times P^T \times R_i$$
$$p_j = (U^T \times w_j \times U + \lambda I)^{-1} \times U^T \times R_j$$

6.4.3 BatchML

Il processo Batch MRSpark2 sfrutta l'algoritmo ALS, implementato nella libreria Apache Spark MLlib per computare un modello di raccomandazione sulla base dei `Ratings` giunti fino a quel momento.

Oltre a MLlib è presente in letteratura la libreria **Apache Mahout**, basata sul Map Reduce (alla base di Hadoop), che offre la possibilità di computazione distribuita e parallelizzata. Presenta però performances peggiori rispetto a MLlib, soprattutto all'aumentare del numero di valutazioni.

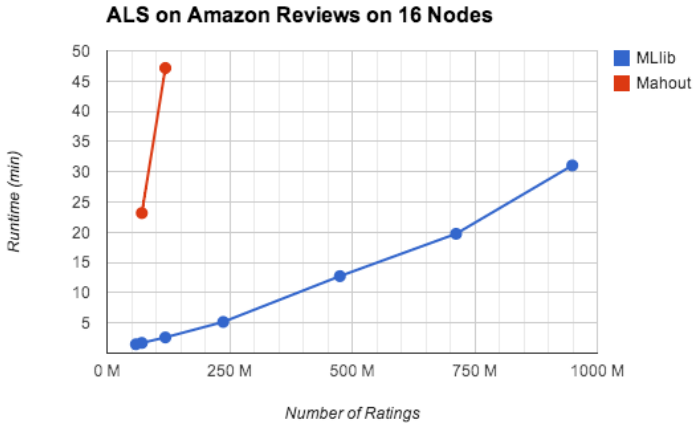


Figura 6.6: Confronto Mahout e MLlib

Il flusso di elaborazione è stato mantenuto il più possibile semplice e lineare: MRSpark2 scarica la versione più aggiornata dei Ratings dal Data Mart, li suddivide in train e test set (split 80% e 20%), computa il modello e lo valuta calcolando il Mean Square Error (MSE).

In Appendice B, Listato B.9 è riportato il codice relativo al raccomandatore.

Il sistema prende come parametri il **rank (k)**, il **fattore di regolarizzazione** (λ) e il **numero di iterazioni**, per i test sono stati utilizzati k = 10, 20 iterazioni e $\lambda = 0.1$, valori raccomandati dalla documentazione MLlib.

La classe `MovieRecommender` non è altro che un Wrapper del ALS di MLlib, pensato per offrire un metodo di accesso standardizzato rispetto agli altri componenti del sistema e alcuni metodi di supporto alla valutazione, salvataggio e caricamento del modello:

6.4.4 RealTimeMovieRec - Intefaccia di analisi Real Time

L'ultimo processo del sistema è pensato per interrogare il modello di raccomandazione, tramite una webapp permette di selezionare user e movie, ottenendo la previsione

sulla raccomandazione. Il sistema è implementato sulla base del framework **Scalatra** [23] grazie al quale è possibile costruire webapp Jetty-based in linguaggio Scala.

Scalatra permette di comporre la webapp come un insieme di endpoint automaticamente gestiti dal framework, le pagine visualizzate sono definibili tramite template perfettamente integrati nel codice Scala. Grazie alla combinazione di Scalatra con il framework SBT è possibile generare componenti software leggeri e facilmente trasportabili, a differenza di webapp implementate con altri framework come può essere Scala Play.

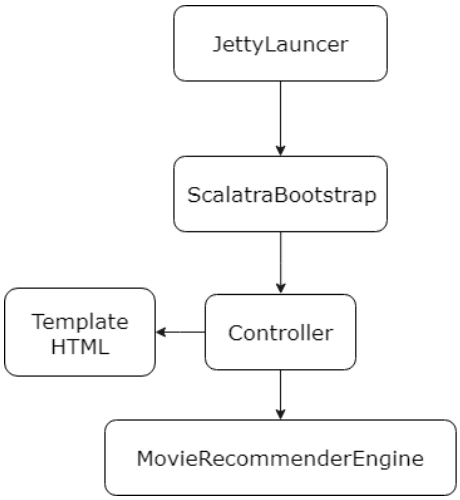


Figura 6.7: Architettura RealTimeMovieRec

I Figura 6.7 è mostrato uno schema molto semplificato dei componenti in gioco:

- **MovieRecommenderEngine** è il core del sistema, si occupa di caricare il modello salvato, decomprimerlo e renderne accessibili le funzionalità;
- Il **Controller** implementa gli entrypoit del sistema, ogni entrypoint è definito tramite una funzione anonima e permette di renderizzare qualsiasi cosa venga restituita in output, che sia una semplice stringa o una pagina HTML completa. Il **Template** è utilizzato come struttura per il rendering di pagine complesse, il Controller si occupa semplicemente di iniettare il contenuto;

- **ScalatraBootstrap** è il componente che gestisce i Controller, delle servlet, permettendone l'accesso tramite entrypoint;
- **JettyLauncher** è la classe Main che permette di eseguire il software fuori dal contesto di SBT, si occupa di attivare il Bootstrap ed eseguire la prima chiamata in modo da attivare il sistema.

In Appendice B, Listato B.10 è mostrato un esempio di endpoint.

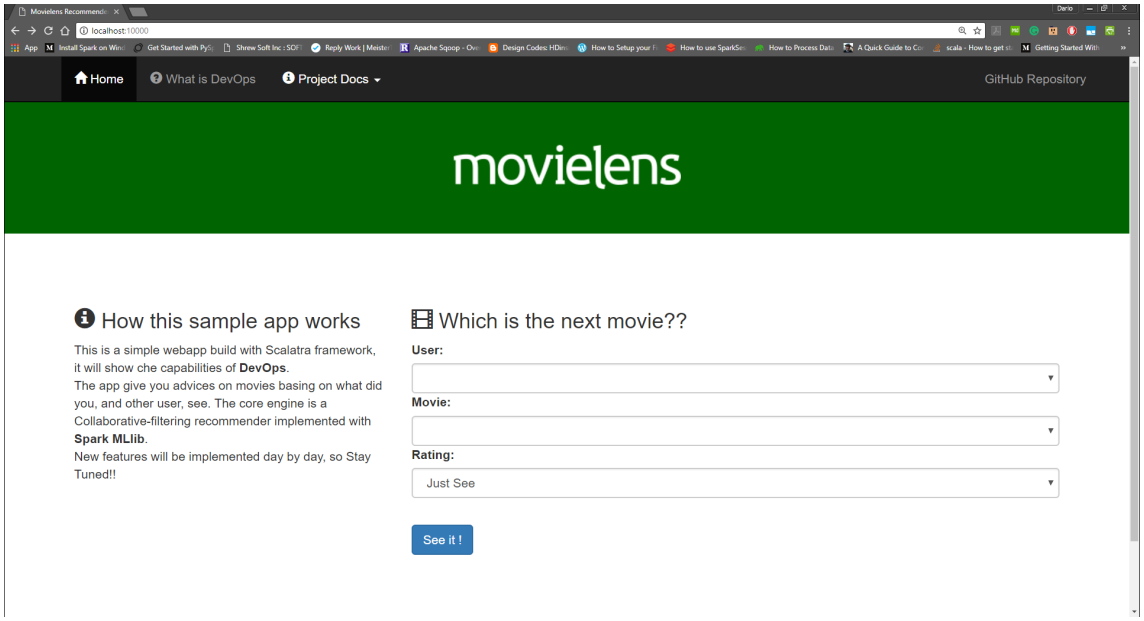


Figura 6.8: Form di Interrogazione

6.5 Storage

Al fine di standardizzare la connessione al Data Warehouse e facilitare il processo di sviluppo, si è deciso di racchiudere tutte le funzioni di interazione con i servizi Cloudera all'interno di una libreria chiamata DevOpsStorage.

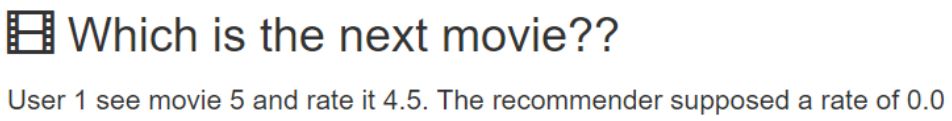
In particolare la libreria offre features per:

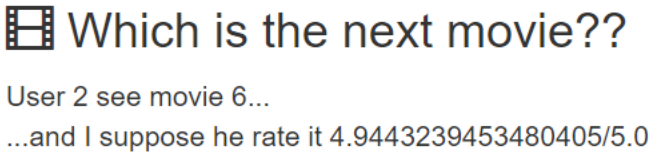
- **Init**:

  - SparkSession, con o senza supporto Hive
  - Connessione a Kudu
  - Connessione a HDFS

- **Kudu**: update, upsert, delete e read di row Kudu in formato DataFrame;
- **HDFS**: salvataggio e lettura di file di testo;
- **Hive**: lettura e scrittura di una tabella, esecuzione di una generica query HQL;
- **Compressione**: zip e unzip di file generici.

In appendice B.11 è mostrato il codice completo.





Capitolo 7

Caso d'Uso: DevOps

7.1 Introduzione

In questo capitolo verrà mostrata l'applicazione delle metodologie definite dal movimento **DevOps**, già analizzate nel Capitolo 3, applicandole ai quattro processi illustrati nel Capitolo precedente. A causa della loro differente natura, i quattro processi presentano problematiche di gestione differenti, che si riconducono a diversi approcci.

7.1.1 Processo Batch

Un processo di tipo Batch, come il BatchETL o MRSpark2, è pensato per elaborare grandi quantità di dati in maniera saltuaria. L'esecuzione del processo può essere:

- **On-demand**: Il processo viene eseguito solo quando necessario, ad esempio per estrarre uno storico, o all'aggiunta di nuovi movies per aggiornare il DWH (BatchETL);
- **Schedulata**: Si utilizzano tool come Cron oppure, Oozie, nel caso sia necessario gestire complessi workflow, per eseguire il processo in un determinato

istante a frequenza regolare. MRSpark2 è eseguito ogni giorno per aggiornare il raccomandatore sulla base della collezione più recente di Ratings.

Indipendentemente dal tipo di esecuzione, un processo Batch è pensato per massimizzare l'efficienza sulle grandi quantità di dati, con il vantaggio di poter vedere il dataset nella sua totalità (o almeno una fetta maggiore) e correlare tra loro elementi appartenenti ad istanti o addirittura epoche differenti. Il tempo di esecuzione, per quando da ottimizzare, è lasciato in secondo piano, dal momento che il processo è comunque eseguito saltuariamente.

Il deploy di una versione aggiornata e funzionante del servizio è, in linea di massima, sicuro; l'unica accortezza sta nel eseguire l'aggiornamento in un momento di inattività.

7.1.2 Processo Real Time

Un processo Real Time agisce come un daemon elaborando dati o offrendo un servizio sempre disponibile. Il processo è pensato per minimizzare il tempo di elaborazione o di risposta, al fine di non rallentare lo stream di dati, processando uno o comunque pochi elementi dello stream indipendentemente dagli altri.

Il deploy di una nuova versione deve essere eseguito con accortezza, è necessario infatti fermare il daemon prima di poter trasferire la nuova versione. Al termine del deploy è fondamentale assicurarsi che tutto sia in esecuzione, monitorando eventuali criticità dovute all'aggiornamento.

Nelle prossime sezioni verranno riprese le sette Best Practices, analizzate nel Capitolo 3, applicandole ai quattro processi e facendo riferimento all'esperienza personale nel percorso di tesi.

7.2 Configuration Management (CM)

Il Configuration Management è una pratica fondamentale per convertire l'esperienza sistemistica, fino ad ora riservata a un ristretto numero di elementi del team, in componenti software gestibili con gli stessi tools usati dai Devs nel loro lavoro.

Ansible è stato utilizzato per convertire le singole responsabilità, messe in luce in Figura 5.1, in un insieme di **Roles** fortemente modulare. E' stata definita un gerarchia di inclusione tra i Roles permettendo di gestire le singole dipendenze in maniera Top-Down a livello di granularità incrementale. In questo modo è possibile distribuire le responsabilità in modo modulare, scalando orizzontalmente in base alle specifiche necessità del caso.

Infine i roles, così come i Playbook delle specifiche istanze, sono definiti tramite il linguaggio YAML, risultano quindi auto-documentati e facilmente comprensibili da tutti i membri del team.

7.2.1 Roles

Sono definiti i Playbook per il Provisioning delle tre istanze che compongono il cluster oltre a quelli per effettuare il deploy sicuro. Di seguito è illustrata la gerarchia di ruoli che compongono i Playbooks.

Cloudera VM

```
cloudera-vm.yml
└── cloudera-vm
    ├── cloudera
    │   ├── cm-env-setup
    │   ├── cm-repo
    │   ├── oracle-java
    │   └── cm-install-path-b
    └── movielens-init
```

Il Provisioning dell'istanza **Cloudera VM** è quindi composto dall'installazione del CDH e del dataset Movielens, compresa la struttura tabellare nel database PostgreSQL e lo script di simulazione.

Essendo l'installazione e configurazione del CDH piuttosto complessa, soprattutto per quanto riguarda la distribuzione delle responsabilità tra i nodi del cluster, si è preferito lasciare tale compito al componente nativo del CDH, il Cloudera Manager. Il Playbook si occupa quindi del Provisioning del nodo master, una volta che il Cloudera Manager è attivo e funzionante la configurazione (compresa la selezione dei servizi Cloudera) procede nell'interfaccia web da esso fornita.

In Appendice C, Listato C.1 è mostrato il Role di installazione ed esecuzione del CDH.

DevOps Worker

```
worker.yml
├── devops-worker
│   ├── sbt
│   ├── spark
│   ├── confluent
│   ├── kafka-jdbc-connector
│   ├── folders
│   │   └── devops-folders
│   └── services
└── prometheus-node-exporter
```

Il Playbook **worker.yml** configura due ruoli:

- Il ruolo *devops-worker* configura a sua volta tutti i ruoli necessari all'esecuzione dei processi di elaborazione, includendo i tool di runtime, la struttura delle cartelle di deploy e i servizi di esecuzione.

- L'istanza sarà inoltre dotata della capacità di esporre metriche relative allo stato hardware, grazie al ruolo *prometheus-node-exporter*.

In Appendice C, Listato C.2 è mostrato il Role di Provisioning del DevOps Worker.

Big Brother

```
big-brother.yml
├── devops-worker
│   ├── sbt
│   ├── spark
│   ├── confluent
│   ├── kafka-jdbc-connector
│   ├── folders
│   └── services
├── ci-server
│   └── jenkins
├── orchestrator
├── prometheus
│   └── grafana
├── prometheus-node-exporter
└── prmetheus-pushgateway
```

L'istanza **big-brother**, avendo un ruolo di supporto, necessita della configurazione di molti servizi. Grazie alla modularità data da Ansible è ovviamente possibile separare i Roles in differenti nodi del cluster.
Si noti che il Big Brother possiede le stesse caratteristiche in termini di ruolo devops-worker dell'istanza DevOps Worker, il motivo è dovuto all'utilizzo della prima istanza come ambiente di Staging, di conseguenza è necessario che essa possa eseguire i processi esattamente come se fosse l'Edge Node del cluster.

Oltre a tali caratteristiche, l'istanza ha la capacità di gestire il Monitoring (grazie al set di ruoli `prometheus*`), il Provisioning e l'Orchestration (grazie al ruolo `orchestrator`).

In Appendice C, Listato C.3 è mostrato il Role di Provisioning del CI Server Jenkins.

7.3 Infrastructure As a Code (IaC)

La possibilità di configurare macchine in maniera modulare e role-based trova la sua massima applicazione se combinata alla capacità di creare istanze a piacimento, in base alle necessità dello specifico caso.

Terraform è il tool utilizzato per gestire l'orchestration del sistema. Tramite un linguaggio puramente descrittivo permette di definire le caratteristiche dell'istanza desiderata. Grazie alla clausola "`provisioning`" è inoltre possibile definire come configurare l'istanza appena creata.

Si è scelto di utilizzare Terraform, piuttosto che tool di Orchestration più specifici come CloudFormation di AWS, proprio per la sua generalità. Terraform può infatti essere usato con differenti provider, rendendolo uno strumento decisamente versatile.

Per il caso d'uso sono stati creati 3 file di configurazione in grado di istanziare le tre macchine esattamente come descritto nel Capitolo 5, sia in termini hardware che software. La configurazione è utilizzabile sia per creare istanze in base al carico di lavoro richiesto, scalando orizzontalmente il sistema, sia per implementare un'Immutable Infrastructure sostituendo istanze compromesse in maniera rapida ed automatica.

7.3.1 Configurazioni

Le tre configurazioni proposte sono molto simili tra loro, sarebbe infatti stato possibile introdurre delle variabili al loro interno in modo da parametrizzarle, si è però preferito mantenere la definizione chiara e completa al fine di creare un componente eseguibile out-of-the-box.

Ogni configurazione è composta da 3 sezioni:

- **Credenziali**: chiave SSH definita a livello di progetto, permette all'istanza

che la utilizza di controllare la creazione e distruzione delle istanze all'interno dello stesso progetto. Inoltre, garantisce l'accesso ad ogni macchina creata, permettendo quindi il Provisioning dopo la creazione.

- **Risorsa**: definisce il comportamento della configurazione. Ogni configurazione proposta permette di istanziare un singolo tipo di macchina, nulla toglie che un solo file gestisca più di un tipo di istanza tramite differenti blocchi `resource`.
- **Output**: definisce quello che Terraform restituisce in output al termine della configurazione. In questo caso viene generato un link che permette di connettersi via SSH alla la nuova istanza.

Tralasciando i comuni parametri di configurazione, ciò che effettivamente caratterizza una risorsa è dato da:

- **Nome**: deve essere univoco all'interno del progetto. Se specificato il parametro `count`, ad ogni nome verrà concatenato un ID incrementale di istanza.
- **Machine Type**: definisce il tipo di istanza in termini di CPU e RAM allocate.
- **Tags**: in maniera molto simile ai ruoli di Ansible, Google Cloud Platform permette di assegnare dei "ruoli" alle istanze. Nella pagina di configurazione è possibile definire delle caratteristiche specifiche per ciascuno di essi. In questo caso specifico i tags sono utilizzati per dichiarare l'insieme di porte del quale deve essere fatto forward in modo da esporre i servizi all'esterno dell'istanza. Ad esempio, il tag `bb` espone la porta 8080 corrispondente alla dashboard di Jenkins.

Provisioning Per quanto riguarda il Provisioning, Terraform dispone di numerosi metodi, dall'esecuzione di comandi bash sulle istanze create, fino all'installazione del client Chef e successiva applicazione delle configurazione. Disponendo di Ansible

per il Configuration Management si è preferito lasciare a tale tool dedicato il Provisioning delle nuove istanze, viene quindi eseguito un comando locale innescando la procedura.

In Appendice C, Listato C.4 è mostrata la configurazione dell'istanza Cloudera VM.

7.3.2 Esecuzione

Riguardo l'applicazione delle configurazione, Terraform offre due possibilità:

Apply

```
terraform apply [dir-or-plan]
```

Listing 7.1: Trraform Apply

Il comando `apply` applica la configurazione definita nei file `.tf` indicati. Con questa modalità Terraform scansiona il provider target in modo da definire il set di azioni necessarie al raggiungimento dello stato definito dalla configurazione (dichiarativa). A default, l'applicazione richiede l'esplicita approvazione da tastiera per essere eseguita, è possibile automatizzare il processo inserendo l'opzione `-auto-approve` alla stringa di esecuzione.

Plan

```
terraform plan [name]
```

Listing 7.2: Trraform Apply

Il comando `plan` permette di definire in maniera statica il set di azioni necessarie per applicare una configurazione. In questo modo è possibile definire un piano statico

ed eseguibile che racchiude l'Orchestration e Provisioning di un determinato cluster. Lo stato del sistema non verrà quindi definito dinamicamente dal metodo apply, rendendo deterministica l'esecuzione.

7.4 Continuous Integration (CI)

Con la Continuous Integration si vuole minimizzare lo sforzo necessario ad integrare le varie features di un prodotto software. Per implementare la tecnica è necessario, nell'ordine:

1. **Features Indipendenti**: Modellare il software come un insieme di features minime, tra di loro il più possibile indipendenti, o almeno strutturate in modo incrementale. Le features saranno quindi nativamente suddivise in un gruppo Core, necessario al funzionamento minimo del sistema, e un gruppo incrementale che aggiunge nuove funzionalità al progredire dello sviluppo.

2. **Test Automatizzati**: Definire un sistema di Test automatizzato che copra in maniera esaustiva la semantica delle features sulla base delle User Story del cliente.

3. **VCS**: Dotarsi di un Version Control System, hostato internamente o esternamente in base alle necessità aziendali.

4. **CI Server**: Inserire nel sistema un'istanza in grado di svolgere il ruolo di Continuous Inegration Server e dotata di tutto il necessario a garantire esecuzione autonoma e monitoring esaustivo.

7.4.1 Struttura del Progetto

Per il PoC in analisi è necessario implementare quattro processi caratterizzati da nature fortemente differenti, ciascuna delle quali presenta problematiche di gestione a

se stanti. Presupponendo un'interfaccia definita e standardizzata, tramite la quale i processi possono interagire tra di loro, è possibile rendere ogni processo indipendente dagli altri. Risulta quindi forzato considerare i 4 processi come effettivamente parte di un unico prodotto software, di conseguenza si è preferito strutturare i requisiti del PoC sotto forma di 4 Progetti.

Questa scelta porta numerosi vantaggi nell'ottica di provare l'applicazione delle metodologie DevOps, risulta infatti possibile implementare soluzioni di gestione perfettamente modellate per il determinato use case, mettendo in luce problematiche e scelte progettuali specifiche. Per ogni progetto è possibile configurare le dipendenze, implementare test e definire flussi di CI specifici, senza timore di intaccare l'esecuzione degli altri componenti.

Al tempo stesso, questa scelta progettuale ha portato alla necessità (analizzata nella sezione relativa al Continuous Testing) di testare il funzionamento incrociato dei quattro processi esattamente come se fossero attivi in produzione.

7.4.2 SBT - gestione delle dipendenze

Ogni processo richiede componenti esterni specifici per garantire la sua esecuzione, ad esempio il RealTimeETL avrà bisogno di Apache Spark Streaming, mentre MR-Spark2 necessita di Apache Spark MLlib. La gestione di tali dipendenze è lasciata al framework **Scala Build Tool (SBT)**.

Grazie a SBT è possibile specificare la lista delle dipendenze di un progetto, il framework si occuperà del download e della configurazione delle stesse in fase di compilazione. SBT offre inoltre numerosi strumenti CLI come la compilazione, il packaging, l'esecuzione e la creazione di un ambiente console identico a quello di esecuzione del progetto.

Il maggior vantaggio di SBT è dunque la possibilità di ricreare l'esatto ambiente di esecuzione richiesto, indipendentemente dalla piattaforma sottostante. Inoltre, grazie al supporto di plug-in ed estensioni, è possibile aggiungere strumenti al framework come la gestione di server jetty, esecuzione di test, creazione di jar standalone, e molto altro.

Ogni progetto deve definire un file `build.sbt`, scritto in linguaggio Scala, contenente l'insieme delle dipendenze necessarie al corretto funzionamento del sistema.

Il Listato C.5, in Appendice C, mostra il `build.sbt` del progetto BatchETL. Come si può osservare, il file è strutturato in diverse sezioni:

- Nome, Versione del package e di Scala sono utili ad identificare il componente all'interno del flusso di sviluppo. E' buona norma modificare il numero di versione ad ogni nuova feature aggiunta, in modo da tener traccia facilmente del processo di sviluppo.

- I **resolvers** servono a SBT per sapere da quale repository scaricare le dipendenze, in questo caso il distributore scelto è Maven Central.

- Quindi vengono le dipendenze specifiche, si è preferito strutturarle in tre gruppi per rendere la configurazione più mantenibile. Nell'ordine sono definite:

  - **Dipendenze Spark**: definiscono l'insieme di librerie strettamente necessarie all'esecuzione del componente. Tramite la keyword `provided` è possibile indicare ad SBT che la libreria è già presente nel sistema, il linking verrà quindi fatto con il jar locale. Questa modalità è utile quando il sistema destinatario contiene già tutte le dipendenze necessarie all'esecuzione, volendo però comporre dei componenti completamente standalone, si è preferito lasciare ad SBT tutta la gestione.

  - **Dipendenze di Test**: la keyword `test` identifica una libreria accessibile solo in ambiente di testing, definito in automatico nel template dei progetti SBT. Tali librerie sono accessibili tramite CLI per l'esecuzione dei test, non verranno però inserite nella versione compilata.
  - **Dipendenze di Monitoring**.

- Infine si trovano le **Opzioni**, utili a modificare il comportamento default di SBT.

7.4.3 Version Control System

Tra i vari VCS presenti sul mercato si è scelto di utilizzare **GitHub** nella sua versione hostata esternamente. Dopo aver testato altri sistemi, quali BitBucket e GitLab, si è preferito GitHub per la sua grande diffusione (prossima alla standardizzazione) nel mondo open source e per la comoda integrazione con il CI Server Jenkins.

La piattaforma, oltre al gestire il versioning dei quattro progetti, offre numerosi servizi accessori come la documentazione sotto forma di Wiki e il tracking delle issue per progetto.

Su GitHub sono inoltre mantenuti e versionati altri progetti di supporto al PoC.

7.4.4 Jenkins + Blue Ocean - Continuous Integration Server

L'istanza Big Brother ospita il CI Sever Jenkins il quale, grazie al supporto di Blue Ocean, permette di strutturare pipeline customizzate per ogni Branch di un progetto Multibranch. Ciascun progetto avrà quindi il proprio **Jenkinsfile** con la definizione della pipeline ottimale per le sue necessità.

Il vantaggio di poter definire la pipeline tramite un file disaccoppiato dal server stesso non è trascurabile: oltre alla comodità di poter modellare la pipeline in un linguaggio descrittivo quale il groovy, essa stessa può essere trattata come un

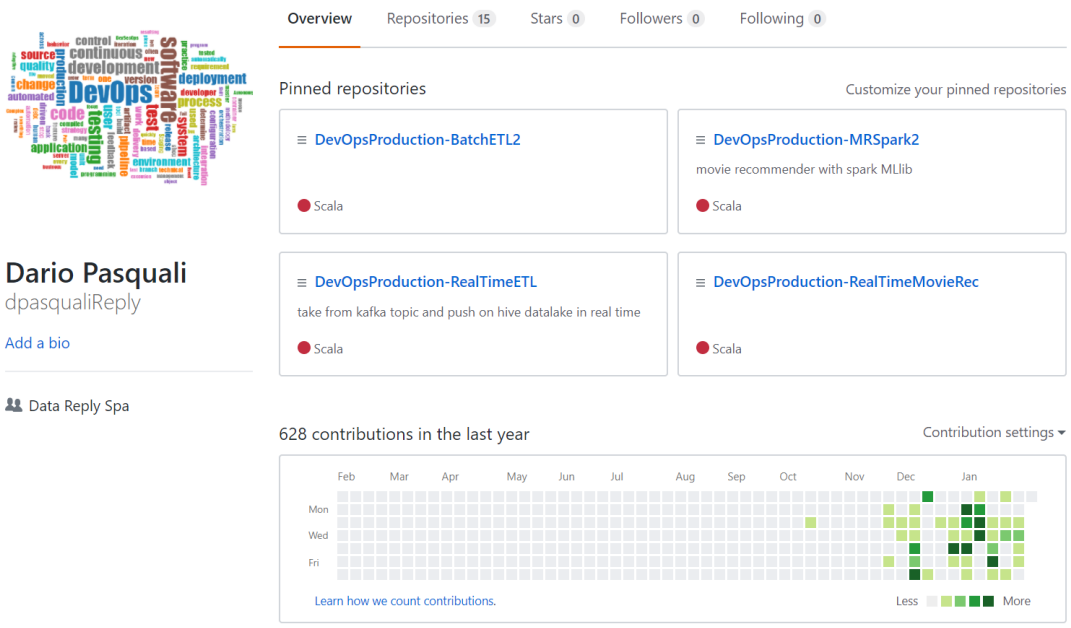


Figura 7.1: Homepage Github

componente software, può essere versionata, testata e condivisa all'interno del team con la certezza che chiunque esegua quella pipeline sul CI server otterrà la stessa sequenza di stage. Inoltre, nel momento in cui viene creato un nuovo Feature Branch partendo dal Mater Branch, anche la pipeline stessa verrà replicata, dando la possibilità di creare una version ad'hoc per la nuova feature, senza il timore di intaccare la versione originale. Ad esempio, se la pipeline Master termina con il deploy in produzione, la versione Feature terminerà con la pull request automatica verso GitHub in modo da richiedere l'integrazione. Nel momento in cui viene approvato il Merge (automaticamente o tramite la Pull Request), Jenkins si occupa di attivare la pipeline Master.

Jenkinsfile

Sebbene i quattro processi abbiano nature decisamente differenti, si è cercato di standardizzare le quattro pipeline di CI (successivamente di CD) in modo da rendere unico ed uniforme il processo di sviluppo. La standardizzazione del processo all'interno del team permette di uniformare la nomenclatura dando una semantica più significativa agli errori. Il Listato C.6, in Appendice C, mostra la pipeline di CI del BatchETL, le altre pipeline sono strutturalmente identiche.

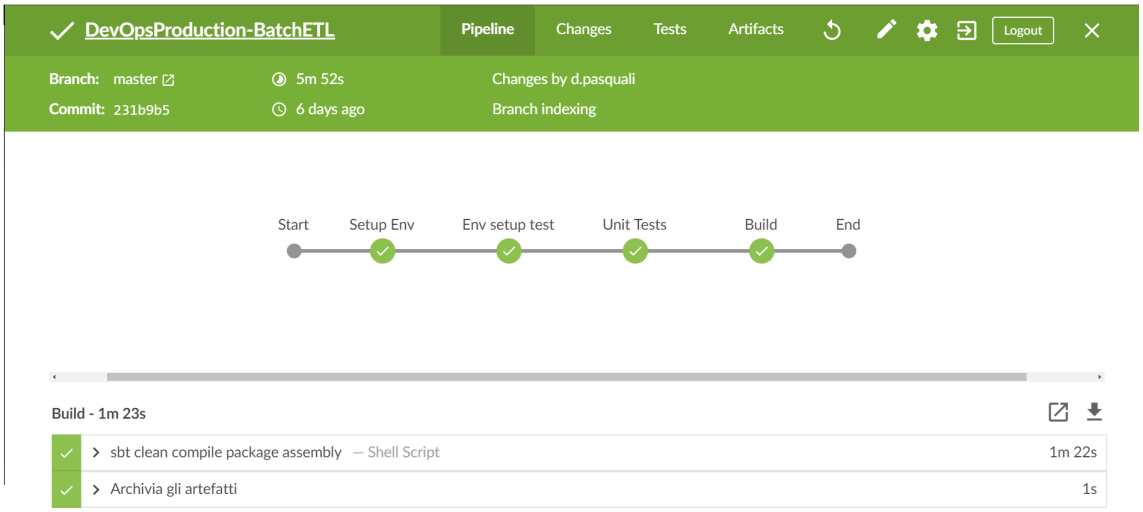


Figura 7.2: Blue Ocean Pipeline di Continuous Integration

Il flusso della pipeline risulta lineare:

Setup Env

Inizialmente questa fase era utilizzata per configurare l'ambiente di esecuzione, ad esempio venivano installati SBT e Spark. Con l'introduzione del Configuration Management la fase è diventata superflua, si è scelto di mantenerla per garantire l'uniformità del processo e nel caso fosse necessario introdurre un agente differente da `any`.

Env Setup Test

Si verifica quindi che l'ambiente sia correttamente configurato, partendo dal presupposto che SBT si occupa di gestire le specifiche, è semplicemente necessario verificare che il framework sia correttamente installato e raggiungibile.

Unit Tests

In questa fase vengono eseguiti degli Unit e Smoke Test al fine di verificare che le specifiche del cliente siano rispettate.

E' fondamentale che gli Unit Test coprano il più possibile il codice scritto e che possano essere eseguiti in tempi brevi (meno di 10 minuti secondo l'eXtreme Programming); questi stessi test devono inoltre essere eseguiti dallo sviluppatore prima di effettuare il commit in modo da evitare a priori un'esecuzione inutile e potenzialmente dannosa della pipeline.

Si noti come, grazie all'astrazione fornita da SBT, l'esecuzione di tutta la batteria di test, complessa a piacimento, si riduca ad un semplice **`sbt test`**, con l'aggiunta del calcolo della Coverage.

Al termine dell'esecuzione, SBT genera una serie di report, Jenkins si occupa di gestirne il tracking offrendo uno strumento molto utile per monitorare l'efficienza del processo e analizzare lo stacktrace in caso di bug.

Build

Al termine dei test la nuova versione può essere compilata ed archiviata, pronta per il deploy o l'integrazione con il Master Branch. La compilazione è eseguita grazie a **`sbt-assembly`** [21], questo plugin SBT permette di creare un cosiddetto *fat JAR* contenente tutte le dipendenze necessarie all'esecuzione del componente software. Il pacchetto risultante, seppur di una dimensione non trascurabile, rende di fatto il componente indipendente dall'ambiente di deploy.

SBT assembly esegue a default tutti i test che trova all'interno del progetto, dal momento che vi è già un'esplicita fase di Unit Test nella pipeline, si è deciso di disabilitare questa funzionalità tramite le opzioni nel file `build.sbt`. La strategia di merge, anch'essa definita in `build.sbt`, sta ad indicare come SBT deve gestire le dipendenze scaricate rispetto a quelle trovate nel sistema nel quale si esegue la compilazione. In questo caso si è deciso di dare priorità alle versioni scaricate, sicuramente più aggiornate di quelle locali.

Inserire la compilazione come parte integrante della pipeline di CI è fondamentale per creare pacchetti standard, monitorabili e compilati in un ambiente identico a quello di produzione.

Post Actions

Al termine della pipeline è necessario notificare il successo o fallimento in modo che il team si comporti di conseguenza. Il blocco esterno, marcato con `post`, permette di inviare un messaggio sul canale Slack utilizzato dal team per le comunicazioni relative al progetto.

In caso di successo è notificata la possibilità di effettuare una pull request in maniera sicura, in caso contrario si raccomanda di analizzare dashboard, stacktrace e codice al fine di risolvere i bug nel minor tempo possibile (idealmente minore di 10 minuti). Trovare bug in questa fase è decisamente significativo, dal momento che quegli stessi test erano funzionanti nell'ambiente di sviluppo ma fallimentari in quello di simil-produzione.

Figura 7.3: Notifica Slack di build completata

7.5 Continuous Testing

Nell'ottica del Continuous Deployment i test, parte integrante della pipeline, sono l'unico baluardo a difesa dal rilascio in produzione di software non funzionante. Combinando un metodo di Sviluppo Test-Driven con il concetto di Coverage del codice, è possibile incrementare ulteriormente il valore apportato da un Test, convertendolo in un efficace indicatore di rischio ed avanzamento.

7.5.1 Unit Tests

ScalaTest

ScalaTest [22] è il tool scelto per la gestione dei Test in maniera TDD. ScalaTest è un framework di testing, distribuito sotto forma di plugin SBT, che permette di strutturare le batterie di test in maniera dichiarativa, molto simile alla descrizione dei casi d'uso dell'utente. ScalaTest permette inoltre di eseguire i Test tramite SBT, restituendo un immediato feedback human-readable e machine-parsable.

I quattro processi sono stati implementati seguendo le pratiche del TDD:

- Ogni singola feature è convertita in specifiche del tipo `"X should/must behave like Y"`, definendo la semantica dei componenti;

- Tali test sono inizialmente pensati per fallire;
- La soluzione è quindi implementata allo scopo di far funzionare i test.
- Il processo viene ripetuto iterativamente all'inserimento di ogni nuova feature.

Il Listato C.7 in Appendice C mostra la specifica Unit Test di MRSpark2, responsabile dell'elaborazione del modello di raccomandazione. Come si può notare, è possibile strutturare i test a blocchi, definendo l'insieme di caratteristiche che un determinato componente del sistema deve avere, con un approccio descrittivo e human-readable.

Inizialmente tutti i test saranno marcati come **`pending`**, indicando che la feature non è ancora stata implementata, mano a mano che lo sviluppo procede, nuovi test saranno attivati e nuove soluzioni implementate, fornendo una metrica dello stato di avanzamento del processo di sviluppo.

L'esecuzione della batteria di test in Listato C.7, genera un lungo log, mostrato dalla dashboard Blue Ocean, che termina con un riassunto basato sulla specifica descrittiva dei test (mostrato in figura 7.4).

```
[info] MRSpec:
[info] The movie recommender
[info] - must be instantiated with given parameters
[info] - must compute a valid model given input ratings
[info] The computed model
[info] - should estimate a new entries with good precision
[info] - should can be saved in zip format and retrieved
[info] Run completed in 23 seconds, 461 milliseconds.
[info] Total number of tests run: 4
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 4, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
```

Figura 7.4: Risultato del test su MRSpark2

Gli Unit Test sono strutturati per testare soltanto i requisiti strettamente funzionali, come la trasformazione apportata dai processi ETL o la validità del modello di Machine Learning. Il Test del funzionamento del sistema nella sua interezza è demandato a un insieme di **Integration Test** esterni, superiori ai quattro processi ed eseguiti in un ambiente identico a quello di produzione.

7.5.2 Integration Test

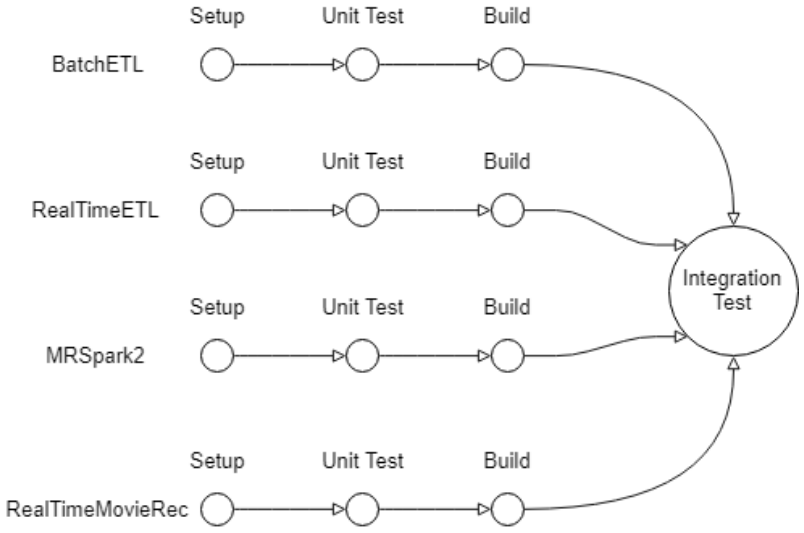


Figura 7.5: Flusso di Integration Test

Staging Environment

Si è quindi definito un ambiente di esecuzione che rispecchiasse in tutto e per tutto quello di produzione, sia in termini di configurazione che di sequenza di azioni eseguite. In questo ambiente vengono fatti interagire i quattro processi, verificando che tutto funzioni in maniera corretta. Al fine di supportare questo tipo di test, è stato necessario creare delle strutture dati accessorie in modo da standardizzare non solo l'ambiente ma anche i dati analizzati, sempre nell'ottica di minimizzare il tempo di esecuzione.

Database PostgreSQL Mentre si è assunto che `Movies` e `Links` siano tendenzialmente statici, il numero di `Tags` e `Ratings` aumenta mano che i dati del backend vengono ingeriti nel Data Warehouse. Al fine di limitare i tempi di esecuzione, e permettere il confronto tra varie esecuzioni della pipeline, sono state definite due **View**, `ratings_test` e `tags_test`, all'interno del Database PostgreSQL che catturano solo i primi 10000 `Ratings` e `Tags` collezionati.

Kafka Connector Un opportuno connettore Kafka, modifica di quello presentato nel Listato B.5, estrae i dati da tali viste e popola i relativi Topics

Test Data Lake All'interno del Data Warehouse si è predisposto un Data Lake secondario, chiamato `datalake_test`, utilizzato per salvare i dati del Integration Test. Le strutture dati Hive vengono create e distrutte ad ogni esecuzione in modo da eseguire ogni volta con il sistema "vergine".

Test Data Mart Allo stesso modo è stato creato un Data Mart Kudu, `datamart_test`, contenente le quattro strutture dati utilizzate dai processi. A differenza delle tabelle Hive è necessario che le strutture Kudu siano già presenti per poter essere scritte. Si sfrutta quindi la rapidità di accesso e la possibilità di Upsert offerta da kudu, per troncare le strutture ed invalidare i metadata ad ogni esecuzione.

IntegrationStagingProject

All'interno del cluster deve essere presente un'istanza con le stesse caratteristiche di quella/e utilizzata per il deploy in produzione. In questo PoC si è scelto di utilizzare il Big Brother stesso, dove esegue anche la pipeline di CI, al suo interno è stato collocato un opportuno progetto responsabile sia della simulazione idempotente e deterministica del flusso di esecuzione, sia dei test necessari alla verifica del funzionamento.

Esecuzione Simulata

Il progetto `IntegrationStagingProject`, presente su GitHub, implementa tale flusso di simulazione. Il Listato C.8 in Appendice C mostra uno pseudocodice dell'elaborazione.

Struttura del Progetto

Il flusso di esecuzione, seppur implementato in un unico blocco, sfrutta il concetto delle **Toggle Features** per introdurre modularità e facilitare il lavoro dello sviluppatore in caso di test rapidi. Ogni gruppo di test può essere abilitato in base alla necessità agendo su un file esterno di configurazione. Allo stesso modo, anche i singoli processi sono configurabili tramite file esterni, il progetto esegue la `spark-submit` fornendo dei file di configurazione appositamente pensati per l'ambiente di Staging.

Staging Deploy

All'interno del progetto è presente una cartella `lib`, SBT a default integra nel classpath tutti i jar inseriti in tale cartella. La pipeline di CD si occuperà quindi di collocare nella directory la nuova versione appena testa e compilata, lasciando inalterati i fat JAR relativi agli altri processi.

7.5.3 Estendere la Pipeline

La pipeline di Continuous Integration deve quindi essere estesa per includere i Test di Integrazione, si aggiungono due Stages:

- Il primo simula il deploy, collocando il package standalone, appena assemblato, nella cartella `lib` dell'ambiente di Staging;
- Il secondo si occupa di eseguire la batteria di Integration Test, ancora una volta sfruttando le potenzialità di `SBT`.

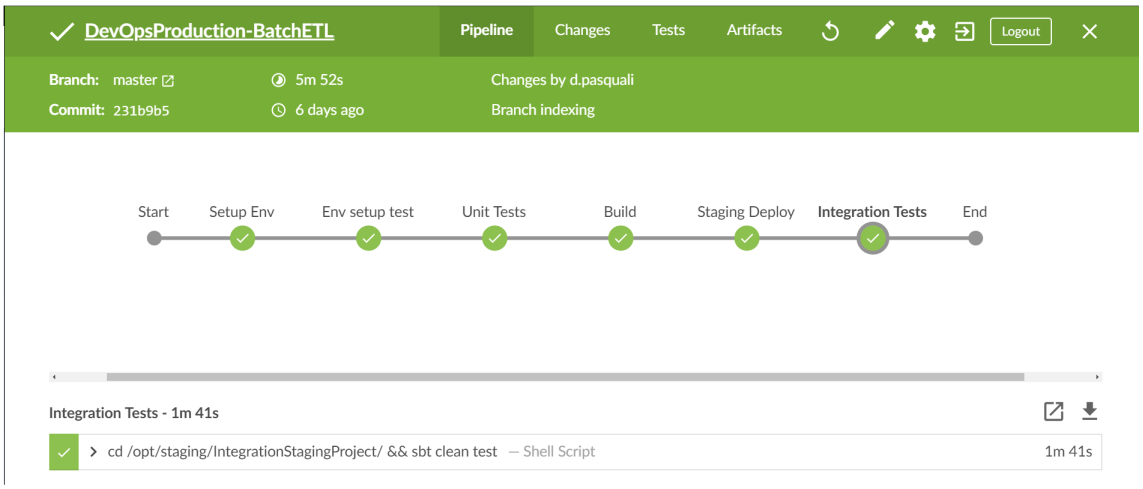


Figura 7.6: Pipeline di Continuous Integration estesa

7.6 Continuous Delivery

Con il Continuous Delivery si vuole estendere la pipeline di Continuous Integration, dando la possibilità di rilasciare in produzione ogni modifica integrata nel Master Branch. Deve essere possibile rilasciare, in maniera sicura e certificata, ogni nuova Feature integrata tramite la pipeline, in questo modo la scelta di effettuare una Release diventa unicamente dipendente dalla strategia di Business e non dal funzionamento del sistema.

Il cambiamento è più culturale che tecnico, nel momento in cui la pipeline è composta da una serie di test efficaci e in grado di garantire una sufficiente Coverage, la scelta di eseguire un rilascio si riconduce alla semplice pressione di un bottone.

Nell'ottica di disaccoppiare **Deploy** e **Release**, entrambe le azioni sono autonomamente controllabili da scelte manuali di business.

7.6.1 Deploy Manuale

La Pipeline di CI è completata, diventando a tutti gli effetti una pipeline di Continuous Deployment, con l'integrazione di due nuovi Stages, come mostrato in Appendice C listato C.10.

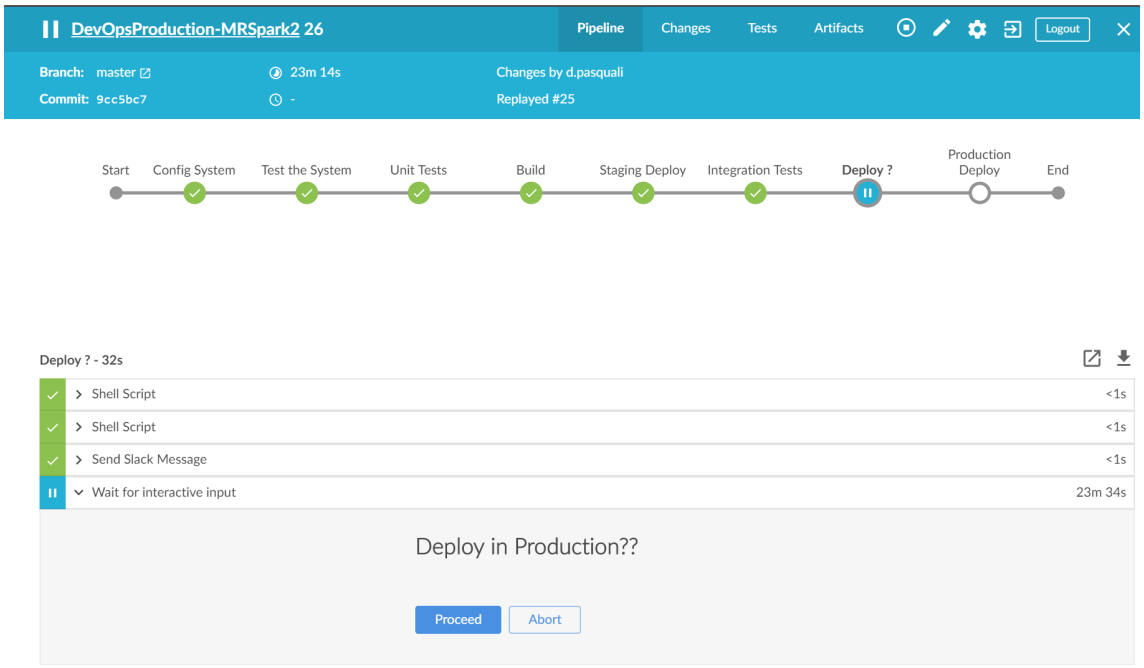


Figura 7.7: Richiesta di Deploy manuale via Jenkins

Deploy ?

Lo stage di **Deploy ?** richiede all'utente un'interazione manuale per decidere se effettuare o meno il Deploy in produzione del nuovo package appena assemblato. Dal momento che il package ha superato con successo la suite di test, del quale il team si fida ciecamente, la scelta di effettuare o meno il Deploy è solo e soltanto strategica. La richiesta di interazione manuale viene anche trasmessa all'ambiente di comunicazione Slack in modo che l'intero team possa decidere quale azione intraprendere.

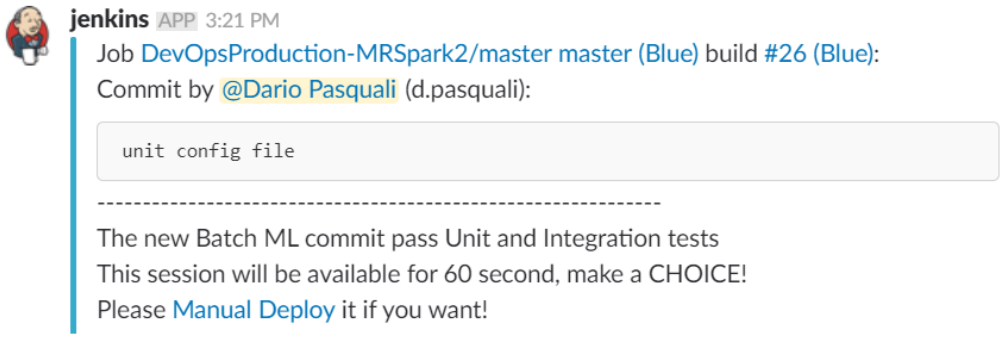


Figura 7.8: Richiesta di interazione manuale via Slack

Come indicato nel messaggio Slack in Figura 7.8, la pipeline è configurata per mantenere aperta la possibilità di deploy manuale per soli 60 secondi. Il motivo è legato a due fattori:

- Gli artefatti compilati sono in ogni caso archiviati e mantenuti per il periodo di retain definito nella configurazione di Jenkins, di conseguenza nel momento in cui il team si fida ciecamente della pipeline, il deploy manuale è eseguibile in qualsiasi momento;

- Jenkins utilizza, per la sua elaborazione, il concetto di "Agente esecutore" per aprire la possibilità a build parallele. In pratica un "agente" non è altro che un core dell'istanza che ospita Jenkins, l'attesa di una interazione utente significa quindi bloccare in maniera sincrona la pipeline, di fatto perdendo un grado di parallelismo.

  Di conseguenza si è scelto di inserire un timeout di attesa al termine del quale il processo fallisce automaticamente, notificando però la corretta archiviazione della nuova versione.

In Figura 7.9 è mostrato il messaggio postato su Slack in caso di consenso al deploy manuale.

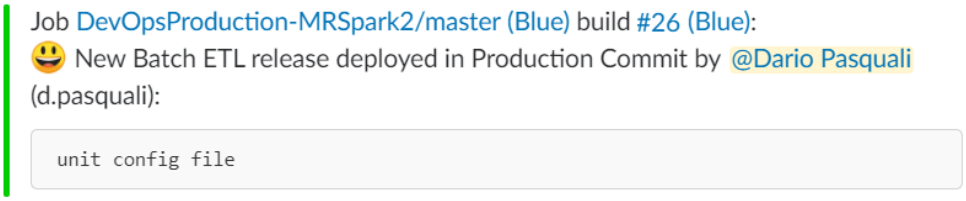


Figura 7.9: Messaggio di avvenuto deploy in produzione

7.6.2 Release Manuale - Toggle Features

Nel momento in cui la nuova versione è stata rilasciata in produzione, non è detto che sia il momento migliore per effettuare il Release di una determinata Features, per permettere cioè agli utenti di utilizzarla nella sua interezza. Le **Toggle Features** possono essere utilizzate per garantire tale possibilità.

Esattamente come mostrato nel `IntegrationStagingProject`, le features del componente vengono racchiuse da strutture di check basate su toggle esterni al componente, in questo modo è possibile rendere accessibile una funzionalità senza la necessità di ricompilare il sistema. Si apre inoltre la possibilità ad effettuare Release mirati, Canary Distribution e Dark Launching abilitando all'utilizzo della feature solo un ristretto numero di utenti appositamente scelto (consciamente o inconsciamente) per la sua valutazione.

In questo PoC son state usate le Toggle Features nella gestione delle versioni del frontend del RealTimeMovieRec, la prima versione del sistema era molto più grezza riguardo l'interazione con il raccomandatore. La webapp presentava movies e users come due semplici liste richiedendo all'utente di interagire introducendo manualmente la combinazione desiderata direttamente nella barra degli indirizzi del browser.

La scelta di mostrare il frontend in Figura 7.10 piuttosto che quello in 6.8 è gestita da un parametro esterno. Le due versioni possono convivere se necessario, ma l'ideale

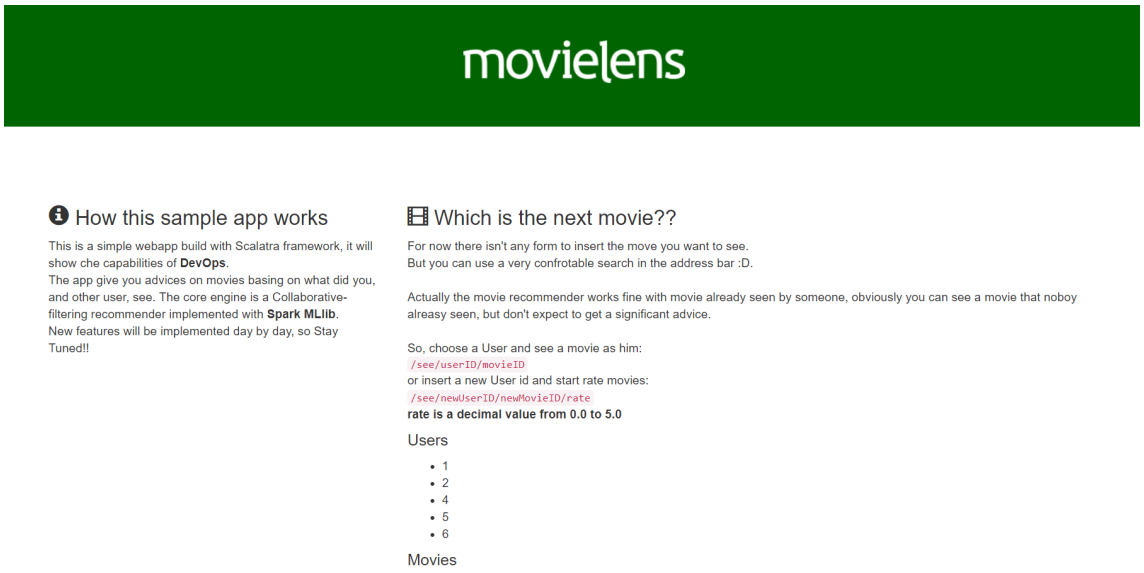


Figura 7.10: Frontend versione grezza

sarebbe tendere all'eliminazione delle vecchia versione a fronte dell'introduzione di una incrementale, ad esempio arricchendo la vista con le locandine dei film estratte dal BatchETL. Mentre la struttura del parametro di toggle rimane invariata, ciò che cambia e il contenuto proposto all'utente.

7.7 Continuous Deployment

Il Continuous Deployment rappresenta l'apoteosi dell'automazione di processo, ogni nuova Feature integrata nel Master Branch viene automaticamente rilasciata in produzione, garantendo qualità e continuità del servizio. Il Continuous Deployment mostra le sue maggiori potenzialità in quelle aziende che hanno a che fare con servizi daemon, fortemente real time, con la necessità di offrire costantemente un servizio. Il più comune esempio sono i siti web o le Paas, dove anche un solo minuto offline provoca un danno economico non trascurabile.

Le organizzazioni che sviluppano principalmente prodotti Batch non riescono a

percepire gli effettivi vantaggi di questa best practice, accontentandosi dell'automatizzare il deploy mantenendo comunque la scelta manuale.

Per raggiungere il livello di confidenza richiesto dal CD, il team deve aver abbracciato con successo le precedenti pratiche e aver costruito passo dopo passo una pipeline in grado di garantire la massima qualità e la totale fiducia riguardo il rilascio del componente in produzione. Infine anche l'ecosistema culturale del team deve essere ottimale, ognuno deve avere a cuore il prodotto e prendersi carico delle proprie responsabilità in armonia con i colleghi.

In questo progetto il Continuous Deployment è garantito dalla presenza di componenti software, basati ancora una volta su **Ansible**, in grado di gestire in maniera sicura il deploy in produzione. La pratica è inoltre supportata da un sistema di Continuous Monitoring in grado di verificare che il prodotto funzioni anche in ambiente di produzione reale.

7.7.1 Script di Deploy

Il deploy, in quanto parte della pipeline, non può più essere gestito con i tradizionali metodi basati su script. Il componente che esegue l'aggiornamento non deve solo assicurarsi che la nuova versione sia collocata nella corretta locazione, deve anche garantire che il componente sia attivo e funzionante. Allo scopo di fornire un controllo ottimale si è preferito gestire in maniera differente il deploy dei componenti Batch rispetto a quelli Real Time.

Deploy Batch

I due componenti Batch proposti hanno metodi di esecuzione differenti: BatchETL è eseguito on-demand all'inserimento di nuovi movies (o in corrispondenza di un nuovo aggiornamento), MRSpark2 è invece eseguito giornalmente a un orario ben

definito. Eseguire un deploy mentre il processo è in esecuzione porterebbe a numerosi problemi, in ogni caso i due processi sono per la maggior parte del tempo offline, quindi è sufficiente evitare di eseguire l'aggiornamento nel periodo di attività.

I **Playbook Ansible** che eseguono il Deploy devono quindi preoccuparsi semplicemente di trasferire i file necessari all'esecuzione nella corretta locazione del Edge Node di produzione. Nel Listato C.11 in Appendice C è mostrato il Playbook di deploy per BatchETL.

La Continuous Deployment Pipeline si occupa di archiviare i package assemblati nella cartella `/opt/deploy/*`, differente in base al processo in analisi. Il Playbook non fa altro che prendere il contenuto di tale cartella è trasferirlo nella directory **`/opt/devops/*`** sull'istanza DevOps Worker. Alla prima esecuzione verranno percepite le modifiche apportate.

Deploy Real Time

Discorso differente deve essere fatto per i processi Real Time presentati, entrambi hanno un comportamento da daemon, devono infatti essere sempre attivi e disponibili in modo da processare i dati in arrivo. Si è scelto di implementare entrambi i processi come dei servizi, accessibili da qualsiasi locazione del DevOps Worker tramite il comando `service *name* start/stop/status`. La configurazione dei file necessari è parte integrante del processo di Provisioning del DevOps Worker tramite Ansible, dando la possibilità di controllare sia l'aggiornamento che l'effettiva esecuzione del servizio.

RealTimeETL Arrestare questo servizio per il breve tempo di trasferimento di una versione aggiornata non comporta un grosso problema. Grazie al tempo di retain dei topics Kafka, gli eventuali ratings sopraggiunti durante il periodo di down verranno processati al riavvio, nel primo blocco di Discretized Stream. Il Playbook di deployment deve quindi assicurarsi soltanto di arrestare il daemon prima dell'aggiornamento e successivamente riattivare il servizio.

RealTimeMovieRec La sospensione di questo servizio è, in linea teorica, più delicata. Per come è stato proposto, l'interazione con il raccomandatore avviene solo tramite interfaccia grafica da parte di utenti interessati all'argomento, non trattando dati personali o comunque importanti il down del servizio comporterebbe semplicemente un maggior tempo di attesa o la visualizzazione di una pagina di Mantenimento.

In pratica però la webapp è in tutto e per tutto un server REST potenzialmente utilizzabile, tramite opportuni endpoint del tipo localhost:10000/raw/see/:user/:movie/:rating, da altri servizi per accedere al raccomandatore. Mettere offline il daemon per un aggiornamento potrebbe quindi significare compromettere le funzionalità di altri servizi correlati.

Al momento il deployment è gestito nella stessa maniera del RealTimeETL, in ogni caso sarà necessario implementare come sviluppo futuro una soluzione basata ad esempio sul **Blue/Green Deployment**, illustrato nel Capitolo 3. Il Listato C.12 in Appendice C mostra il Playbook di deployment del RealTimeETL.

Il Playbook è suddiviso in tre fasi:

1. Prima di tutto i servizi vengono arrestati;

2. La nuova versione viene trasferita, nello stesso identico modo del Deployment Batch;

3. Infine i servizi vengono riattivati, in questo caso specifico il playbook si assicura anche che il Connettore Kafka sia ancora attivo. Si noti inoltre che i servizi vengono lanciati con `enabled: yes`, questo significa che verranno eseguiti automaticamente all'avvio del sistema, nell'ordine specificato dai relativi file di configurazione.

7.8 Continuous Monitoring

il Continuous Monitoring permette di verificare che ogni fase del ciclo di vita del prodotto sia eseguita garantendo gli standard qualitativi necessari alla creazione di un sistema di valore. Tre sono le dimensioni significative da tenere in considerazione nell'implementazione di un valido sistema di monitoring:

- **Stato del Sistema**: deve essere garantito che ogni nodo e servizio all'interno del cluster sia attivo e funzionante. Questo comporta il monitoring delle risorse hardware del sistema e dello stato dei servizi fondamentali a sviluppo e funzionamento;

- **Processo di Sviluppo**: il processo di sviluppo deve essere monitorato non per dare valutazioni o demeriti riguardo l'efficienza del singolo sviluppatore, ma per correggere iterativamente le fasi che fanno da collo di bottiglia.

- **Prodotto in Produzione**: sul prodotto, una volta rilasciato in produzione, devono essere eseguiti Test (idealmente gli stessi della pipeline di Continuous Deployment) atti a verificare che i requisiti funzionali, non funzionali e qualitativi richiesti dal cliente siano rispettati.

L'ecosistema di **Prometheus** permette di arricchire queste tre dimensioni con metriche utili a valutare il ciclo di vita del sistema, combinandolo con **Grafana** è

stato possibile creare dashboard interattive e allarmi notificati direttamente all'ambiente Slack, utilizzato dal team per comunicare.

Prometheus colleziona metriche provenienti da fonti differenti in base alle necessità. Le soluzioni implementate dipendono fortemente dalla natura del processo e della dimensione in esame.

Tipi di Metriche

Prometheus supporta svariati tipi di metriche [16], i due più significativi sono:

Gauge Un valore puramente numerico che può sia crescere che decrescere all'interno della sessione. Ad esempio l'MSE relativo al raccomandatore è un Gauge.

Counter Valore numerico unicamente incrementale all'interno della sessione, il numero di predizioni richieste al RealTimeMovieRec è un Counter.

7.8.1 Stato del Sistema

Stato Hardware - Node Exporter

Monitorare lo stato hardware consiste nel verificare che non vi siano picchi di sovrautilizzo continuativi, tali da compromettere le funzionalità del sistema. Il componente utilizzato è il **Node Exporter** un tool in grado di campionare le risorse hardware generando un set di metriche estremamente significativo.

Un Exporter ha inoltre il compito di rendere accessibili le metriche collezionate a chiunque sia interessato ad analizzarle. Il node_exporter offre quindi un endpoint, accessibile alla porta `9100` (Figura 7.11), dove sono elencate le metriche collezionate.

Uno dei vantaggi di Prometheus sta nel formato di definizione delle metriche, essendo descrittivo e rappresentato in puro formato testuale, è particolarmente semplice creare Exporters custom, in grado di mettere a disposizione metriche significative per il proprio caso d'uso.

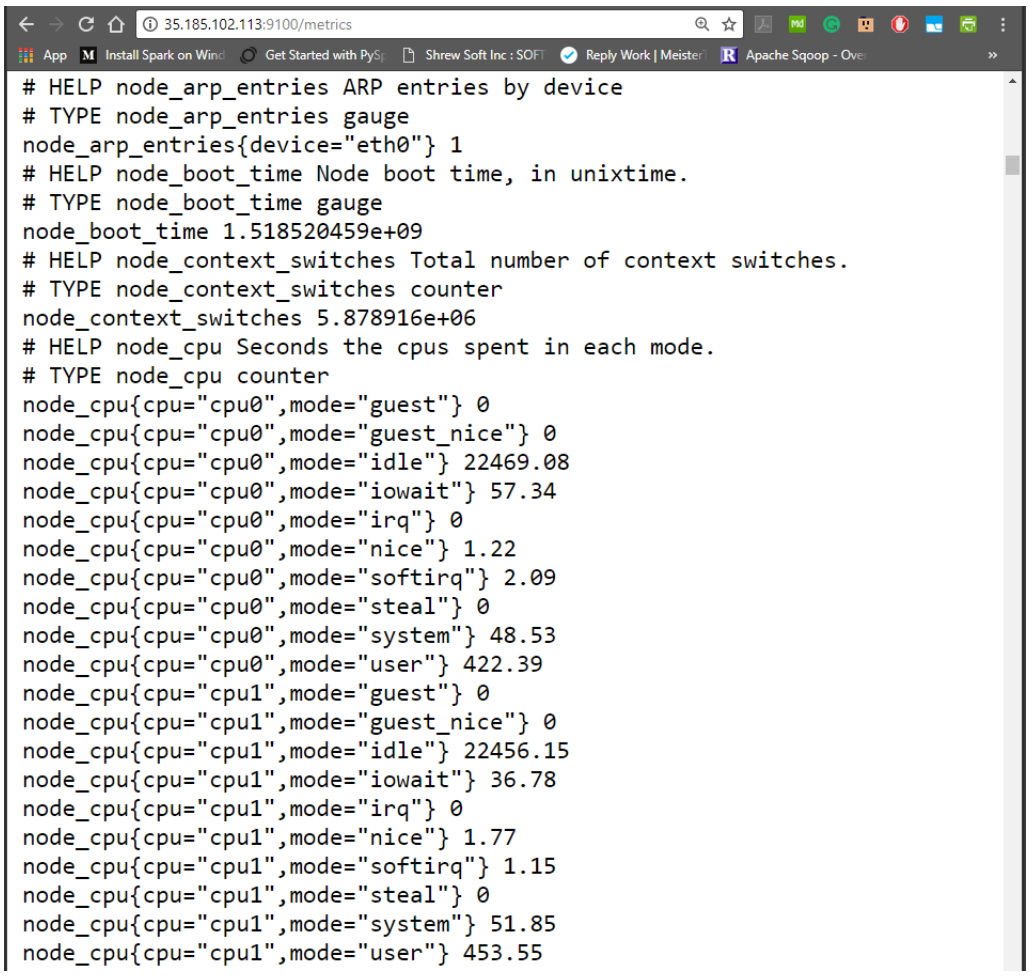


Figura 7.11: Metriche esposte dal Node Exporter

Prometheus si occupa dello scraping di tale endpoint mentre **Grafana**, basandosi sui dati estratti da Prometheus, mostra una dashboard utile all'analisi dello stato del sistema. Un Node Exporter è collocato sia sul DevOps Worker che sul Big Brother,

la dashboard Grafana permette di selezionare la sorgente di interesse.

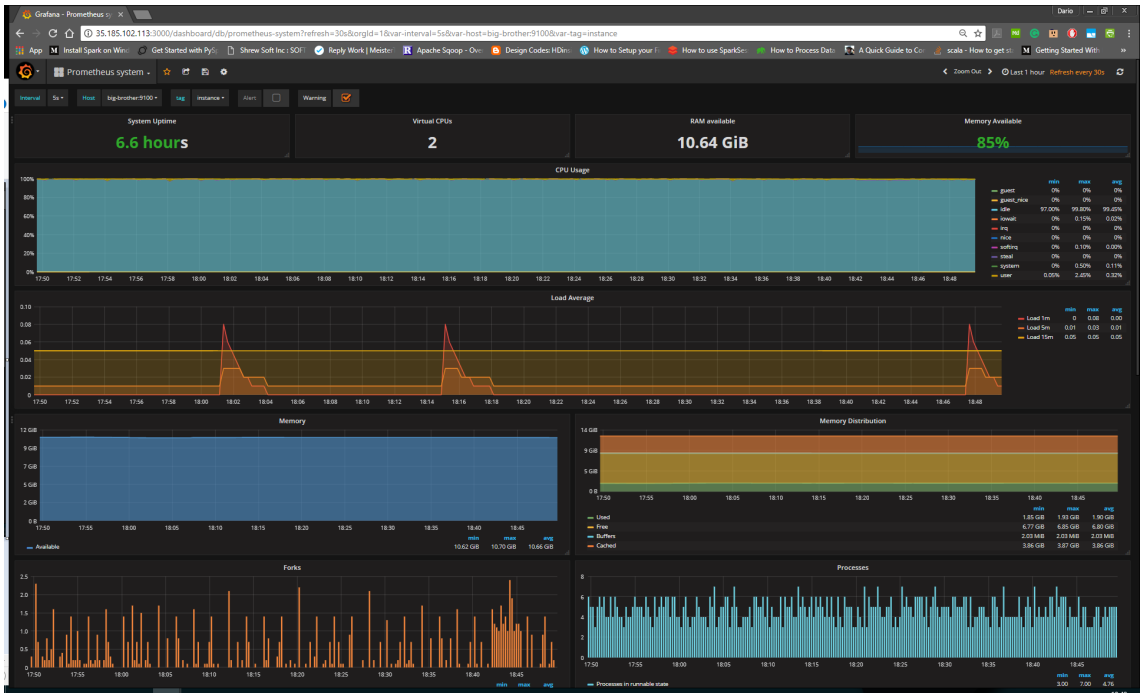


Figura 7.12: Dashboard Grafana del Node Exporter

Stato Hardware di Cloudera VM

La piattaforma Cloudera CDH offre nativamente un tool utile al monitoring hardware e software del sistema, ho valutato quindi che fosse superfluo inserire un ulteriore sistema di monitoring.

Come si può notare in Figura 7.13, Cloudera Manager presenta una dashboard unificata per stato hardware e software, dando una comoda visione di insieme. La Health issue indicata nel servizio HDFS è dovuta al sottodimensionamento del cluster, Cloudera infatti richiede almeno 3 nodi (master compreso), mentre al momento il sistema è distribuito su un solo nodo.

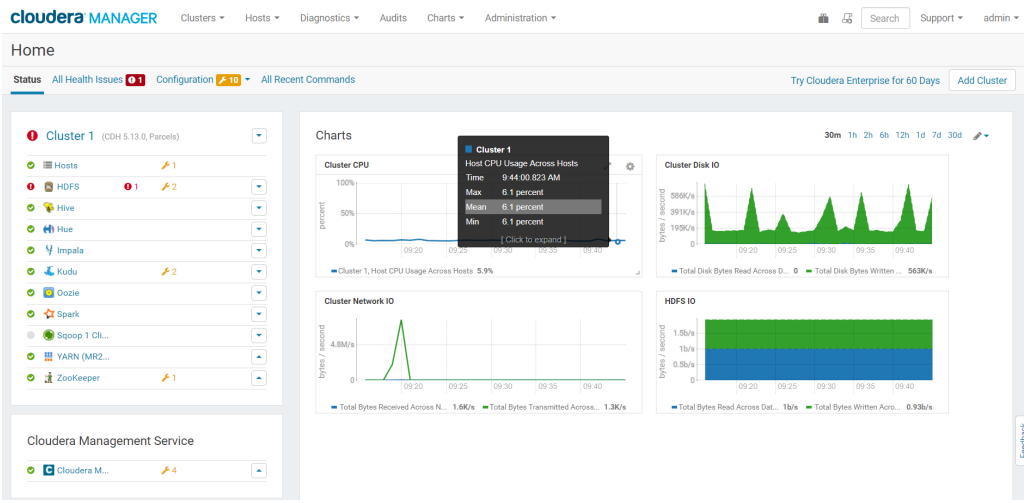


Figura 7.13: Dashboard Cloudera Manager

Stato Software - DevOpsMetricsExposer

Monitorare lo stato Software consiste nell'assicurarsi che tutti i servizi, necessari al corretto funzionamento del caso d'uso, siano attivi, funzionanti e raggiungibili. A tal scopo è stato implementato un Exporter custom, denominato **DevOpsMetricExposer**, in grado di analizzare lo stato del sistema e fornire un endpoint accessibile allo scraping di Prometheus.

DevOpsMetricExposer è una webapp Jetty-based, strutturalmente identica al RealTimeMovieRec, che offre una serie di endpoint utili per il monitoring. Il seguente gruppo di endpoint è pensato per essere utilizzato da un utente o un eventuale tool di controllo che desidera conoscere lo stato di uno specifico componente del sistema:

- `/isonline/:machine`: verifica che l'istanza target sia raggiungibile da un ping;
- `/count/hive/:database/:table`: conteggia il numero di elementi all'interno della specifica struttura dati del Data Lake;
- `/count/kudu/:database/:table`: conteggia il numero di elementi all'interno della struttura dati indicata del Data Mart. Monitorare il numero di elementi

in Data Lake e Data Mart è fondamentale per dare maggior valore alle metriche del raccomandatore e per garantire la consistenza del Data Warehouse;

- `/service/status/:machine/:service`: accede all'istanza indicata e verifica che il servizio richiesto sia attivo;

Sono inoltre offerti due endpoint più generali:

- `/metrics/fresh`: chiama in sequenza gli endpoint suddetti per ogni componente del sistema, l'idea è quella di collezionare le metriche per ogni dinamica in gioco e fornire una visione d'insieme dello stato del cluster. Questo endpoint è invocato periodicamente tramite un comando cron per mantenere aggiornate le metriche,
- `/metrics`: endpoint utilizzato a default da Prometheus per lo scraping delle metriche. Dal momento che il referesh delle metriche è una procedura relativamente onerosa in termini di tempo (mai superiore ai 2 minuti), mentre lo scraping di Prometheus viene eseguito ogni 30 secondi, si è pensato di mantenere in-memory lo stato attuale delle metriche e restituirlo in maniera passiva in corrispondenza dello scraping. L'aggiornamento delle metriche è eseguito da chiamate schedulate ogni 2 minuti.

Come estrarre lo stato

Per verificare lo stato di un'istanza e dei servizi su di essa attivi, si è pensato di sfruttare ancora una volta le capacità di **Ansible**. Prima di ogni esecuzione, allo scopo di garantire l'Idempotenza, Ansible esegue una serie di test atti a verificare lo stato corrente del target.

Il tool offre inoltre la **`dry-check mode`** con la quale è possibile eseguire soltanto la fase preliminare di analisi dello stato verificando di fatto se è necessario eseguire

azioni sul sistema.

Il DevOpsMetricExposer definisce quindi due Playbook parametrizzati: il primo per verificare lo stato di un servizio (Listato C.13), il secondo per verificare lo stato di un'istanza (Listato C.15).

Dashboard

Inifine Grafana offre dashboard per monitorare visivamente lo stato. Si è preferito mantenere la visualizzazione semplice nell'idea che possa permettere a chiunque di capire lo stato del sistema a colpo d'occhio.

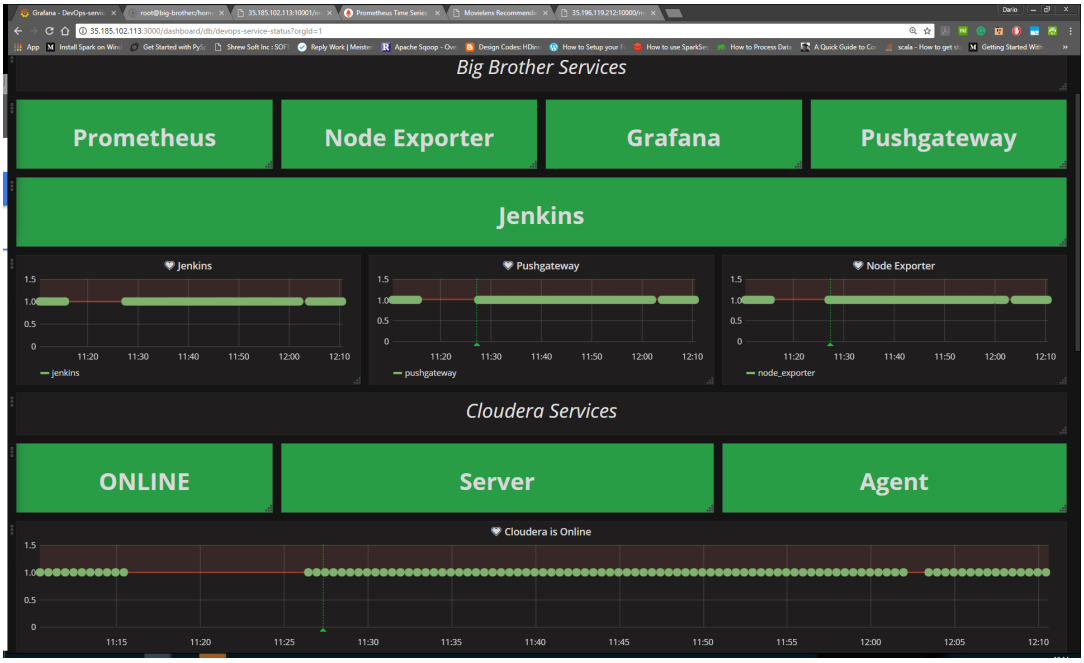


Figura 7.14: Stato di Cloudera VM e Big Brother

I grafici mostrati in Figura 7.14 e 7.15, affiancati alle metriche nominali, sono utili sia per valutare l'andamento dello stato nel tempo, sia per innescare allarmi in caso di stato offline prolungato.

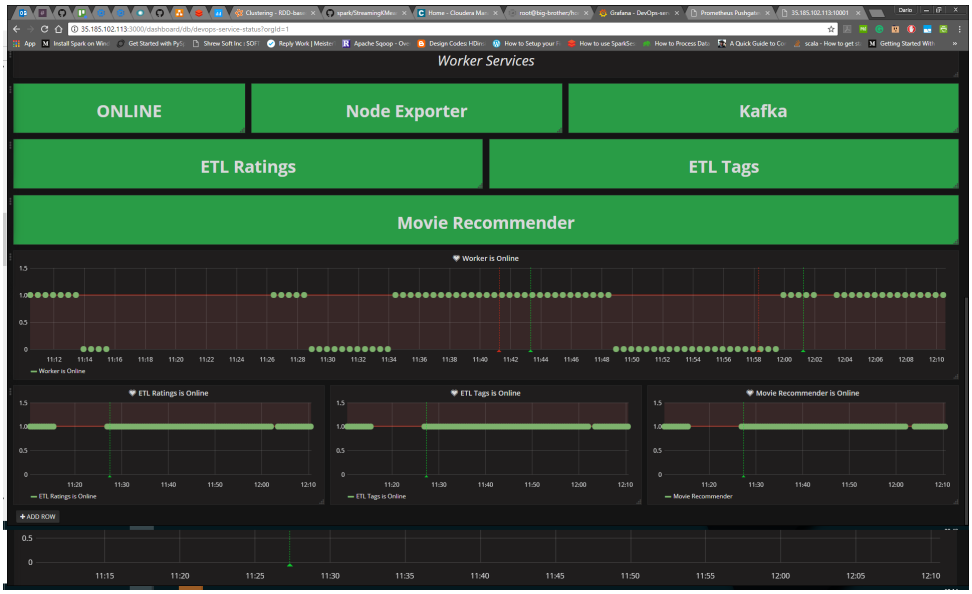


Figura 7.15: Stato del DevOps Worker

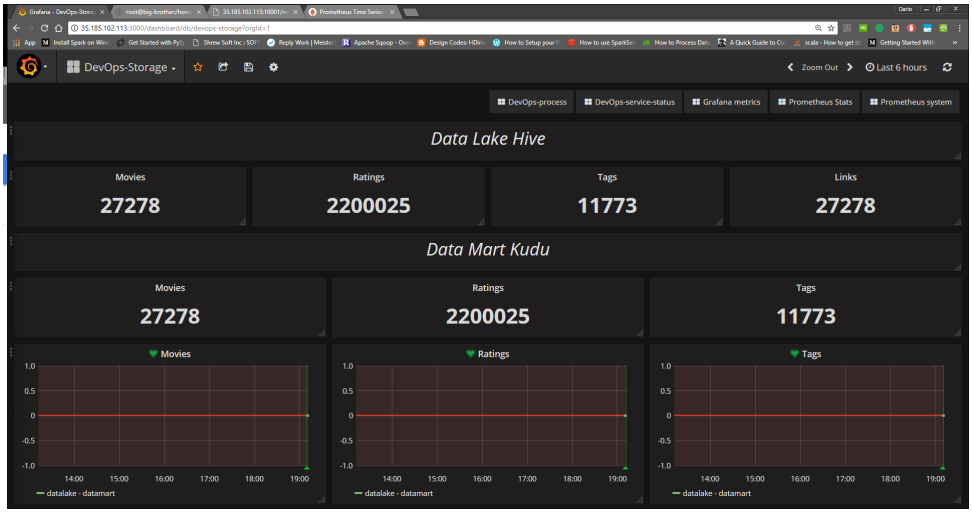


Figura 7.16: Dashboard di memorizzazione

Infine, in Figura 7.16 è mostrata la dashboard relativa allo stato del Data Warehouse. Ancora una volta i grafici sono utilizzati per rilevare disallineamenti prolungati tra il numero di elementi di Data Lake e Data Mart.

7.8.2 Processo di Sviluppo

Ogni Stage della Continuous Deployment Pipeline è monitorato al fine di ottimizzare il processo, esecuzione dopo esecuzione. Le metriche di interesse sono date nello specifico dai risultati dei test e dai tempi di esecuzione delle pipeline, osservati sia tramite la Dashboard Blue Ocean sia da quella default di Jenkins, arricchita da alcuni plugin.

Blue Ocean

La Dashboard Blue Ocean è comoda per seguire il flusso della pipeline durante la sua esecuzione. Una pecca di Jenkins sta nel parsing e la visualizzazione dei log, Blue Ocean fa fronte a questo problema integrando un sistema autonomo di visualizzazione, strutturato in una dashboard simile a un terminale Linux. I Log vengono visualizzati come uno stream aggiornato automaticamente dal sistema. La Dashboard permette inoltre di analizzare separatamente l'output di ogni step di ogni Stage rendendo facile il lavoro di analisi in caso di fallimento.

La pecca di Blue Ocean sta nel non offrire una visione di insieme di tutto il processo, inoltre la Tab di test archiviati è ancora in versione preliminare, non mostrando di fatti nulla di più di un messaggio di conferma.

Jenkins

La dashboard Jenkins, arricchita da alcuni plugin di monitoring, permette di tenere sotto controllo il processo nel suo insieme, in particolare è possibile vedere l'andamento dello stato delle build nel tempo, e i tempi di esecuzione per ogni singola fase. In questo modo è possibile identificare immediatamente eventuali colli di bottiglia ottimizzando il processo di sviluppo.

Full Stage View Questa dashboard, in Figura 7.17, permette di visualizzare l'andamento della pipeline nella sua interezza, mettendo in luce i tempi di elaborazione dei singoli Stage, correlati tra esecuzioni dello storico.

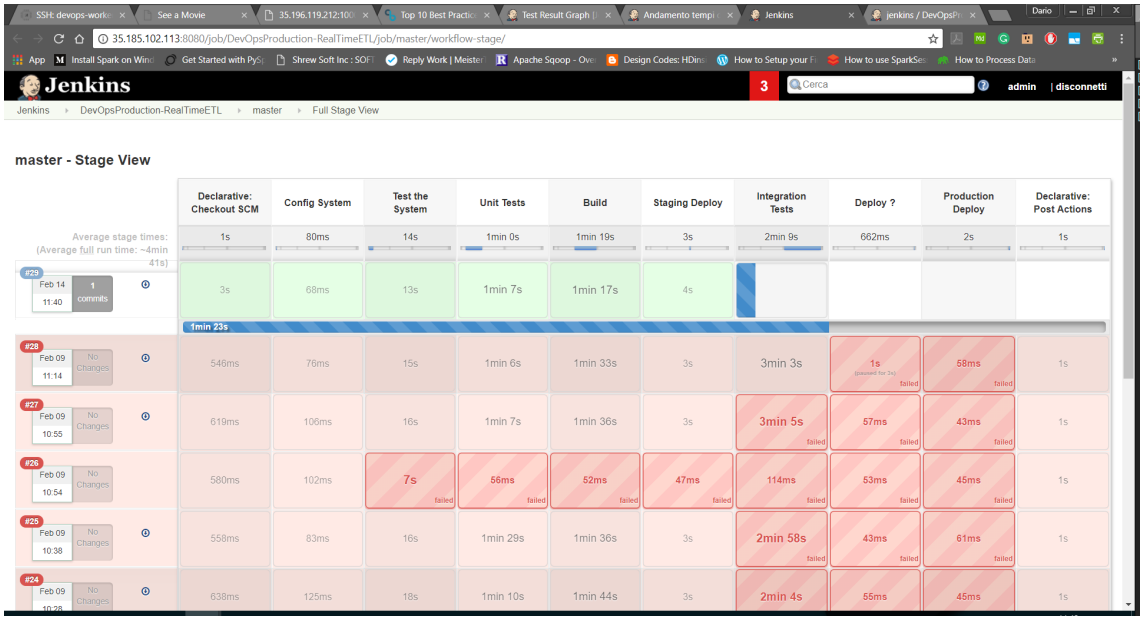


Figura 7.17: Full Stage View Jenkins

Test Result Analyzer La seconda Dashboard è pensata per correlare graficamente l'esito dei singoli Test tra diverse esecuzioni della pipeline. A colpo d'occhio è possibile valutare l'andamento dei test nel tempo, percependo la qualità della pipeline implementata. Ancora una volta, piuttosto che dedicare una persona all'analisi delle dashboard, è più significativo mostrarle in tempo reale a tutto il team, in questo modo si ha la massima trasparenza su ogni commit e si aumenta la responsabilizzazione del singolo individuo.

7.8.3 Stato dei Processi

Monitorare lo stato dei processi è fondamentale per assicurarsi che tutto funzioni come atteso e che le performance siano tali da rispettare gli standard qualitativi

richiesti. Sono state adottate differenti soluzioni di monitoring, sia a causa di necessità tecniche dipendenti dalla natura dei processi, sia a fine di sperimentare differenti metodologie.

BatchETL e MRSpark2

Il protocollo standard di Prometheus è di effettuare uno Scraping periodico sui processi da monitorare, al fine di estrarre le metriche aggiornate. Essendo BatchETL e MRSpark2 processi Batch, essi sono per lo più offline, rendendo inefficace l'approccio tradizionale. L'ecosistema Prometheus fornisce uno tool di supporto, il **Pushgateway**, appositamente pensato per collezionare le metriche di processi batch.

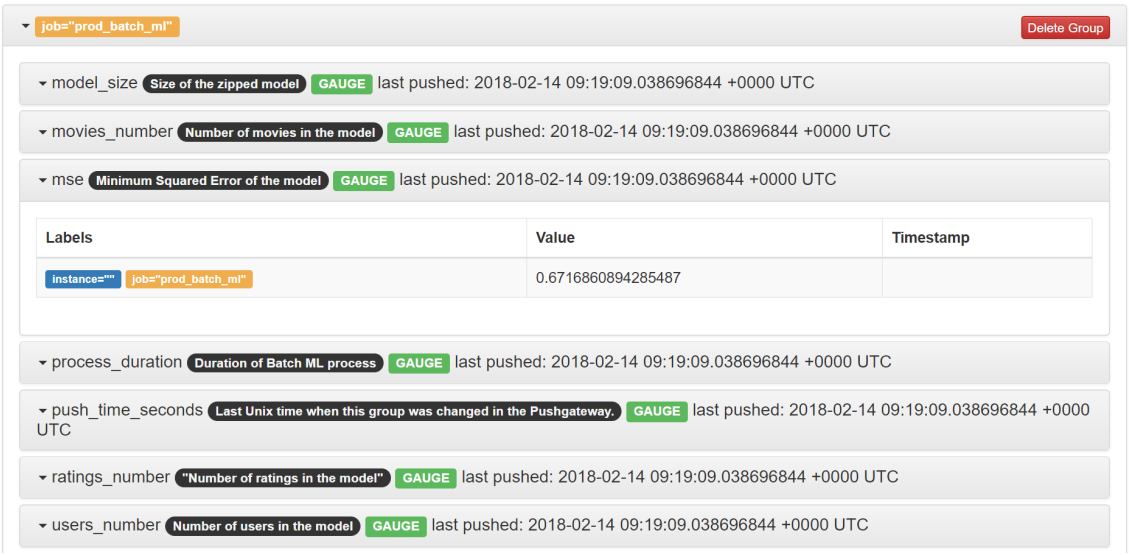


Figura 7.18: Metriche del processo Batch ML

Come si può intuire dal nome, la logica di scraping viene invertita, sono i processi stessi a doversi preoccupare di fare il push delle metriche verso il Pushgateway. In Figura 7.18 sono mostrate le metriche generate da MRSpark2, si noti come ogni metrica sia arricchita dall'istante di rilevamento in modo da poterla tracciare nel

tempo. Il Pushgateway si occupa di esporre tutte le metriche consegnate, mantenendole per un periodo di retain configurabile.

Questo tipo di pratica appartiene alla categoria **White-box** e richiede l'inserimento del calcolo delle metriche direttamente all'interno del codice, risulta quindi una pratica invasiva, facilmente applicabile solo quando il codice è strutturato in modo modulare e robusto. In ogni caso è bene ricordare che il calcolo delle metriche white-box richiede un tempo di elaborazione aggiuntivo a quello dell'effettivo processo; introdurre troppe metriche comporta quindi un'aggiunta significativa ai tempi di esecuzione. In casi di elaborazione Batch Big Data, dove i tempi sono di base già lunghi, il delay introdotto è difficilmente percepibile, è comunque bene minimizzare le metriche introducendo solo quelle strettamente necessarie.

Il Listato C.16 mostra l'introduzione delle metriche in MRSpark2, il client Pushgateway è facilmente integrabile in Scala grazie a SBT, per ogni metrica (in questo caso solo Gauge), è necessario specificare tipo, valore e descrizione. La raccolta delle metriche è effettuata dal CollectorRegistry del quale viene fatto il push verso il gateway al termine dell'elaborazione.

RealTimeMovieRec

Il monitoring di processi real time si riduce all'interrogazione periodica di un endpoint dedicato allo scraping, che espone le metriche di interesse per il processo, esattamente come mostrato per gli **Exporters**. Il RealTimeMovieRec presenta un endpoint `/metrics` che restituisce le metriche collezionate ed aggiornate ad ogni nuova richiesta di raccomandazione. Il Prometheus Server è configurato per effettuare lo scraping di tale endpoint.

RealTimeETL

Per quanto si tratti di un processo real time, a causa di Spark Streaming risulta particolarmente complesso effettuare il monitoring del processo. La causa è legata alla natura distribuita dell'elaborazione per cui un componente di monitoring deve essere serializzabile per poter essere trasferito tra i nodi del cluster; inoltre, a causa della distribuzione stessa, risulta poco significativo monitorare il numero di `ratings` estratti e processati dato che essi saranno elaborati da differenti nodi in modo non deterministico.

Dashboard

Lo stato dei processi è mostrato tramite una Dashboard Grafana, mostrata in Figura 7.19, che mette in luce le performances del raccomandatore.

7.9 Stima del vantaggio

In questa ultima sezione si vuole dare una stima del vantaggio, sia economico che temporale, ottenuto da un'azienda che sfrutti le metodologie DevOps.

7.9.1 Scenario

Un'azienda di consulenza informatica offre ai propri clienti un servizio di sviluppo progetti totalmente autonomo. Dopo la consegna e discussione dei requisiti con il cliente, l'azienda si occupa dello sviluppo del prodotto in ogni sua fase, dalla definizione del modello sino al deploy e monitoring successivo.

Al momento della commissione, il cliente ha la possibilità di scegliere tra due differenti team di sviluppo: un team tradizionale, che sfrutta la metodologia Agile, e un team DevOps in grado di curare il prodotto a 360°.

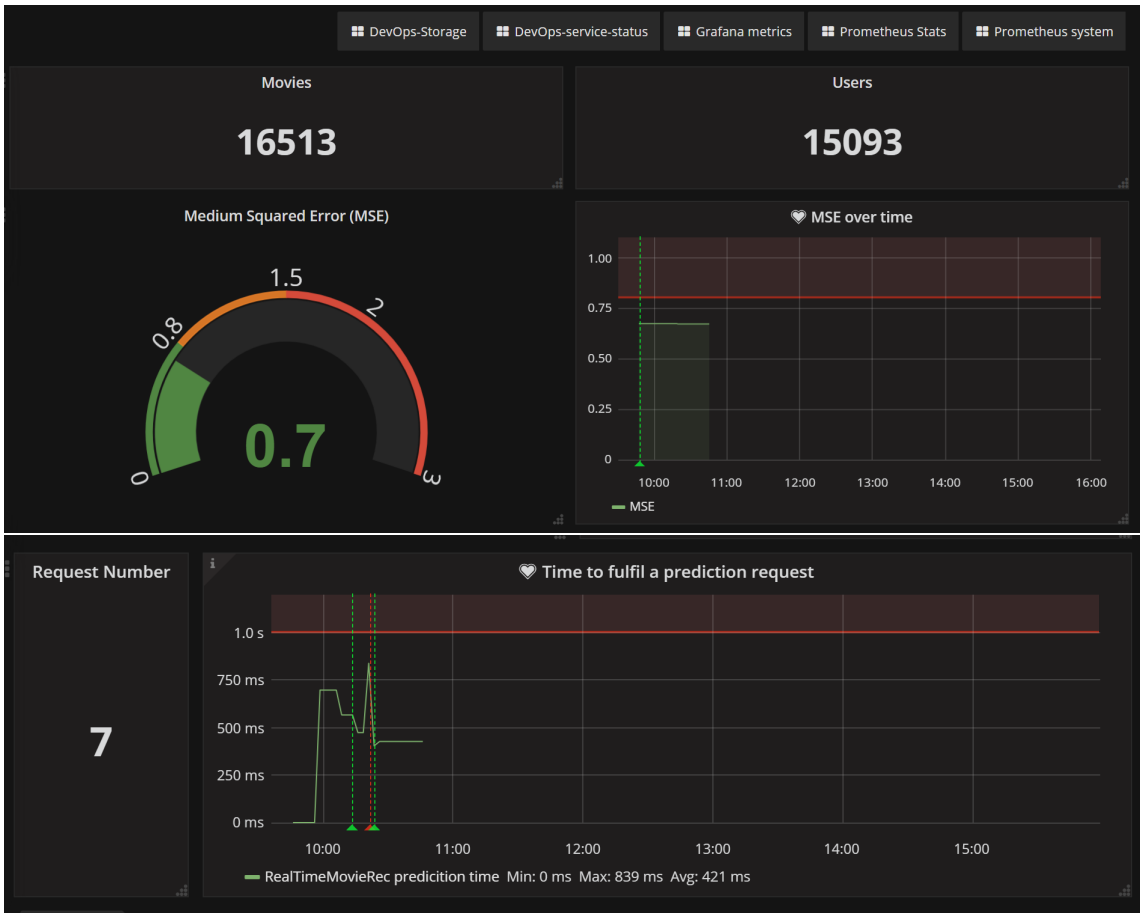


Figura 7.19: Stato del Raccomandatore

Team Agile

Si suppone che il team Agile lavori in questo modo:

- Il team è in realtà suddiviso in due sotto-team: un team di Dev, proporzionale alla complessità del progetto, e un di Ops, proporzionale al numero di Devs. Entrambi i team lavorano full-time al progetto.

- Dopo una prima fase iniziale, necessaria allo sviluppo del core del sistema, il team sfrutta la metodologia Agile per rilasciare versioni incrementali ogni 5 giorni lavorativi.

Team DevOps

Il team DevOps invece lavora nel seguente modo:

- L'applicazione del metodo DevOps necessità di un periodo di lavoro preventivo, necessario al deploy della pipeline di Continuous Deployment nel sistema del cliente. Il lavoro è svolto da un Senior Consultant, esperto di DevOps, supportato da un Ops. Il processo di setup richiede 20 giorni lavorativi.

- Il team DevOps è composto da un certo numero di sviluppatori full-time, proporzionale alla complessità del progetto, affiancato dal Ops (part-time) che ha supportato il deploy della pipeline.

- Dopo una fase iniziale, necessaria allo sviluppo del core, il team rilascia versioni incrementali ogni 3 giorni lavorativi.

7.9.2 Dati di stima

- Uno sviluppatore (tradizionale o DevOps) full-time costa 1;

- Lo stesso sviluppatore a part-time costa la metà;

- Il Senior Consultant costa 2.

- In media ci sono 20 giorni lavorativi al mese.

7.9.3 Stima

Supponendo che l'azienda riesca a condensare i requisiti del progetto in un numero finito di Features a complessità unitaria, si vuole stimare il costo imposto al cliente, in termini economici e temporali, nei due differenti approcci.

I seguenti grafici indicano in rosso il team DevOps e in blu il team Tradizionale. In Figura 7.20 è mostrato l'andamento di costo dell'intero processo, espresso per generalità in giornate lavorative.

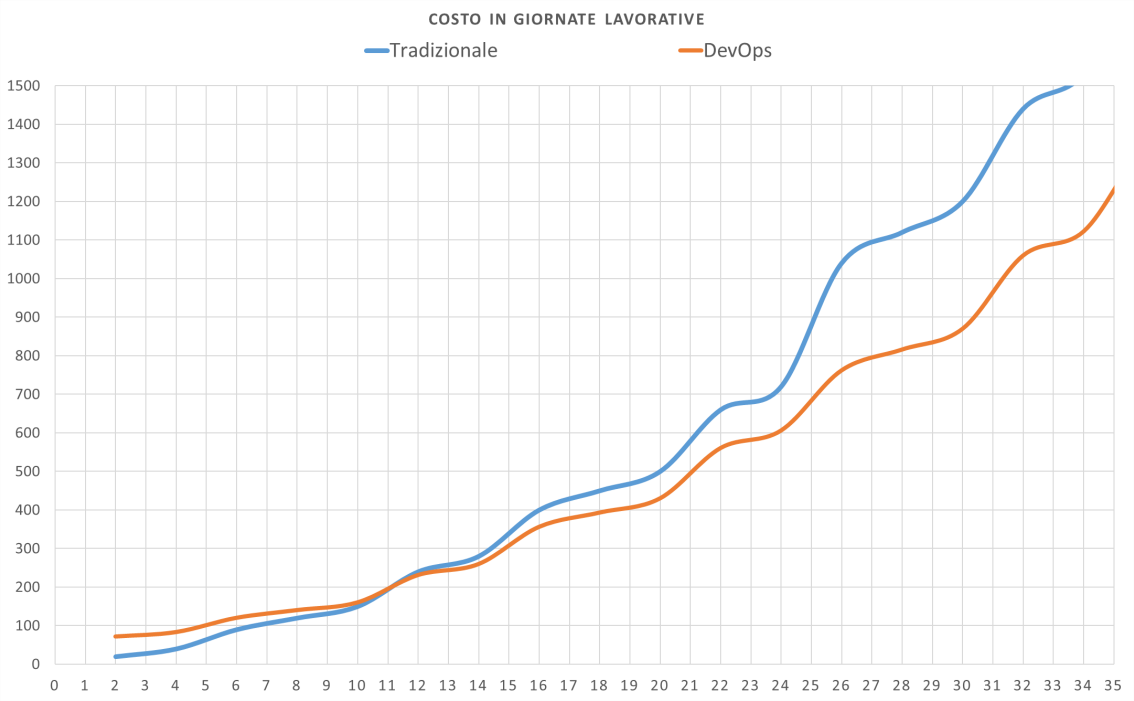


Figura 7.20: Stima del costo all'aumentare della complessità

In Figura 7.21 è mostrato invece il tempo necessario al completamento del progetto, espresso in mesi di lavoro.

Come si può notare, l'approccio DevOps richiede un investimento iniziale maggiore, sia in termini economici che temporali, il motivo è ovviamente dato dal deploy della pipeline e relativo costo associato. Non vi è quindi un effettivo vantaggio nel applicare il DevOps per progetti di piccole dimensioni.

I vantaggi economici iniziano a mostrarsi prima di quelli temporali, con progetti di almeno 11 Features il costo stimato è minore rispetto al modello Tradizionale. Il vantaggio temporale si ha solo con progetti ancor più complessi (almeno 19 Features), in grado di ammortizzare il cold-start iniziale di deploy.

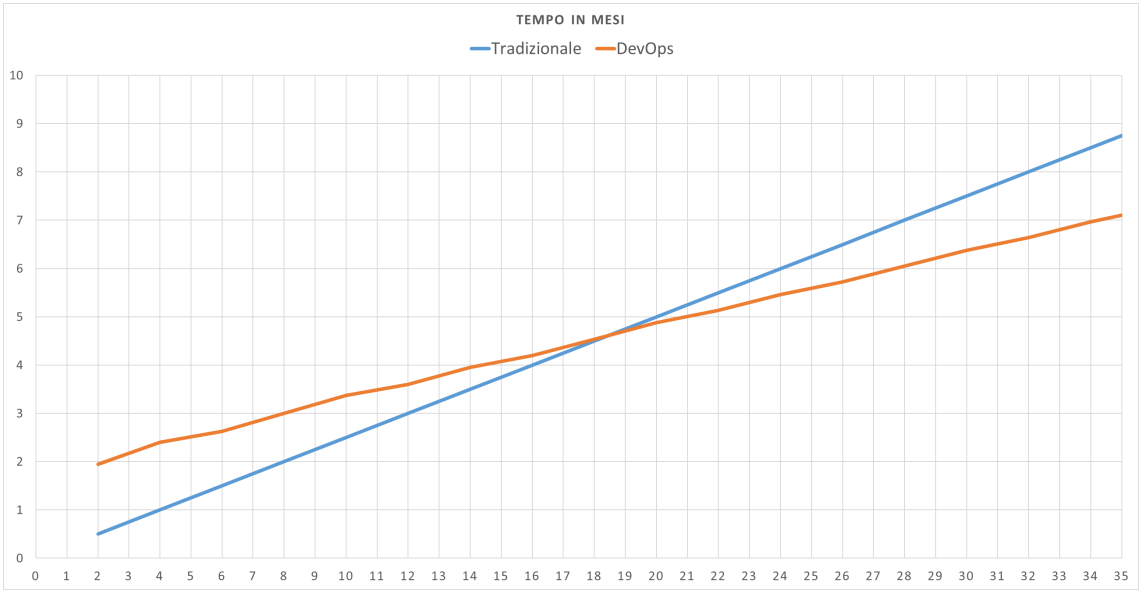


Figura 7.21: Stima del tempo di sviluppo all'aumentare della complessità

Allo scalare della complessità il divario non fa altro che aumentare, a complessità 50 vi sono circa 750 giornate di differenza, corrispondenti a circa 2 mesi di lavoro. Si ricorda inoltre che la stima è fatta supponendo un cliente totalmente estraneo alla tecnologia DevOps, in caso di clienti con un'adeguata architettura di deployment i vantaggi iniziano a manifestarsi ben prima. Inoltre il vantaggio aumenta mano a mano che il team prende confidenza con la metodologia, diminuendo ulteriormente il tempo necessario al release di una nuova feature.

Infine i vantaggi non sono soltanto economici, vi è un guadagno non calcolabile in termini di organizzazione, modularità e struttura del progetto e del sistema aziendale, convertibili in risparmio di tempo e denaro.

Capitolo 8

Conclusioni

In questa tesi è stato studiato il movimento DevOps in ogni sua sfaccettatura, presentandolo sia da un punto di vista culturale sia dal punto di vista prettamente implementativo.

Cultura DevOps

Sono stati analizzati a fondo i principi presentati dal movimento DevOps, mettendo in luce i vantaggi da essi portati e le difficoltà legate all'adozione. Ciascuna delle sette pratiche presentate apporta un significativo valore aggiunto al prodotto, al processo di sviluppo ma soprattutto alla qualità del lavoro di team.

Certo lo sforzo necessario a una completa adozione non è trascurabile, ritengo comunque che, una volta presa coscienza del problema, possano essere i diretti interessati a svolgere il ruolo di motrice del cambiamento.

Architettura

L'architettura Big Data presentata rispecchia i comuni modelli a livello Enterprise, rappresentando una struttura facilmente adattabile alle casistiche dello specifico caso d'uso. Il Configuration Management, implementato con Ansible, porta un ulteriore

valore aggiunto, permettendo di definire servizi e responsabilità sotto forma di ruoli, modulari, auto-contenuti e definiti tramite un linguaggio di facile comprensione e condivisione.

L'introduzione dell'Infrastructure As a Code, basata su Terraform, completa il processo di Orchestration e Provisioning, rendendo lo scaling orizzontale automatico e facilmente customizzabile. Grazie alla combinazione di Terrafom e Ansible è possibile implementare il modello a Immutable Infrastructure in maniera più modulare rispetto all'utilizzo di un cluster container-based.

Processi

Nel caso d'uso presentato si è voluto mostrare un comune modello di pipeline di elaborazione in ambito Big Data. Identificando due tipi principali di task (Estraction Transformation Loading e Machine Learning) e due principali tipi di elaborazione (Batch e Real Time) si è cercato di implementare le quattro principali combinazioni task-elaborazione, in modo da strutturare un modello di carattere generale.

Il raccomandatore di film, basato sul Collaborative Filtering e implementato con l'algoritmo Alternating Least Square, è pensato per essere un valido PoC senza la pretesa di utilizzo in produzione. Il modello sviluppato permette di generare raccomandazioni con un MSE di 0.7 e RMSE di 0.84. Lo stesso algoritmo è stato inoltre utilizzato per computare un modello binario, in grado soltanto di esprimere un giudizio positivo o negativo. I risultati in questo caso sono più soddisfacenti, si ha un'Accuracy 87%, Precision 94% e Recall 95%. Questo tipo di modello genera però raccomandazioni meno significative.

Strettamente parlando, il processo RealTimeMovieRec non è un sistema di Machine Learning real time, dal momento che i dati processati vengono semplicemente utilizzati per l'interrogazione del modello, senza però incrementarne la conoscenza. Il processo risulta comunque un valido caso d'uso di applicazione delle best practices

a un servizio web.

Pipeline di elaborazione

La pipeline presentata definisce due stream di elaborazione, utilizzando tecnologie all'avanguardia. Sviluppandola mi è stato permesso di apprendere nozioni nuove, parzialmente trascurate duranti gli anni di studio universitario.

Ottimizzazione del processo con le pratiche DevOps

Il processo di sviluppo dei quattro progetti è stato strutturato secondo le Best Practices presentate dal movimento DevOps.

Strutturare il processo di sviluppo in piccoli blocchi auto-contenuti, nell'ottica del Continuous Integration, ha permesso di coprire al meglio le specifiche progettuali, componendo un prodotto modulare e facilmente mantenibile.

I test, ideati secondo i dogmi del Test Driven Development, riescono a coprire la maggior parte delle casistiche di errore, garantendo un alto livello di qualità, consono con gli standard richiesti dal mercato. L'utilizzo di un ambiente di Staging Test permette inoltre di analizzare il comportamento del sistema in un ambiente protetto, del tutto identico a quello di produzione, prevenendo il rilascio della quasi totalità di bug.

La Pipeline finale di Continuous Deployment consente effettivamente di rilasciare nuove features in maniera rapida e automatica, aprendo la possibilità a deploy giornalieri, man mano che avanza lo sviluppo e l'integrazione. Il vantaggio apportato dal Continuous Delivery e dalle Toggle Features garantisce al mondo del business di poter scegliere il momento più opportuno per andare in produzione, disaccoppiando a tutti gli effetti funzionamento e valore commerciale.

Infine, le dashboard e il sistema di alert del Continuous Monitoring, permettono di chiudere il loop di sviluppo, dando un feedback immediato riguardo ogni aspetto

del ciclo di vita del prodotto. Introducendo metriche e notifiche direttamente nell'ambiente di comunicazione del team, è stato possibile incrementare la produttività e la trasparenza responsabilizzando il singolo.

Considerazioni sul legame tra DevOps e Big Data

Dopo aver analizzato entrambi gli ambiti, credo fermamente che il mondo dei Big Data possa trarre enorme vantaggio dalle tecniche presentate dal movimento DevOps. I problemi di gestione dati e di progettazione architetturale possono essere saggiamente gestiti con gli strumenti proposti, permettendo di automatizzare sia il deploy infrastrutturale del cluster, sia il rilascio continuo di features in produzione. La pipeline di Continuous Deployment, arricchita da numerosi stage di test, permette di rilasciare software ad alta qualità, caratteristica fondamentale nel momento in cui un singolo processo è responsabile dell'elaborazione di una grande quantità di informazioni. L'industrializzazione e automazione del rilascio, permette infine di concentrare lo sforzo del team sullo sviluppo invece che sulla correzione e il mantenimento.

Per quanto riguarda l'abstraction gap di adozione, la mia esperienza è limitata a una visione fortemente di supporto. In un'azienda di consulenza, quale Data Reply, è più complesso introdurre le best practicies rispetto a un'azienda con un proprio prodotto. In ogni caso l'ecosistema proposto è facilmente convertibile in un layer di supporto Enterprise fruibile al cliente come qualsiasi altro servizio di consulenza.

Vantaggi per il business

E' stato inoltre mostrato come l'adozione delle metodologie DevOps, possa apportare un reale vantaggio al business aziendale. Nonostante sia necessario un sforzo iniziale, dovuto al deploy dell'architettura necessaria alla pipeline di CD, questo sforzo è ripagato già dal primo progetto, aumentando ulteriormente man mano che il team

prende confidenza con gli strumenti. Ovviamente questo vale solo per progetti di medie dimensioni, applicare la tecnologia a piccoli applicativi comporta soltanto una perdita in termini di tempo e denaro.

Esperienza in Data Reply

Sono pienamente soddisfatto del periodo di tirocinio svolto presso Data Reply Spa. Questa esperienza mi ha permesso di incrementare notevolmente il mio bagaglio culturale, esplorando aspetti del mondo informatico trascurati in precedenza. Il lavoro "sul campo" è stato fondamentale per capire al meglio i problemi e le motivazioni che hanno spinto il mondo verso l'ideazione del movimento DevOps, probabilmente svolgendo la stessa tesi senza la collocazione in un'azienda il lavoro risultate sarebbe stato meno completo e consapevole.

Capitolo 9

Sviluppi Futuri

Pipeline e Processi di elaborazione

Volendo migliorare le fasi del processo di elaborazione, nell'ottica di offrire un servizio di raccomandazioni real time, due sono i punti principali di ottimizzazione:

- Il connettore Kafka JDBC, per quanto performante, adotta un approccio a polling, generando un'interazione a mini-batch piuttosto che real time. Si potrebbe migliorare il processo introducendo un layer di Change Data Caputre (CDC) in grado di ascoltare le modifiche fatte sulla sorgente e inviare al connettore i nuovi dati aggiunti secondo una logica di Push.

- Si potrebbe inoltre completare la panoramica dei processi, implementando un effettivo Machine Learning Real Time. Ad esempio si potrebbe applicare un clustering real time (con l'approssimazione stream dell'algoritmo K-means) per raggruppare i movies e gli utenti per gruppi di similarità.

Ottimizzazione del processo

Nell'ottica del Continuous Deployment, i test sono l'unica fonte di protezione dal rilascio di bug in produzione. Contribuiscono inoltre a definire la qualità e il valore

del prodotto finale. Al momento i test sono definiti e strutturati da specializzati membri del team sulla base del Test Driven Development, non si ha però la certezza assoluta che tale gruppo di persone riesca a garantire la totale copertura delle casistiche di errore.

Potrebbe essere interessante studiare una versione speculare del processo, eseguendo il mining dei log in produzione si potrebbero rilevare i casi di errore (potenzialmente sfuggiti alla fase di testing) incrementando la popolazione di test in maniera iterativa. Il feedback loop, parallelo alla pipeline di produzione, sarebbe utile per migliorare la batteria di test mano a mano che il prodotto viene sviluppato.

Ecosistema DevOps

Infine, si potrebbe racchiudere le tecniche implementate in un framework, open source e in grado di fornire supporto al processo di sviluppo in maniera out-of-the-box. In questo modo si faciliterebbe la diffusione culturale, contribuendo allo sviluppo di prodotti migliori.

Appendice A

A.1 Configuration Management

Playbook Ansible

```
- hosts: worker
  become: yes
  become_user: root
  tasks:
  - name: Install Wget
    yum:
      name: wget
      state: latest
      update_cache: true

  - name: Install cURL
    yum:
      name: curl
      state: latest
      update_cache: true
```

```
- name: Install JDK
  yum:
    name: java-1.8.0-openjdk
    state: present



- name: Donwload SBT
  get_url:
    url: http://dl.bintray.com/sbt/rpm/sbt-0.13.12.rpm
    dest: /opt/sbt-0.13.12.rpm
    mode: 0440

- name: Install SBT
  yum:
    name: /opt/sbt-0.13.12.rpm
    state: present
```

Listing A.1: Esempio di playbook Ansible SBT

Playbook come Role

```
- hosts: worker
  become: yes
  become_user: root
  roles:
  - SBT
```

Listing A.2: Playbook implementato come Role

A.2 Infrastructure As a Code

Configurazione Terraform

```
resource "google_compute_instance" "worker" {
    project = "devops-project"
    zone = "us-east1-b"
    count = "${var.server_count}"
    name = "devops-worker-${count.index}"
    machine_type = "n1-standard-2"
}
```

Listing A.3: Configurazione Terraform GCP

A.3 Continuous Integration

```
pipeline {
  agent any
  environment {
    VAR = 'value'
  }
  stages {
    stage('Unit Tests') {
      steps {
        sh 'sbt clean coverage test coverageReport'
      }
    }
    stage('Build') {
      steps {
```

```
        sh 'sbt clean compile package assembly'
      }
    }
    stage('Integration Tests') {
      steps {
        sh 'cd IntegrationEnv/ && sbt clean test'
      }
    }
  }
  post {
    success {
      echo "Success"
    }
    failure {
      echo "Fail"
    }
  }
}
```

Listing A.4: Jenkinsfile

Appendice B

B.1 Ingestion

Estrazione Sqoop

```
sqoop import --connect 'jdbc:postgresql://localhost:7432/postgres'
    --username 'cloudera-scm'
    --password $CLOUDERA_PASSWD
    --table 'movies'
    --hive-table 'datalake.movies'
    --hive-import
    --check-column id
    --append
    --incremental 'append'
```

Listing B.1: Movies Sqoop Import

B.2 ETL

B.2.1 BatchETL

Transformation

```
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.functions.udf

object ETL {

  def enrichMovies(movies : DataFrame, links : DataFrame) : DataFrame = {

    val fullLink : String => String = {tmdb =>
      "https://www.themoviedb.org/movie/"+tmdb
    }
    val fullLinkUDF = udf(fullLink)

    var outMovies = movies
      .join(links, movies("movieid")===links("movieid"))

    val movieCols = Seq("movieid", "title", "genres", "full")

    outMovies = outMovies.withColumn("full",
        fullLinkUDF(outMovies("tmdbid")))
      .toDF("id", "movieid", "title", "genres", "linkid", "linkmovieid",
          "imdbid", "tmdbid", "full")
      .select(movieCols.head, movieCols.tail: _*)
      .toDF("movieid", "title", "genres", "link")
```

```
    outMovies
  }
}
```

Listing B.2: Trasformazione di Movies e Links

B.2.2 Loading

```
object BatchETL {

  def main(args: Array[String]): Unit = {

    Caricamento della configurazione da application.conf
    Inizializzazione Storage - SparkSession
    Inizializzazione Storage - Kudu

    HIVE_MOVIES = "datalake.movies"
    HIVE_LINKS = "datalake.links"
    KUDU_MOVIES = "impala::datamart.movies"

    if(not storage.esists(KUDU_MOVIES)){
      Chiudi SparkSession
      exit()
    }

    movies = storage.readFromHive(HIVE_MOVIES)
    links = storage.readFromHive(KUDU_MOVIES)

    outMovies = ETL.enrichMovies(movies, links)
```

```
    storage.upsert(outMovies, KUDU_MOVIES)

    Chiudi SparkSession
  }
}
```

Listing B.3: Pseudocodice del processo BatchETL

Creazione Struttura dati Kudu

```
CREATE TABLE datamart.movies
(
  movieid INT NOT NULL,
  title STRING NULL,
  genres STRING NULL,
  link STRING NULL,
  PRIMARY KEY (movieid)
)
PARTITION BY HASH (movieid)
PARTITIONS 2
STORED AS KUDU
TBLPROPERTIES
    ('kudu.master_addresses'='cloudera-vm.c.endless-upgrade-187216.internal')
```

Listing B.4: Creazione della tabella movies nel Data Lake

B.2.3 RealTimeETL

Kafka JDBC Source Connector

```
name=psql-devops
connector.class=io.confluent.connect.jdbc.JdbcSourceConnector
tasks.max=10

connection.user=cloudera-scm
connection.password=$CLOUDERA_PASSWD
connection.url=jdbc:postgresql://cloudera-vm:7432/postgres?
user=cloudera-scm&password=$CLOUDERA_PASSWD&ssl=false
mode=incrementing
incrementing.column.name=id

table.types=TABLE
table.whitelist= tags, ratings

validate.non.null=false

topic.prefix=psql-m20-
```

Listing B.5: kafka JDBC Connector

Configurazione Pipeline Kafka

```
bootstrap.servers=localhost:9092

key.converter=org.apache.kafka.connect.json.JsonConverter
key.converter.schemas.enable=true
```

```
value.converter=org.apache.kafka.connect.json.JsonConverter
value.converter.schemas.enable=true

internal.key.converter=org.apache.kafka.connect.json.JsonConverter
internal.value.converter=org.apache.kafka.connect.json.JsonConverter
internal.key.converter.schemas.enable=false
internal.value.converter.schemas.enable=false

offset.storage.file.filename=/opt/connectm20.offsets
```

Listing B.6: Configurazione della pipeline Kafka

Transformation

```
package it.reply.data.pasquali.engine

import it.reply.data.pasquali.model.TransformedDFs
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.{DataFrame, SparkSession}

object ETL {

  def transformTags(jsonDF : DataFrame) : TransformedDFs = {

    val cols = Seq("id", "userid", "movieid", "tag", "timestamp")
    val colsKudu = Seq("userid", "movieid", "tag", "time")

    val toHive = jsonDF
```

```
  .withColumn("id", jsonDF("payload.id"))
  .withColumn("userid", jsonDF("payload.userid"))
  .withColumn("movieid", jsonDF("payload.movieid"))
  .withColumn("tag", jsonDF("payload.tag"))
  .withColumn("timestamp", jsonDF("payload.timestamp"))
  .select(cols.head, cols.tail: _*)

  val toKudu = toHive.toDF("id", "userid", "movieid", "tag", "time")
  .select(colsKudu.head, colsKudu.tail: _*)

  TransformedDFs(toHive, toKudu)
}

def transformRatings(jsonDF : DataFrame) : TransformedDFs = {

  val cols = Seq("id", "userid", "movieid", "rating", "timestamp")
  val colsKudu = Seq("userid", "movieid", "rating", "time")

  val toHive = jsonDF
  .withColumn("id", jsonDF("payload.id"))
  .withColumn("userid", jsonDF("payload.userid"))
  .withColumn("movieid", jsonDF("payload.movieid"))
  .withColumn("rating", jsonDF("payload.rating"))
  .withColumn("timestamp", jsonDF("payload.timestamp"))
  .select(cols.head, cols.tail: _*)

  val toKudu = toHive.toDF("id", "userid", "movieid", "rating", "time")
  .select(colsKudu.head, colsKudu.tail: _*)
```

```
    TransformedDFs(toHive, toKudu)
  }

  def transformRDD(jsonStringRDD : RDD[String],
    spark : SparkSession,
    tableName : String) : TransformedDFs = {

    val jsonDF = spark.sqlContext.jsonRDD(jsonStringRDD)

    tableName match {
      case "tags" => ETL.transformTags(jsonDF)
      case "ratings" => ETL.transformRatings(jsonDF)
    }
  }
}
```

Listing B.7: Real Time Transformation

La case class `TransformedDFs` non è altro che un wrapper di due DataFrame, uno destinato al Data Lake e l'altro al il Data Mart.

Spark Streaming Sink Connector

```
package it.reply.data.pasquali

import com.typesafe.config.ConfigFactory
import it.reply.data.pasquali.engine.ETL
import it.reply.data.pasquali.model.TransformedDFs
import kafka.serializer.StringDecoder
import org.apache.spark.sql.SparkSession
```

```
import org.apache.spark.streaming.kafka.KafkaUtils
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.streaming.{Seconds, StreamingContext}

object StreamMono {

  var toHive = true
  var onlyDebug = false
  var conf : SparkConf = null
  var sc : SparkContext = null
  var ssc : StreamingContext = null
  var spark : SparkSession = null
  var storage : Storage = null
  var topics : Set[String] = Set()
  var kafkaParams : Map[String, String] = null

  def main(args: Array[String]): Unit = {

    // Caricamento della configurazione da application.conf

    val KAFKA_BOOTSTRAP_ADDR = "localhost"
    val KAFKA_BOOTSTRAP_PORT = "9092"
    val KAFKA_GROUP = "devops"
    val KUDU_ADDR = "cloudera-vm"
    val KUDU_PORT = "7051"

    val SPARK_APPNAME = "Real Time ETL"
    val SPARK_MASTER = "local"
```

```
val KUDU_DATABASE = "datamart"
val HIVE_DATABASE = "datalake"
val KUDU_TABLE_BASE = "impala::"

storage = Storage()
.init(SPARK_MASTER, SPARK_MASTER, true)
.initKudu(KUDU_ADDR, KUDU_PORT, KUDU_TABLE_BASE)

val offset = args(1)
val singleTopic = args(0)

val mode = if(offset.contains("largest")) "append" else "overwrite"

initStreaming(SPARK_APPNAME,
SPARK_MASTER,
10,
KAFKA_BOOTSTRAP_ADDR,
KAFKA_BOOTSTRAP_PORT,
offset,
KAFKA_GROUP,
singleTopic)

val spark = storage.spark
val tableName = args(0).split("-")(2)

val messages = KafkaUtils.createDirectStream[String, String,
   StringDecoder, StringDecoder](
ssc, kafkaParams, topics)
```

```
messages.foreachRDD( rdd => {

  if(rdd.isEmpty){
    println("[ INFO ] Empty RDD")
  }
  else{

    val stringRDD = rdd.map(entry => entry._2)
    val dfs : TransformedDFs = ETL.transformRDD(stringRDD, spark,
        tableName)

    dfs.toHive.printSchema()
    dfs.toKudu.printSchema()

    if(onlyDebug){
      dfs.toHive.show()
      dfs.toKudu.show()
    }
    else{
      println("[ INFO ] ====== Save To Hive Data Lake ======")
      storage.writeDFtoHive(dfs.toHive, mode, HIVE_DATABASE, tableName)

      println("[ INFO ] ====== Save To Kudu Data Mart ======")
      storage.upsertKuduRows(dfs.toKudu,
          s"${KUDU_DATABASE}.${tableName}")
    }
  }
})
ssc.start()
```

```
    ssc.awaitTermination()
  }

  def initStreaming(appName : String,
    master : String,
    fetchIntervalSec : Int,
    bootstrapServer : String,
    bootstrapPort : String,
    offset : String,
    groupID : String,
    singleTopic : String): Unit ={

    conf = new SparkConf().setMaster(master).setAppName(appName)
    sc = SparkContext.getOrCreate(conf)
    ssc = new StreamingContext(sc, Seconds(fetchIntervalSec))
    ssc.checkpoint("checkpoint")

    this.topics = Set[String](singleTopic)

    kafkaParams = Map[String, String](
    "bootstrap.servers" -> s"${bootstrapServer}:${bootstrapPort}",
    "auto.offset.reset" -> offset,
    "group.id" -> groupID)
  }
}
```

Listing B.8: Real Time Transformation

B.2.4 Analysis

MRSpark2

```
object Main {

  def main(args: Array[String]): Unit = {

    // Caricamento della configurazione da application.conf

    val SPARK_APPNAME = "Batch ML"
    val SPARK_MASTER = "local[*]"

    val MODEL_PATH = "model/m20Model"
    val MODEL_ARCHIVE_PATH = "model/m20Model.zip"
    val TEST_FRACTION = 0.2
    val TRAIN_FRACTION = 0.8

    val KUDU_ADDRESS = "cloudera-vm"
    val KUDU_PORT = "7051"
    val KUDU_RATINGS_TABLE = "ratings"
    val KUDU_DATABASE = "datamart"
    val KUDU_TABLE_BASE = "impala::"

    var rank = 10
    var loops = 10
    var lambda = 0.1

    if(args.nonEmpty)
    {
```

```
  rank = args(0).toInt
  loops = args(1).toInt
  lambda = args(2).toDouble
}

val spark = SparkSession.builder()
.appName(SPARK_APPNAME)
.master(SPARK_MASTER)
.getOrCreate()

val sc = spark.sparkContext

val storage = Storage()
.init(SPARK_MASTER, SPARK_APPNAME, false)

storage.initKudu(KUDU_ADDRESS, KUDU_PORT, KUDU_TABLE_BASE)

val ratings = storage.readKuduTable(s"${KUDU_DATABASE}.
  ${KUDU_RATINGS_TABLE}").rdd

//Split del dataset in Test e Train Set
val Array(rawTrain, rawTest) =
    ratings.randomSplit(Array(TRAIN_FRACTION, TEST_FRACTION))

val testSet = rawTest.map{ case Row(userID, movieID, rating, time) =>
Rating(userID.asInstanceOf[Long].toInt,
movieID.asInstanceOf[Long].toInt,
rating.asInstanceOf[Double])}
```

```
    val trainSet = rawTrain.map{ case Row(userID, movieID, rating, time)
        =>
    Rating(userID.asInstanceOf[Long].toInt,
    movieID.asInstanceOf[Long].toInt,
    rating.asInstanceOf[Double])}

    // Train del Modello
    val mr = MovieRecommender()
    .initSpark(spark)
    .trainModel(trainSet, rank, loops, lambda)

    //Valuta il Modello
    val mse = mr.evaluateModel_MSE(testSet)

    println(s"Actual MSE is ${mse}")

    mr.storeModel(MODEL_PATH)
    storage.zipModel(MODEL_PATH, MODEL_ARCHIVE_PATH)

    Try(Path(MODEL_PATH).deleteRecursively())

    spark.stop()
  }
}
```

Listing B.9: Computazione e Verifica del modello

Endpoint Scalatra

```
get("/see") {
    if(collabModel == null)
        initSpark()

    val user = params("inputUser").toInt
    val movie = params("inputMovie").toInt

    val rating = params("inputRating")
    if(rating.contains("Just See"))
        redirect(url(s"/see/$user/$movie"))
    else
        redirect(url(s"/see/$user/$movie/${rating}"))
}
```

Listing B.10: Esempio di Endpoint Scalatra

B.2.5 Storage

```
package it.reply.data.pasquali

import java.io.File

import com.typesafe.config.ConfigFactory
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.fs.{FileSystem, Path}
import org.apache.kudu.spark.kudu._
import org.apache.spark.sql._
import org.zeroturnaround.zip.ZipUtil
```

```
import scala.sys.process._

case class Storage() {

  val defaultThrift : String = "thrift://localhost:9083"

  var kuduMaster : String =
      "cloudera-vm.c.endless-upgrade-187216.internal"
  var kuduPort : String = "7051"
  var KUDU = ""

  var KUDU_TABLE_BASE = ""

  var hdfsServer : String =
      "cloudera-vm.c.endless-upgrade-187216.internal"
  var hdfsPort : String = "8020"
  val HDFS = s"${hdfsServer}:${hdfsPort}"

  var kuduContext : KuduContext = null
  var spark : SparkSession = null

  var hdfs : FileSystem = null



  def init(withHive : Boolean) : Storage = {
    init("local[*]", "Movie Recommender", withHive)
  }
```

```
def init(master : String, appName : String, withHive: Boolean) :
    Storage = {

  if(withHive)
    spark = SparkSession
      .builder()
      .master(master)
      .appName(appName)
      .enableHiveSupport()
      .getOrCreate()
  else
    spark = SparkSession
    .builder()
    .master(master)
    .appName(appName)
    .getOrCreate()
  this
}

def init(master : String, appName : String, hiveThriftServer: String,
    hiveThriftPort : String) : Storage = {

  spark = SparkSession
    .builder()
    .master(master)
    .appName(appName)
    .config("hive.metastore.uris",
       s"thrift://${hiveThriftServer}:${hiveThriftPort}")
    .enableHiveSupport().getOrCreate()
```

```
  this
}

def initKudu(master : String, port : String, baseTable : String) :
    Storage = {

  kuduMaster = master
  kuduPort = port
  KUDU = s"${kuduMaster}:${kuduPort}"
  kuduContext = new KuduContext(KUDU, spark.sparkContext)
  KUDU_TABLE_BASE = baseTable

  this
}

def initHDFS(server : String, port : String) : Storage = {

  hdfsServer = server
  hdfsPort = port

  val hadoopConfig = new Configuration()
  hadoopConfig.set("fs.defaultFS", HDFS)
  hdfs = FileSystem.get(hadoopConfig)

  this
}

def upsertKuduRows(rows : DataFrame, table : String) : Unit = {
```

```
  val tableName = s"${KUDU_TABLE_BASE}${table}"
  kuduContext.upsertRows(rows, tableName)

}

def updateKuduRows(rows : DataFrame, table : String) : Unit = {

  val tableName = s"${KUDU_TABLE_BASE}${table}"
  kuduContext.updateRows(rows, tableName)

}

def deleteKuduRows(keys : DataFrame, table : String) : Unit = {

  val tableName = s"${KUDU_TABLE_BASE}${table}"
  kuduContext.deleteRows(keys, tableName)

}

def readKuduTable(kuduTestTable: String): DataFrame ={

  val df = spark.sqlContext.read.options(
    Map(
      "kudu.master" -> KUDU,
      "kudu.table" -> s"${KUDU_TABLE_BASE}${kuduTestTable}"
    )
  ).kudu
```

```
  df
}

def storeHDFSFile(filePath : String, fileName : String, data :
    Array[Byte]) : Unit = {

  val path = new Path(filePath)

  if(!hdfs.exists(path))
    hdfs.mkdirs(path)

  val fullPath = new Path(filePath+"/"+fileName)

  hdfs.create(fullPath).write(data)
}

def readHDFSFile(filePath : String) : Array[Byte] = {
  return null
}

def readHiveTable(table : String) : DataFrame = {
  spark.sql(s"select * from $table")
}

def runHiveQuery(query : String) : DataFrame = {
  spark.sql(query)
}
```

```
def writeDFtoHive(df : DataFrame, mode : String, db : String, table :
    String) : Unit = {
  df.write.mode(mode).saveAsTable(s"${db}.${table}")
}

def remoteSecureCopy(inputFile : String,
                    remoteUser : String, remoteHost : String, remotePath
                        : String) : String = {

  s"scp -v ${inputFile} ${remoteHost}@${remoteHost}:/${remotePath}" !!
}

def zipModel(sourceDir : String, outName : String) : Unit = {
  ZipUtil.pack(new File(sourceDir), new File(outName))
}

def unzipModel(zipFile : String, outDir : String) : Unit = {
  ZipUtil.unpack(new File(zipFile), new File(outDir))
}

def closeSession() : Unit = {

  if(hdfs != null){
    hdfs.close()
    hdfs = null
  }

  if(spark != null) {
    spark.close()
```

```
      spark = null
    }


    kuduContext = null
  }
}
```

Listing B.11: Libreria DevOpsStorage di supporto

Appendice C

C.1 Configuration Management

Cloudera VM

```
# tasks file for cloudera

- name: Config Environment
  include_role:
    name: cm-env-setup

- name: Setup repository
  include_role:
    name: cm-repo

- name: Install Oracle Java
  include_role:
    name: oracle-java

- name: Install Cloudera Manager with Path B
  include_role:
```

```
    name: cm-install-path-b


- name: Wait for cloudera to start up before proceeding.
  shell: "curl -D - --silent --max-time 5 http://localhost:7180/cmf"
  register: result
  until: (result.stdout.find("403 Forbidden") != -1) or
      (result.stdout.find("200 OK") != -1) and
      (result.stdout.find("Please wait while") == -1)
  retries: "60"
  delay: "10"
  changed_when: false
  check_mode: no
```

Listing C.1: Playbook role cloudera

DevOps Worker

```
# tasks file for devops-worker


- name: Install SBT
  include_role:
    name: sbt


- name: Install Spark
  include_role:
    name: spark


- name: Install Confluent
  include_role:
```

```
    name: confluent

- name: Setup Kafka JDBC Source connector
  include_role:
    name: kafka-jdbc-connector

- name: Set hive-site.xml
  copy:
    src: hive-site.xml
    dest: /opt/spark-2.2.0-bin-hadoop2.7/conf/hive-site.xml
    owner: root
    group: root
    mode: 0644
  become: true
  become_flags: '-i'

- name: setup folders
  include_tasks: folders.yml

- name: setup services
  include_tasks: services.yml
```

Listing C.2: Playbook role devops-worker

Jenkins

```
# tasks file for jenkins

- name: Define jenkins_repo_url
```

```
  set_fact:
    jenkins_repo_url: "{{ __jenkins_repo_url }}"
  when: jenkins_repo_url is not defined

- name: Define jenkins_repo_key_url
  set_fact:
    jenkins_repo_key_url: "{{ __jenkins_repo_key_url }}"
  when: jenkins_repo_key_url is not defined

- name: Define jenkins_pkg_url
  set_fact:
    jenkins_pkg_url: "{{ __jenkins_pkg_url }}"
  when: jenkins_pkg_url is not defined

# Setup/install tasks.
- include_tasks: setup.yml

- name: Create firewall HTTP exception to allow web GUI
  firewalld:
    service: http
    permanent: true
    state: enabled

- name: Create firewall exception for port 8080/tcp
  firewalld:
    port: "{{ jenkins_http_port }}/tcp"
    permanent: true
    state: enabled
```

```
# Configure Jenkins init settings.
- include_tasks: settings.yml

# Make sure Jenkins starts, then configure Jenkins.
- name: Ensure Jenkins is started and runs on startup.
  service: name=jenkins state=started enabled=yes

- name: Wait for Jenkins to start up before proceeding.
  shell: "curl -D - --silent --max-time 5 http://{{ jenkins_hostname
      }}:{{ jenkins_http_port }}{{ jenkins_url_prefix }}/cli/"
  register: result
  until: (result.stdout.find("403 Forbidden") != -1) or
      (result.stdout.find("200 OK") != -1) and
      (result.stdout.find("Please wait while") == -1)
  retries: "{{ jenkins_connection_retries }}"
  delay: "{{ jenkins_connection_delay }}"
  changed_when: false
  check_mode: no

- name: Get the jenkins-cli jarfile from the Jenkins server.
  get_url:
    url: "http://{{ jenkins_hostname }}:{{ jenkins_http_port }}{{
        jenkins_url_prefix }}/jnlpJars/jenkins-cli.jar"
    dest: "{{ jenkins_jar_location }}"
  register: jarfile_get
  until: "'OK' in jarfile_get.msg or 'file already exists' in
      jarfile_get.msg"
  retries: 5
  delay: 10
```

```
  check_mode: no



# Update Jenkins and install configured plugins.
- include_tasks: plugins.yml

# Make Jenkins a sudoers to run pipelines without any trouble
- include_tasks: sudoers.yml
```

Listing C.3: Playbook role jenkins

C.2 Infrastructure As a Code

Orchestration e Provisioning di Cloudera VM

```
provider "google" {
    credentials = "${file("terraform-admin.json")}"
    project     = "endless-upgrade"
    region      = "us-east1-b"
}

resource "google_compute_instance" "worker" {
    project = "endless-upgrade"
    zone = "us-east1-b"
    name = "cloudera-vm"
    machine_type = "n1-highmem-4"

    tags = ["coudera", "http", "https"]
```

```
    boot_disk {
        initialize_params {
            image = "centos-7-v20171213"
            size = "35"
        }
    }

    network_interface {
        network = "default"
        access_config {
        }
    }

    provisioner "local-exec" {
        command = "sleep 90; ansible-playbook -i
            '${google_compute_instance.worker.name},'
            --private-key=~/.ssh/ansible_rsa $ANSIBLE_HOME/cloudera.yml -e
            'ansible_ssh_user=dario_pasquali93' -e
            'host_key_checking=False'"
    }
}

output "worker" {
    value = "${google_compute_instance.worker.self_link}"
}
```

Listing C.4: Configurazione Terraform Cloudera VM

C.3 Continuous Integration

SBT

```
name := "BatchETL2"
version := "2.1"
scalaVersion := "2.11.8"



resolvers ++= Seq(
    "All Spark Repository -> bintray-spark-packages" at
        "https://dl.bintray.com/spark-packages/maven/"
)

libraryDependencies ++= Seq(

    "org.apache.spark" % "spark-core_2.11" % "2.2.0",
    "org.apache.spark" % "spark-sql_2.11" % "2.2.0",
    "org.apache.hadoop" % "hadoop-common" % "2.7.0",
    "org.apache.spark" % "spark-hive_2.11" % "2.2.0",
    "org.apache.spark" % "spark-yarn_2.11" % "2.2.0",
    "org.apache.kudu" % "kudu-spark2_2.11" % "1.5.0"
)

libraryDependencies ++= Seq(

    "com.typesafe" % "config" % "1.3.2",
    "org.scalatest" %% "scalatest" % "2.2.2" % "test"
)
```

```
libraryDependencies ++= Seq(

    "io.prometheus" % "simpleclient" % "0.1.0",
    "io.prometheus" % "simpleclient_common" % "0.1.0",
    "io.prometheus" % "simpleclient_hotspot" % "0.1.0",
    "io.prometheus" % "simpleclient_pushgateway" % "0.1.0",
)



assemblyMergeStrategy in assembly := {
    case PathList("META-INF", xs @ _*) => MergeStrategy.discard
    case x => MergeStrategy.first
}

test in assembly := {}
```

Listing C.5: Dipendenze Batch ETL

Pipeline di CI per BatchETL

```
pipeline {
    agent any

    stages {
        stage('Setup Env') {
            steps {
                echo 'Setup the system'
                echo 'wget, curl, java, sbt and spark are now installed by
                    Config Management system :)'
```

```
        }
    }
    stage('Env setup test') {
        steps {
            sh 'java -version'
            sh 'sbt about'
        }
    }
    stage('Unit Tests') {
        steps {
            sh 'sbt clean coverage test coverageReport'
            archiveArtifacts 'target/test-reports/*.xml'
            archiveArtifacts 'target/scala-2.11/scoverage-report/*'
        }
    }
    stage('Build') {
        steps {
            sh 'sbt clean compile package assembly'
            archiveArtifacts(artifacts: 'target/scala-*/*.jar',
                fingerprint: true)
        }
    }
}
post {
    success {
        script {
            header = "Job <${env.JOB_URL}|${env.JOB_NAME}>
                <${env.JOB_DISPLAY_URL}|(Blue)>"
```

```
            header += " build
                <${env.BUILD_URL}|${env.BUILD_DISPLAY_NAME}>
                <${env.RUN_DISPLAY_URL}|(Blue)>:"
            message = "${header}\n :smiley: New Batch ETL Build
                Success, make a Pull Request if you want to merge with
                Master."

            author = sh(script: "git log -1 --pretty=%an",
                returnStdout: true).trim()
            commitMessage = sh(script: "git log -1 --pretty=%B",
                returnStdout: true).trim()
            message += " Commit by <@${author}> (${author}): ```
                ${commitMessage} ``` "
            color = '#00CC00'
        }
        slackSend(message: message, baseUrl:
            'https://devops-pasquali-cm.slack.com
        /services/hooks/jenkins-ci/', color: color, token:
            'ihoCVUPB7hqGz2xI1htD8x0F')
    }
    failure {
        script {
            header = "Job <${env.JOB_URL}|${env.JOB_NAME}>
                <${env.JOB_DISPLAY_URL}|(Blue)>"
            header += " build
                <${env.BUILD_URL}|${env.BUILD_DISPLAY_NAME}>
                <${env.RUN_DISPLAY_URL}|(Blue)>:"
            message = "${header}\nThe Build Failed.\n"
```

```
                author = sh(script: "git log -1 --pretty=%an",
                    returnStdout: true).trim()
                commitMessage = sh(script: "git log -1 --pretty=%B",
                    returnStdout: true).trim()
                message += " Commit by <@${author}> (${author}): ```
                    ${commitMessage} ``` "
                color = '#990000'
            }
            slackSend(message: message, baseUrl:
                'https://devops-pasquali-cm.slack.com
            /services/hooks/jenkins-ci/', color: color, token:
                'ihoCVUPB7hqGz2xI1htD8x0F')
        }
    }
}
```

Listing C.6: Jenkinsfile CI

C.4 Continuous Testing

Specifica Test di MarSpark2

```
class MRSpec
extends FlatSpec
with Matchers
with OptionValues
with Inside
with Inspectors
with BeforeAndAfterAll{
```

```
var mr : MovieRecommender = null
var config : Config = null

var CONF_DIR = ""
var CONFIG_FILE = "BatchML_test.conf"



"The movie recommender"
    must "be instantiated with given parameters" in {
    val SPARK_APPNAME = config.getString("bml.spark.app_name")
    val SPARK_MASTER = config.getString("bml.spark.master")

    mr = MovieRecommender().initSpark(SPARK_APPNAME, SPARK_MASTER)

    assert(mr.spark != null)
    assert(mr.spark.sparkContext.master == "local[*]")
    assert(mr.sc != null)
}

it must "compute a valid model given input ratings" in {

    val timer = gaugeDuration.startTimer()

    val rawTrain = mr.sc.parallelize(Seq(

    (1 ,1, 1, 5, "time"),
    (2 ,1, 2, 5, "time"),
```

```
(3 ,2, 1, 5, "time"),
(4 ,2, 3, 5, "time"),
(5 ,2, 4, 5, "time"),
(6 ,2, 5, 0, "time"),
(7 ,2, 6, 5, "time"),

(8 ,4, 1, 5, "time"),
(9 ,4, 2, 5, "time"),
(10 ,4, 4, 5, "time"),
(11 ,4, 5, 0, "time"),
(12 ,4, 6, 5, "time"),

(13 ,5, 1, 5, "time"),
(14 ,5, 2, 5, "time"),
(15 ,5, 4, 5, "time"),
(16 ,5, 5, 0, "time"),
(17 ,5, 6, 5, "time"),

(18 ,6, 1, 5, "time"),
(19 ,6, 3, 5, "time"),
(20 ,6, 4, 5, "time"),
(21 ,6, 5, 0, "time")
))

val trainSet = rawTrain.map{
    case (userID, movieID, rating, time) =>
        Rating(userID, movieID, rating.toDouble)
}
```

```
    mr.trainModel(trainSet, 10, 10, 0.1)

    assert(mr.model != null)
}

"The computed model"
    should "estimate a new entries with good precision" in {

    val rawTest = mr.sc.parallelize(
        Seq(
        (1, 5, 0, "time"),
        (1, 6, 5, "time")
        )
    )

    val testSet = rawTest.map{ case (userID, movieID, rating, time) =>
        Rating(userID, movieID, rating.toDouble)}

    val mse = mr.evaluateModel_MSE(testSet)

    assert(mse < 0.01)
}

it should "can be saved in zip format and retrieved" in {

    val MODEL_PATH = config.getString("bml.recommender.model_path")
    val MODEL_ARCHIVE_PATH =
        config.getString("bml.recommender.model_archive_path")
```

```
    if(new File(MODEL_PATH).exists())
    {
        Path(MODEL_PATH).deleteRecursively()
    }

    if(new File(MODEL_ARCHIVE_PATH).exists())
    {
        Path(MODEL_ARCHIVE_PATH).delete()
    }

    mr.storeModel(MODEL_PATH)
    assert(new File(MODEL_PATH).exists)
    mr.model = null

    val storage = Storage()
    storage.zipModel(MODEL_PATH, MODEL_ARCHIVE_PATH)
    assert(new File(MODEL_ARCHIVE_PATH).exists)

    Path(MODEL_PATH).deleteRecursively()

    storage.unzipModel(MODEL_ARCHIVE_PATH, MODEL_PATH)
    mr.loadModel(MODEL_PATH)
    assert(mr.model != null)

    Path(MODEL_PATH).deleteRecursively()
    }
}
```

Listing C.7: Specifica dei Test per MRSpark2

Integration Test

```
beforeAll(){

    Carica i toggle dal file di configurazione

    Carica la configurazione di Batch ETL
    Carica la configurazione di Batch ML
    Carica la configurazione di Real Time ETL
    Carica la configurazione di Real Time ML

    Inizializza Spark tramite Storage
    Inizializza la connessione a Kudu tramite Storage
    Inizializza la connessione ad Hive tramite Storage

    Avvia Confluent Schema Registry
    Avvia il Kafka JDBC Source Connector

    Attendi che il topic sia popolato
}

//Smoke Test Spark -------------------------------------
if( toggleSpark == True ){
    Verifica che la Spark Session sia stata creata e sia attiva
}

//Smoke Test Sqoop -------------------------------------
if( toggleSqoop == True ){
```

```
    Verifica che la tabella movies esista nel datalake_test e contenga
        dati
    Verifica che la tabella links esista nel datalake_test e contenga dati
}

//Smoke Test Kafka JDBC Connector -----------------------
if( toggleKafka == True ){
    Esegui un Kafka Console Consumer per psql-m20-ratings_test
    Verifica che il Kafka JDBC Connector abbia creato il topic
        psql-m20-ratings_test e che contenga dati
    Termina il Kafka Console Consumer

    Esegui un Kafka Console Consumer per psql-m20-tags_test
    Verifica che il Kafka JDBC Connector abbia creato il topic
        psql-m20-tags_test
    e che contenga dati
    Termina il Kafka Console Consumer
}

//Test Processo Batch ETL -------------------------------
if( toggleBatchETL == True ) {
    Spark Submit del Job Batch ETL
    Attendi la terminazione del processo
    Verifica che la struttura dati movies_enriched esista nel
        datamart_test
    Verifica che essa contenga un movie sample
}

//Test Processo Real Time ETL ---------------------------
```

```
if( toggleRealTimeETL == True ){
    Spark Submit del Job Real Time ETL per i ratings
    Attesa della elaborazione dei dati nei topics di test
    Termina il Job

    Spark Submit del Job Real Time ETL per i tags
    Attesa della elaborazione dei dati nei topics di test
    Termina il Job

    Verifica che la tabella ratings esista nel datalake_test
        e che contenga il rating sample nel formato corretto
    Verifica che la tabella ratings esista nel datamart_test
        e che contenga il rating sample nel formato corretto

    Verifica che la tabella tags esista nel datalake_test
        e che contenga il tag sample nel formato corretto
    Verifica che la tabella tags esista nel datamart_test
        e che contenga il tag sample nel formato corretto
}

//Test Processo Batch ML ---------------------------
if( toggleBatchML == True ){
    Spark Submit del Job Batch ML
    Attendi la fine del processo
    Verifica che il modello sia stato salvato nella directory corretta
    Verifica che il MSE del modello sia valido
}

//Test Processo Real Time ML -------------------------
```

```
if( toggleRealTimeML == True ){
    Avvia il servizio di Real Time ML
    Attendi che il servizio sia online
    Verifica che il servizio sia online
    Verifica che computi una valida raccomandazione
    Termina il servizio
}

afterAll() {
    if( toggleRealTimeETL == True ){
        Elimina la tabella ratings dal datalake_test
        Elimina la tabella tags dal datalake_test

        Tronca la tabella ratings nel datamart_test
        Tronca la tabelle tags nel datamart_test
    }

    if( toggleBatchETL == True ){
        Tronca la tabella enriched_movies nel datamart_test
    }

    if( toggleBatchML == True ){
        Elimina il modello computato
    }

    Chiudi la SparkSession

    Termina il Kafka JDBC Source Connector
```

```
    Termina Confluent Kafka distruggendo i topics


    Elimina il file di offset di Kafka
}
```

Listing C.8: IntegrationStagingProject

Staging Deploy

```
pipeline{
    ...
    stages{
        ...
        stage('Staging Deploy') {
            steps {
                sh 'sudo cp target/*/*assembly*.jar /opt/deploy/batch_etl'
                sh 'sudo cp conf/* /opt/deploy/batch_etl'
                sh 'sudo cp target/*/*assembly*.jar
                    /opt/staging/IntegrationStagingProject/lib'
            }
        }
        stage('Integration Tests') {
            steps {
                sh 'cd /opt/staging/IntegrationStagingProject/ && sbt
                    clean test'
            }
        }
    }
    ...
```

```
}
```

Listing C.9: Staging Test nella Pipeline

C.5 Continuous Delivery

CD nella pipeline di MRSpark2

```
failMessage = ""

pipeline {
    ...
    environment {
        DEPLOY_TARGET = 'devops-worker'
    }
    stages {
        ...
        stage('Deploy ?') {
            steps {
                script {
                    header = "Job <${env.JOB_URL}|${env.JOB_NAME}>
                        <${env.JOB_URL}|${env.BRANCH_NAME}>
                        <${env.JOB_DISPLAY_URL}|(Blue)>"
                    header += " build
                        <${env.BUILD_URL}|${env.BUILD_DISPLAY_NAME}>
                        <${env.RUN_DISPLAY_URL}|(Blue)>:"
                    message = "${header}\n"
```

```
author = sh(script: "git log -1 --pretty=%an",
    returnStdout: true).trim()
commitMessage = sh(script: "git log -1 --pretty=%B",
    returnStdout: true).trim()
message += " Commit by <@${author}> (${author}): ‘‘‘
    ${commitMessage} ‘‘‘ "
message += "-----------------------------------"
message += "\nThe new Batch ML commit pass Unit and
    Integration tests"
message += "\nThis session will be available for 60
    second, make a CHOICE!"
message += "\nPlease <${env.RUN_DISPLAY_URL}|Manual
    Deploy> it if you want!"
color = '#36ABCC'
slackSend(message: message, baseUrl:
    'https://devops-pasquali-cm.slack.com/
services/hooks/jenkins-ci/', color: color, token:
    'ihoCVUPB7hqGz2xI1htD8x0F')

try {
    timeout(time: 60, unit: 'SECONDS') {
        userInput = input(id: 'DeployBML', message:
            'Deploy in Production??')
    }
}
catch(err) {
    failMessage = "Deploy session expired or aborted"
    error("Deploy session expired or aborted")
}
```

```
            }
        }
    }
    stage('Production Deploy') {
        steps {
            echo 'Safe to Deploy in Production, Great Job :D'
            sh "sudo ansible-playbook -i \'${DEPLOY_TARGET},\'
                --private-key=/home/xxpasquxx/.ssh/ansible_rsa_key
                /opt/DevOpsProduction-Orchestrator
            /ansible/deploy/batch_ml_deploy.yml -e
                \'ansible_ssh_user=xxpasquxx\' -e
                \'host_key_checking=False\'"
        }
    }
}
...
}
```

Listing C.10: jenkinsfile di Continuous Deployment per MRSpark2

C.6 Continuous Deployment

Deploy di BatchETL

```
- hosts: all
  become: yes
  become_user: root
  tasks:
    - name: Deploy Batch ETL Process in Production
```

```
      copy:
      src: /opt/deploy/batch_etl/
      dest: /opt/devops/batch_etl/
      owner: root
      group: root
      mode: 0777
      force: yes
```

Listing C.11: Playbook deploy Batch

Deploy di RealTimeETL

```
- hosts: all
  become: yes
  become_user: root
  tasks:
    - name: Ensure Real Time ETL services are down
        service:
        name: "{{ item }}"
        state: stopped
      with_items:
      - devops-rtetl-ratings
      - devops-rtetl-tags

    - name: Deploy Real Time ETL in Production
      copy:
        src: /opt/deploy/realtime_etl/
        dest: /opt/devops/realtime_etl/
        owner: root
```

```
      group: root
      mode: 0777
      force: yes

  - name: Ensure Kafka JDBC connector is running
    service:
      name: devops-kafka-jdbc-connector
      state: started
      enabled: yes

  - name: Ensure Real Time ETL services are running and start on boot
    service:
      name: "{{ item }}"
      state: started
      enabled: yes
    with_items:
    - devops-rtetl-ratings
    - devops-rtrtl-tags
```

Listing C.12: Playbook deploy Real Time

C.7 Continuous Monitoring

Check dello stato di un servizio

```
- hosts: all
  become: yes
  become_user: root
  tasks:
```

```
  - name: Ensure {{ service_pretty }} is running and starts on boot
    service:
      name: "{{ service }}"
      state: started
      enabled: yes
```

Listing C.13: Playbook parametrizzato per testare lo stato di un servizio

Eseguito con:

```
ansible-playbook -i 'devops-worker,'
    --private-key=/home/xxpasquxx/.ssh/ansible_rsa_key
    /opt/DevOpsProduction-Orchestrator/ansible/test/test-service.yml -e
    'ansible_ssh_user=xxpasquxx' -e 'host_key_checking=False'
    --extra-vars "service_pretty=RealTimeMovieRec service=devops-rtml"
```

Listing C.14: Esecuzione di un Playbook Ansible parametrizzato

Check dello stato di un nodo

```
ansible -i 'devops-worker,'
    --private-key=/home/xxpasquxx/.ssh/ansible_rsa_key -e
    'ansible_ssh_user=xxpasquxx' -e 'host_key_checking=False' -m ping


devops-worker | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Listing C.15: Ping dello stato di un'istanza

Estrazione metriche MRSpark2

```
object Main {


    def main(args: Array[String]): Unit = {
        val ENV = configuration.getString("bml.metrics.environment")
        val JOB_NAME = configuration.getString("bml.metrics.job_name")
        val GATEWAY_ADDR =
            configuration.getString("bml.metrics.gateway.address")
        val GATEWAY_PORT =
            configuration.getString("bml.metrics.gateway.port")
        val LABEL_MSE =
            s"${configuration.getString("bml.metrics.labels.mse")}"
        val pushGateway : PushGateway = new
            PushGateway(s"$GATEWAY_ADDR:$GATEWAY_PORT")
        val registry = new CollectorRegistry


        val gaugeMSE : Gauge = Gauge.build().name(LABEL_MSE)
        .help("Mean Squared Error of the model").register(registry)


        [... Train, Test, Store ...]


        gaugeMSE.set(mse)
        pushGateway.push(registry, s"${ENV}_${JOB_NAME}")
    }
}
```

Listing C.16: Metriche White Box in MRSpark2

Bibliografia

[1] *Alternating Least Square*. URL: https://datasciencemadesimpler.wordpress.com/tag/alternating-least-squares/.

[2] *Blue Ocean*. URL: https://jenkins.io/projects/blueocean/.

[3] *Change data capture*. URL: https://en.wikipedia.org/wiki/Change_data_capture.

[4] *Chef, Configuration Management Tool*. URL: https://www.chef.io/chef/.

[5] *DevOpsProduction - Orchestrator*. URL: https://github.com/dpasqualiReply/DevOpsProduction-Orchestrator/tree/master/ansible.

[6] *Gartner*. URL: https://www.gartner.com/it-glossary/big-data.

[7] *Grafana*. URL: https://grafana.com/.

[8] *Grouplens - Social Computing Research at the University of Minnesota*. URL: https://grouplens.org/.

[9] Jason Hand. «ChatOps – Managing Operations in Group Chat». In: (). URL: https://victorops.com/ebooks/chatops-managing-operations-group-chat/.

[10] F. Maxwell Harper e Joseph A. Konstan. «The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 ». In: (2015). URL: https://kudu.apache.org/kudu.pdf.

[11] *HashiCorp Configuration Language*. URL: https://github.com/hashicorp/hcl.

[12] Charlie Swift Steven Edwards Lucy Bishop Ed Pearcey Tom Mugridge Simon Bell Rosie Robinson James Manktelow Keith Jackson. *Lewin's Change Management Model*. URL: https://www.mindtools.com/pages/article/newPPM_94.htm.

[13] *Jenkins CI*. URL: https://jenkins.io/.

[14] John Riedl Jesse Vig Shilad Sen. «The Tag Genome: Encoding Community Knowledge to Support Novel Interaction». In: (). URL: http://files.grouplens.org/papers/tag_genome.pdf.

[15] Gene Kim Kevin Beher George Spafford. *The Phoenix Project - A Novel About IT, DevOps, and Helping Your Business Win*. URL: https://itrevolution.com/book/the-phoenix-project/.

[16] *Metriche offerte da Prometheus*. URL: https://prometheus.io/docs/concepts/metric_types/.

[17] *Movielens*. URL: https://grouplens.org/datasets/movielens/20m/.

[18] Ruben Director product operations. *Our take on devops, part 1: The history lesson*. URL: https://novemberfive.co/blog/devops-history/.

[19] *Prometheus*. URL: https://prometheus.io/.

[20] *Puppet, Configuration Management Tool*. URL: https://puppet.com/.

[21] *sbt-assembly*. URL: https://github.com/sbt/sbt-assembly.

[22] *ScalaTest - simply productive*. URL: http://www.scalatest.org/.

[23] *Scalatra*. URL: http://scalatra.org.

[24] *SoundCloud*. URL: https://soundcloud.com/.

[25] *Spark Streaming + Kafka Integration*. URL: https://spark.apache.org/docs/2.2.0/streaming-kafka-0-8-integration.html.

[26] Dan Burkert Jean-Daniel Cryans Adar Dembo Mike Percy Silvius Rus Dave Wang Matteo Bertozzi Colin Patrick McCabe Andrew Wang Todd Lipcon David Alves. «Kudu: Storage for Fast Analytics on Fast Data». In: (2015). URL: https://kudu.apache.org/kudu.pdf.

[27] Marcel Kornacker Alexander Behm Victor Bittorf Taras Bobrovytsky Casey Ching Alan Choi Justin Erickson Martin Grund Daniel Hecht Matthew Jacobs Ishaan Joshi Lenni Kuff Dileep Kumar Alex Leblang Nong Li Ippokratis Pandis Henry Robinson David Rorke Silvius Rus John Russell Dimitris Tsirogiannis Skye Wanderman-Milne Michael Yoder. «Impala: A Modern, Open-Source SQL Engine for Hadoop». In: (). URL: http://cidrdb.org/cidr2015/Papers/CIDR15_Paper28.pdf.

Ringraziamenti

Vorrei ringraziare di cuore tutte le persone che mi hanno sostenuto e accompagnato in questo periodo di tesi e negli anni universitari.

Ringrazio il mio relatore, professor Claudio Sartori, per la sua disponibilità e il sostegno datomi durante la tesi.

Ringrazio il mio tutor aziendale, Marco Gatta, per avermi permesso di svolgere l'esperienza di tirocinio per tesi presso Data Reply, spero di poter continuare a lungo la collaborazione creata.

Ringrazio tutti i colleghi e amici di Data Reply, in particolare il team di Big Data Engineering per avermi accolto e guidato durante questa tesi. Un grazie speciale va a Michele, sempre disponibile per un consiglio o incoraggiamento, Alberto fondamentale punto di riferimento del primo periodo in azienda e infine "il Team di Annalisa": Giovanni, Eros, Andrea, Alessandro, Alberto e Annalisa, per le fantastiche giornate e serate passate insieme.

Ringrazio Federico e Paolo, senza i quali questi anni universitari non sarebbero stati gli stessi. Grazie a Filippo, Michele, Francesco e Federico per le serate passate oltre i confini della realtà. Grazie alla Cumpa per il loro costante sostegno.

Un ringraziamento particolare va ad Anna, presenza fondamentale di questo ultimo anno, la ringrazio per tutto l'affetto, il sostegno, la forza, la fiducia e il coraggio che mi ha dato e continua tutt'ora a darmi.

Infine voglio ringraziare la mia famiglia: i miei genitori, Elisabetta e Silvio, per avermi insegnato l'amore e la responsabilità, ed aver concretamente investito sul mio futuro, mio fratello Gabriele per avermi aiutato a superare i miei limiti, la Nonna Anna, dalla quale ho ereditato ingegno e creatività, e la Nonna Eva per avermi insegnato a non arrendermi.