ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Tesi di Laurea Magistrale in
Sistemi Distribuiti

Progetto e Valutazione di Pipeline di Big Data Analytics per Progetti di Sostenibilità

CANDIDATO
Lorenzo Piazza

RELATORE
Chiar.mo Prof. Ing. Paolo Bellavista

CORRELATORE
Dott. Ing. Alberto Cavalucci

Sessione IV

Anno Accademico 2019/2020

"*La perseveranza è ciò che rende l'impossibile possibile,*
il possibile probabile, e il probabile certo"
(Robert Half)

Indice

Introduzione

«Rafforzare i mezzi di attuazione e rinnovare il partenariato mondiale per lo sviluppo sostenibile»: è con questo obiettivo, il numero 17, che si conclude l'Agenda ONU 2030, un programma ratificato nel 2015 da tutti i 193 Paesi membri delle Nazioni Unite con lo scopo di perseguire un modello di sviluppo sostenibile a livello globale.

L'Agenda delinea una lista di 17 Sustainable Development Goal (SDG), obiettivi accomunati da un'idea di prosperità mondiale che sia sensibile alle questioni sociali ed ambientali. Tra questi figurano obiettivi molto ambiziosi come la lotta alla povertà, alle ineguaglianze sociali, l'eliminazione della fame nel mondo, il diritto all'educazione e il contrasto al cambiamento climatico.

L'obiettivo 17 sottolinea quanto il coinvolgimento dei governi, del settore privato e di tutti i civili sia indispensabile per raggiungere traguardi così alti, che senza una solida sinergia scadrebbero in ideali utopistici. È proprio sotto questa spinta che nasce il partenariato tra le aziende Hera, Conad e Camst e il progetto Hera SDG, all'interno del quale si inserisce il mio lavoro di tesi.

Hera SDG si propone di creare una Smart Sustainable Community in cui vengano incentivati i comportamenti virtuosi e sostenibili dei cittadini attraverso un meccanismo di ricompensa.

Il progetto viene supportato e reso possibile dalla presenza di un'infrastruttura IT sottostante (applicazioni mobili, siti web, dispositivi IoT) che permetta di tracciare i comportamenti effettuati degli utenti aderenti alla Community e di assegnare automaticamente le ricompense in *token* virtuali. Sempre grazie all'infrastruttura IT, gli utenti avranno la possibilità di spendere presso i marketplace delle aziende Hera, Conad e Camst i token guadagnati.

La Community Hera SDG, quindi è una potenziale sorgente di grandi quantità di dati che, se raccolti, gestiti e analizzati in modo opportuno, possono diventare una ricchezza per la comunità stessa.

L'obiettivo di questa tesi è quello di realizzare un Pipeline di Big Data Analytics che accompagni i suddetti dati nell'intero processo che va dalla loro importazione, alla loro memorizzazione ed infine al loro processamento e analisi. La pipeline dovrà presentare un'architettura adeguata alla gestione di dati potenzialmente estesi dal punto di vista del volume, della velocità con cui vengono generati e dell'eterogeneità. Dovrà inoltre offrire un Frontend attraverso il quale i Business Manager della Community potranno condurre analisi coadiuvate da tecniche avanzate di Machine Learning e prendere decisioni *data-driven* volte al miglioramento della Community stessa.

Grazie a questa tesi, il progetto Hera SDG risulterà arricchito di uno strumento con il quale poter avvalorare i dati raccolti e generare valore su tre fronti: economico, sociale e ambientale.

La tesi è strutturata in tre capitoli. Il capitolo 1 è dedicato interamente al progetto Hera SDG, fornendo sia una presentazione ad alto livello che una descrizione più dettagliata dell'infrastruttura tecnologica su cui si basa.
Nel capitolo 2 vengono poi trattate da un punto di vista teorico e nozionistico le tecnologie e i Framework Big Data che ho utilizzato per realizzare la Pipeline di Big Data Analytics. Tra queste, Apache Kafka, HDFS, Apache Spark e alcune tecniche di Machine Learning.
Il capitolo 3, infine, è dedicato alla realizzazione della pipeline descrivendo il processo di sviluppo e i requisiti che lo hanno guidato. Vengono inoltre descritti i componenti della pipeline da un punto di vista implementativo, di performance e di ottimizzazioni condotte.

L'intero codice del progetto è mantenuto in una repository GitHub, accessibile al link presente nel riferimento [76] della Bibliografia.

Capitolo 1 – Il progetto Hera SDG

Il capitolo primo introduce il progetto Hera SDG attorno al quale orbita il lavoro di questa tesi. La descrizione del progetto viene proposta seguendo un approccio top-down, ossia partendo da quelli che sono i concetti più generali e di più alto livello, per poi scendere nei dettagli più specifici e tecnici.

Nel primo paragrafo viene presentato il contesto in cui è nato il progetto, elencando i principali promotori, gli obiettivi che si pone di raggiungere e le modalità con cui realizzarli.

Il secondo paragrafo, invece, si concentra sulla piattaforma ICT utilizzata dal progetto, descrivendola da un punto di vista tecnico e architetturale. La descrizione è talvolta arricchita da digressioni inerenti alle tecnologie utilizzate.

1.1 Contesto generale e obiettivi

Hera SDG nasce come progetto condiviso tra le aziende Hera, Conad e Camst, unitesi in un partenariato a cui fa capo Hera. Il nome del progetto rimanda ai Sustainable Development Goals (SDG) elencati nell'Agenda 2030 per lo Sviluppo Sostenibile, un programma sottoscritto nel settembre 2015 dai governi dei 193 Paesi membri dell'ONU. Gli SDG sono una lista di 17 obiettivi accomunati da un'idea di sviluppo e prosperità mondiale che non sia solo di carattere economico ma anche sensibile alle questioni sociali ed ambientali. In questa lista figurano quindi obiettivi come la lotta alla povertà, alle ineguaglianze sociali, l'eliminazione della fame nel mondo, il diritto all'educazione e il contrasto al cambiamento climatico.

L'Agenda 2030 sottolinea come per raggiungere obiettivi così alti senza scadere nell'utopia sia di fondamentale importanza un ampio coinvolgimento non solo dei Paesi, ma in primis degli individui e delle realtà locali. Infatti, solo avviando una solida collaborazione che sia tessuta a partire dai singoli e motivata dalla condivisione di un traguardo comune è possibile fare comunità e raggiungere gli obiettivi prefissati. A questo proposito, l'ultimo goal della lista è intitolato "Partnerships for the Goals" e recita: «The SDGs can only be realized with strong global partnerships and cooperation. A successful development agenda requires inclusive partnerships — at the global, regional, national and local levels — built upon principles and values, and upon a shared vision and shared goals placing people and the planet at the centre».[2]

E ancora, nei sotto-target 17.16: «Enhance the global partnership for sustainable development, complemented by multi-stakeholder partnerships that mobilize and share knowledge, expertise, technology and financial resources, to support the achievement of the sustainable development

goals in all countries, in particular developing countries» [2], e 17.17: «Encourage and promote effective public, public-private and civil society partnerships, building on the experience and resourcing strategies of partnerships».[2]

Hera SDG propone la creazione di un modello comunitario che incentivi comportamenti di sostenibilità e quindi persegue direttamente il Goal 17. Si punta alla creazione di una Smart Sustainable Community, con protagonisti i cittadini e le aziende del territorio, che abbia il duplice obiettivo di aggregare l'offerta di prodotti e servizi sostenibili e di valorizzare e incentivare tutti quei comportamenti virtuosi che l'individuo compie ogni giorno, con le motivazioni più svariate. In questo modo sarà possibile promuoverli a comportamenti di comunità e non resteranno comportamenti isolati e individuali.

Il termine smart indica che la comunità è supportata dalla presenza di un progetto ICT. È anche un richiamo al modello di smart cities cui sempre più città si stanno adeguando, e al quale questo progetto è inerente. Sustainable invece si riferisce al tipo di sviluppo che il progetto pone come obiettivo della comunità. Uno sviluppo che, riportando la definizione citata alla Commissione mondiale sull'ambiente e lo sviluppo dell'ONU (1987), «soddisfa le necessità delle attuali generazioni senza compromettere la capacità delle future generazioni di soddisfare le proprie».

In fase di pianificazione del progetto è stata stilata una lista delle azioni che si desidera promuovere sin dalla fase di sperimentazione. Sono di seguito riportate in Tabella 1.1.

PARTNER	COMPORTAMENTO INCENTIVATO
GRUPPO HERA	Acquisto Energia Elettrica da fonti rinnovabili CO_2 risparmiata da Diario dei consumi Energia Elettrica Acquisto GAS metano CO_2 free CO_2 risparmiata da Diario dei consumi GAS metano Autolettura consumo gas Autolettura consumo acqua Segnalazione fuga acqua stradale Invio elettronico della bolletta Download my Hera Iscrizione ai servizi on line Attivazione alert autolettura
CONAD	Acquisto prodotti sostenibili recupero bottiglie di plastica per filiera bottle to bottle
CAMST	Acquisto piatti/menù sostenibili (carbon e water footprint + aspetti nutrizionali) Acquisto piatti last minute (take away) Raccolta differenziata on site sulle frazioni di rifiuto generate dal pasto

Tabella 1.1 – Lista dei comportamenti incentivati da ogni partner e che verranno testati in fase di sperimentazione del progetto.

I comportamenti sopra elencati spaziano negli ambiti di azione di diversi SDG. Volendo citare qualche esempio: promuovere il consumo e l'acquisto di piatti con un ridotto carbon e water footprint oppure il consumo di piatti/menù sani ed equilibrati da un punto di vista nutrizionale concorre agli obiettivi del Goal 2 («porre fine alla fame nel mondo, raggiungendo la sicurezza alimentare, migliorare la nutrizione e promuovere un'agricoltura sostenibile») e del Goal 3 («assicurare la salute e il benessere per tutti e tutte le età»).

La creazione di una community che responsabilizza e incentiva i cittadini a comportarsi sostenibilmente concorre al Goal 11 («rendere le città e gli insediamenti umani inclusivi, sicuri, duraturi e sostenibili»). Ancora, promuovere la riduzione degli sprechi sia in termini energetici che materici, incentivare il conferimento differenziato dei rifiuti, privilegiare servizi e prodotti che hanno avviato un percorso di minimizzazione delle emissioni di CO2 e incentivare l'utilizzo del digitale a sfavore del cartaceo contribuisce ai Goal 12 («garantire modelli sostenibili di produzione e di consumo»), 13 («combattere i cambiamenti climatici») e 15 («proteggere, ripristinare e favorire un uso sostenibile dell'ecosistema terrestre»).

L'idea di Hera SDG è quella di promuovere ciascuna di queste azioni attraverso un meccanismo di ricompensa. Tramite opportuni dispositivi Internet of Things (IoT), infatti, sarà possibile verificare e registrare ogniqualvolta uno dei comportamenti virtuosi sopra elencati verrà messo in pratica da un cittadino della community. Il comportamento verrà premiato corrispondendo al cittadino che l'ha effettuato un premio in token, una sorta di moneta digitale. I token potranno essere accumulati su un portafoglio digitale e personale, detto wallet, e in seguito essere spesi in uno o più marketplace associati alla community per usufruire di scontistiche monetarie. In questi marketplace i soggetti partner offrono prodotti e/o servizi etici e sostenibili perseguendo un duplice fine. In primis, quello di far confluire la capacità di spesa degli utenti nel raggiungimento sempre più esteso degli obiettivi di sostenibilità SDG. Allo stesso tempo, quello di stimolare forme di consumo all'interno di un ecosistema comunitario.

In Figura 1.1 viene riassunto graficamente il meccanismo di ricompensa appena descritto.

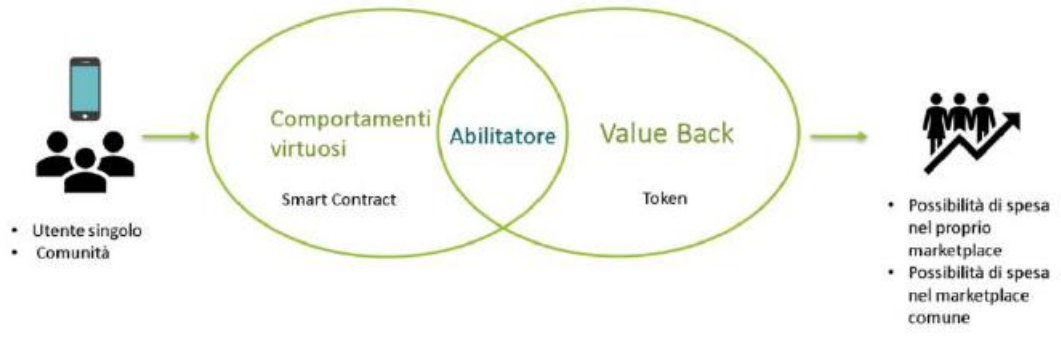


Figura 1.1 – Meccanismo di ricompensa su cui si basa il progetto Hera SDG.

Prima di proseguire ed esporre nei dettagli il progetto, è utile fare una breve digressione che introduca alla conoscenza dei partner coinvolti e a comprendere meglio il contesto in cui si inserisce.

Il Gruppo Hera nasce nel 2002 dall'aggregazione di 11 aziende municipalizzate emiliano-romagnole. Da allora ha intrapreso un percorso di crescita che l'ha portata ad essere oggi una delle maggiori multiutility a livello nazionale. Hera pone al centro della propria missione e dei propri valori la sostenibilità e lo sviluppo sostenibile. Opera principalmente in tre settori: ambiente (gestione dei rifiuti), idrico (acquedotto, fognature e depurazione) ed energia (distribuzione e vendita di energia elettrica, gas e servizi energia). Si aggiungono poi l'illuminazione pubblica e i servizi di telecomunicazione.[3]

Ad oggi, il partenariato di riferimento della comunità è composto da tre imprese, come detto in precedenza Hera, Conad e Camst, ma in prospettiva ci sono i presupposti e le basi per un coinvolgimento sempre maggiore di altre realtà presenti sul territorio. Ciononostante, la potenziale comunità di partenza è già rilevante se si considera che nel panorama italiano il Gruppo Hera soddisfa i bisogni di 4,3 milioni di cittadini distribuiti in 330 comuni dell'Emilia-Romagna, Friuli-Venezia Giulia, Marche, Toscana e Veneto.[3] A questi vanno aggiunti i cittadini coinvolti nei rispettivi business di Conad, consorzio aggregante 7 grandi cooperative territoriali che complessivamente associano 2713 dettaglianti, e Camst, che ogni giorno fornisce pasti a centinaia di migliaia di clienti fra scuole, ospedali, aziende e fiere.

Il progetto Hera SDG ha quindi tutte le carte in regola per apportare un discreto impatto sulla sostenibilità del territorio. Tuttavia, una buona riuscita dello stesso dipenderà in gran parte dalla capacità di coinvolgimento degli utilizzatori e dalla loro risposta. Saranno indispensabili campagne di comunicazione e di engagement che suscitino nel cittadino il desiderio di compiere azioni virtuose e stimolino in lui un senso di appartenenza alla comunità.

Come già sottolineato, Hera SDG è un progetto che si basa su una piattaforma ICT dalla quale non può prescindere. Il prossimo paragrafo si occupa di descriverne la struttura e il funzionamento.

1.2 Descrizione tecnica

La piattaforma ICT del progetto Hera SDG è un sistema distribuito che può essere suddiviso logicamente in tre livelli:

- Blockchain.
- Backend.
- Frontend.

A ciascuno di essi è dedicata in seguito una sezione di questo paragrafo. Il lettore consideri la Figura 1.2 qui sotto riportata come immagine di riferimento per orientarsi nella descrizione di ciascun livello.

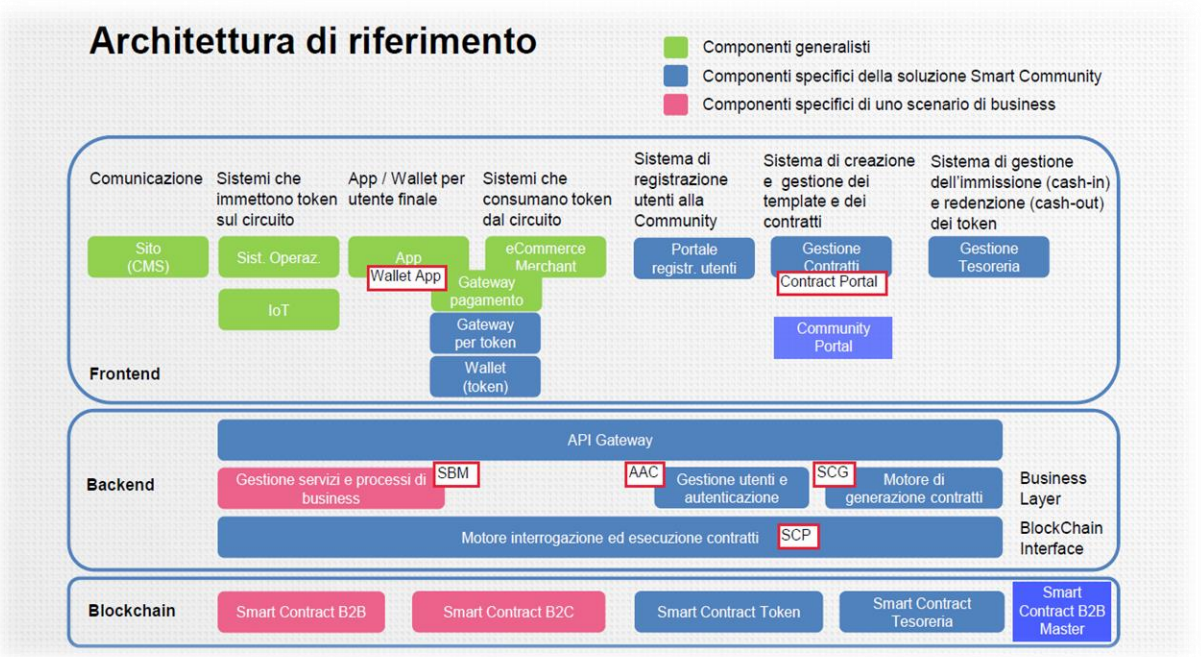


Figura 1.2 – Architettura del progetto Hera SDG.

1.2.1 Blockchain layer

Il livello infrastrutturale su cui si basa la piattaforma è composto da una blockchain sulla quale vengono distribuiti degli smart contract.

La blockchain può essere definita come un registro digitale distribuito e immutabile.[5] Distribuito in quanto ad essa partecipano più nodi tra loro paritari. Questi condividono i dati salvati sul registro, vi possono liberamente accedere in lettura e possono richiedere la scrittura di nuove informazioni. Immutabile invece si riferisce al fatto che, una volta che un dato è stato

aggiunto alla blockchain, non potrà più essere modificato né tantomeno eliminato, a meno di invalidare l'intero registro.

È possibile memorizzare sul registro delle cosiddette transazioni, cioè dei dati strutturati che sono formati da più campi, anche di tipo eterogeneo. Spesso però, accade che la memorizzazione di una transazione sulla blockchain non avvenga appena è richiesta ma solo dopo che più transazioni sono state aggregate insieme a formare un blocco. Memorizzare il blocco significa aggiungerlo agli altri blocchi già presenti e collegarlo a quello più recente. Si viene così a formare una "catena di blocchi", da cui deriva il nome della tecnologia.

Basandosi sui principi tecnologici della blockchain sono state implementate diverse piattaforme, soluzioni e applicazioni. Un esempio è la piattaforma Ethereum, progetto open source lanciato nel 2015. Come altre blockchain, anch'essa mette a disposizione una criptovaluta, chiamata Ether, ma ciò che la differenzia dalle altre è il fatto di essere una blockchain programmabile.[6] Infatti, offre agli sviluppatori la possibilità di distribuire e memorizzare su di essa dei programmi informatici, detti smart contract, e di farli eseguire dalla Ethereum Virtual Machine (EVM).

Gli smart contract sono codici che formalizzano un insieme di regole e accordi riguardanti il rapporto tra più parti, non necessariamente collaborative, proprio come avviene in un contratto. E, come ogni contratto, anche gli smart contract sono composti da un insieme di condizioni e di effetti. Ogniqualvolta una delle condizioni viene soddisfatta, gli effetti ad essa associati vengono automaticamente eseguiti. Il tutto avviene in modo trasparente ai partecipanti dell'accordo e coerente con quanto concordato alla stipulazione del contratto.

Come infrastruttura del progetto è stata scelta una blockchain Ethereum principalmente per due ragioni. In primo luogo, perché essa verrà utilizzata come registro distribuito dove tenere traccia dei comportamenti virtuosi e delle transazioni di valore, sia riguardanti i token trasferiti verso i wallet digitali come incentivo premiale che quelli spesi in uno dei marketplace associati. A questo proposito, la tecnologia blockchain offre:

- Trasparenza: tutti i partner del progetto, Hera, Conad e Camst, hanno la possibilità di consultare e tenere monitorati i dati presenti sulla blockchain. Dal momento che le transazioni sono registrate in forma anonimizzata, avendo l'accortezza di non registrare dati personali, non ci sono problemi di privacy da gestire.
- Immutabilità: una volta che un dato è stato scritto e distribuito sulla blockchain non può essere modificato (e quindi nemmeno corrotto) senza il consenso della maggioranza dei nodi appartenenti alla rete.

- Sicurezza e integrità dei dati: la sicurezza del sistema è garantita dal meccanismo del consenso tra i nodi della rete, dalla marca temporale presente in ogni transazione e dalla cifratura con meccanismo a chiave pubblica-privata.
- Decentralizzazione: il registro distribuito permette ai partner del progetto di associarsi in una comunità paritaria, disintermediando e rendendo superflua la presenza di un ente centrale che certifichi i dati.

Il secondo motivo a supporto della scelta di una blockchain Ethereum è la possibilità di distribuzione ed interazione con gli smart contract, i quali all'interno del progetto ricoprono un ruolo determinante. Vengono infatti usati per la gestione di tutti gli aspetti concernenti le regole di premialità e di incentivo dei comportamenti virtuosi. Grazie alle proprietà della blockchain elencate in precedenza, questi contratti trovano nella blockchain la possibilità di essere autenticati e certificati nella loro validità e legittimità senza la presenza di alcun intermediario (e.g. un notaio). In questo modo vengono abbattuti i costi di un'eventuale gestione notarile e semplificato il processo burocratico.

Nel progetto HSC sono previste quattro categorie di smart contract (SC):

SC Tesoreria

Questo smart contract definisce la lista di business partner[1], la lista dei business store[2], e le regole finanziarie dell'iniziativa.

Tra queste vi sono i processi relativi all'immissione e alla redenzione di token dalla community. La prima operazione è detta cash-in e avviene quando un business partner effettua un bonifico bancario sul conto di tesoreria dell'abilitatore della community[3]. Questi mantiene monitorati i movimenti del conto (tramite Internet Banking) e, non appena nota un nuovo versamento, provvede ad effettuare un trasferimento di token a beneficio del business partner mittente del bonifico. Per farlo sfrutta la funzione executeCashIn dello SC. Il quantitativo di token che vengono trasferiti dipende ovviamente dal tasso di cambio token/euro, che è fissato a 1.0, e che è una delle regole finanziarie specificabili in questo contratto.

---

[1] Nella community sono presenti quattro tipi di "attori" diversi: l'*abilitatore della community* (Hera), i *business partner*, ossia le aziende che fanno parte del partenariato del progetto (Hera, Conad e Camst), i *business store*, ossia i punti vendita (online o offline) affiliati ai business partner ed infine gli *end user*, ossia gli individui che sono clienti dei business store, che accumulano e spendono i token.

[2] *ibid.*

[3] *ibid.*

La redenzione di token dalla community invece è detta cash-out e consiste nel processo opposto. Esso avviene in due step: inizialmente un business store fa la richiesta di cash out, tramite la funzione requestCashOut dello SC, trasferendo un quantitativo di token verso lo SC tesoreria. Qui i token vengono bloccati in uno stato intermedio di custodia e non sono utilizzabili in alcun modo. In seguito, l'abilitatore della community, che mantiene monitorate le richieste di cash out, effettua il bonifico monetario al business store e conferma l'operazione di cash out con la funzione confirmCashOut dello SC. Questa funzione rimuove dal bilancio dello SC tesoreria i token che erano in custodia e li trasferisce allo SC token.
Sia le operazioni di cash-in che quelle di cash-out vengono registrate nella blockchain.

SC Token

Lo SC Tesoreria è stato realizzato per gestire le funzionalità come immissione e redenzione dei token dal sistema ma non prevede le funzionalità di gestione dei token, come ad esempio il monitoraggio del bilancio totale di token generati, il bilancio di token posseduto da un certo account, il trasferimento degli stessi tra diversi account e/o smart contract del sistema. A questo proposito viene utilizzato lo SC Token, che è un SC di tipo ERC20[4].

SC B2B master

Questo smart contract definisce l'insieme delle regole di business, ossia la lista di tutti i possibili comportamenti premianti e i relativi premi associati, per ogni partner. Serve come guida per la stipula dei contratti B2B e B2C.

SC B2B

Questo smart contract viene stipulato tra un business partner ed uno dei suoi store affiliati. In esso, a partire dal SC B2B master, viene declinato un sottoinsieme di specifici prodotti o comportamenti che si desidera premiare in quel particolare business store.
Lo SC B2B è necessario per permettere flessibilità e maggiore libertà di azione al singolo store che in questo modo può decidere, per le più svariate ragioni, di premiare l'acquisto di determinati prodotti ma non quello di altri, anche se il B2B master li aveva elencati tra i premianti (e.g. diversi supermercati Conad possono premiare l'acquisto di prodotti differenti).

[4] ERC20 (Ethereum Request Comment20) è uno standard per la realizzazione di smart contract che gestiscono dei token di tipo fungibile. Beni fungibili sono beni che tra loro sono tutti equivalenti ed interscambiabili (e.g. il denaro). Seguendo questo standard è possibile originare un proprio token e creare uno SC che si occupi della sua gestione. È possibile anche personalizzare il token scegliendo un nome e alcune proprietà come, ad esempio, il numero massimo di decimali consentiti.

SC B2C

Questo smart contract viene stipulato tra un business partner e i suoi end user[5]. Permette di specificare l'insieme completo dei comportamenti premianti riconosciuti da quel partner e i relativi premi offerti. Inoltre, permette di specificare le policy di privacy ed eventuali informazioni aggiuntive che sono richieste per la corretta registrazione dei comportamenti dell'end user (e.g. codice cliente Hera, numero di carta Conad, ecc.).

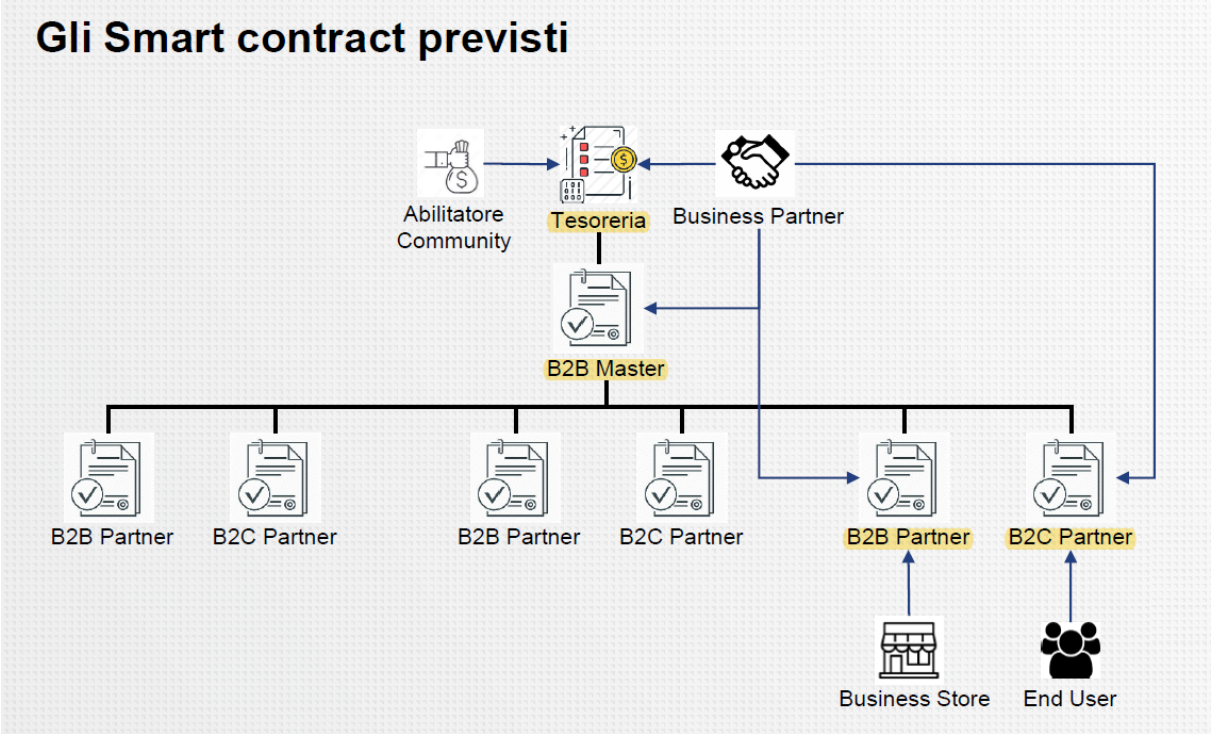


Figura 1.3 – Tassonomia degli Smart Contract previsti nel progetto Hera SDG.

Una volta comprese le tipologie di smart contract previste nel progetto, riassunte in Figura 1.3, è possibile approfondire l'argomento descrivendo le modalità e la sequenza temporale con cui essi vengono creati e stipulati.

1. Creazione e sottoscrizione degli SC Tesoreria e B2B master: inizializzazione della community.

In fase di inizializzazione della community spetta ad Hera, in qualità di abilitatore, la creazione dello SC Tesoreria, dello SC B2B master e la loro registrazione sulla blockchain. Questo significa che appartiene ad Hera il compito di definire le regole finanziare, di business e di

[5] *Ivi*, pag.9.

compilare la lista dei business partner con i relativi store. Il tutto, però, avverrà secondo quanto negoziato e sottoscritto offline tra Hera e gli altri partner.
A questo punto, tramite il loro account su Contract Portal, che è uno dei componenti di frontend, i business partner potranno visualizzare sia l'HTML che il codice Solidity[6] dei due contratti registrati e sottoscriverli.

2. Creazione e sottoscrizione degli SC B2B.
Solo una volta che quanto descritto al punto 1 è avvenuto, i business partner hanno la possibilità di declinare contratti B2B con i loro store mediante una procedura composta da due passaggi. Per prima cosa Hera, sotto proposta e richiesta di un business partner, crea un template di contratto a partire dal B2B master e lo pubblica. La creazione del template avviene sfruttando il Contract Portal mentre la pubblicazione avviene sul Community Portal, un altro componente di frontend. Previa approvazione del partner, i singoli store hanno la possibilità di visualizzare il template e compilarlo con i propri dati e referenze, creando in questo modo lo SC B2B che provvederanno a registrare su blockchain.

3. Creazione e sottoscrizione degli SC B2C.
Analogamente a quanto descritto nel punto 2 e sempre a partire dallo SC B2B master, i template degli SC B2C verranno creati e in seguito pubblicati su Community Portal. L'end user potrà quindi visualizzarli tramite il Community Portal e sottoscriverli grazie al suo wallet.

Ai fini di una maggiore flessibilità si è pensato di organizzare le attività della community per "iniziative", prevedendo di creare uno Smart Contract Tesoreria e uno Smart Contract B2B master per ogni iniziativa attiva. Infatti, iniziative diverse potrebbero avere una lista di partner o caratteristiche finanziarie diverse, come ad esempio il tasso di cambio token/euro. Il progetto HSC è in grado di supportare la presenza di più iniziative attive nello stesso momento. Tuttavia, ogni iniziativa deve essere indipendente e non interagente con le altre. A questo proposito anche i token di ogni iniziativa devono essere diversi e quindi non inter-spendibili.
In conclusione, il connubio blockchain-smart contract permette di realizzare all'interno del progetto un registro distribuito nel quale poter memorizzare tutti i dati rilevanti alla community e con il quale poter definire e regolamentare le relazioni tra i diversi individui della community. Il sistema viene realizzato prescindendo dalla necessità di un'entità super partes che validi e

[6] Solidity è un linguaggio di programmazione per la scrittura di Smart Contract.

certifichi i dati, ma ciononostante sia il cittadino sia i partner economici coinvolti possono fidarvisi in termini di sicurezza e veridicità.

1.2.2 Backend layer

Il livello di backend del progetto è composto da un insieme di cinque microservizi web che espongono delle API secondo lo stile architetturale REST[7]. Esse sono quindi invocabili con delle chiamate tramite protocollo HTTP e questo permette ai servizi di essere disponibili a gran parte delle piattaforme oggi presenti, vista la larga diffusione del protocollo.

L'architettura a microservizi

La scelta di realizzare il backend con un'architettura orientata ai microservizi è stata dettata dalla consapevolezza riguardo ai vantaggi che essa garantisce rispetto all'opposto modello architetturale monolitico. Questa presa di coscienza si è iniziata a diffondere tra la comunità scientifica a partire dall'avvento del modello architetturale Service Oriented Architecture (SOA), che poi si è evoluto nella variante a microservizi.

Un microservizio è un blocco di codice di dimensioni limitate che realizza una specifica funzionalità di business dell'applicazione, viene eseguito in maniera indipendente dagli altri servizi ed espone determinate funzionalità che sono invocabili secondo un'interfaccia ben definita. Volendone elencare le caratteristiche, un microservizio è:

- **Indipendente/autonomo**: viene eseguito da un processo dedicato ed indipendente dagli altri microservizi della stessa applicazione. Si dice anche che i microservizi sono tra loro debolmente accoppiati (loose coupled) in quanto le dipendenze l'uno dall'altro sono ridotte al minimo: le interazioni avvengono secondo quanto specificato dall'interfaccia esposta, che per natura è (o almeno dovrebbe essere) immutabile e non vi è condivisione di dati in quanto ogni microservizio possiede un proprio database.
- **Specializzato**: ogni microservizio si limita a realizzare una specifica funzionalità di business dell'applicazione.
- **Riusabile**: dal momento che un microservizio implementa una limitata funzionalità della nostra applicazione, è molto probabile che questa funzionalità possa tornare utile anche allo sviluppo di altre applicazioni che condividono la stessa esigenza. In questo caso sarà

[7] Pseudonimo di Representational State Transfer, REST è uno stile architetturale per i sistemi distribuiti che prevede la trasmissione di dati sul protocollo HTTP sfruttando i suoi metodi specifici per il recupero di informazioni (GET), per la modifica (POST, PUT, DELETE, PATCH) e per altri scopi (OPTIONS, ecc.).
(Wikipedia, https://it.wikipedia.org/wiki/Representational_State_Transfer - consultato il 16 dicembre 2020)

possibile utilizzare il microservizio già esistente, accelerando così il time-to-market della nuova applicazione.

- **Componibile**: diversi microservizi possono essere aggregati insieme per implementare una funzionalità più complessa.

Grazie ai microservizi è possibile riformulare il design architetturale di un'applicazione software suddividendola nelle sue funzionalità di base e implementando per ciascuna di esse un servizio che la realizzi. L'applicazione risulterà quindi composta da tanti processi indipendenti che però possono interagire tramite richieste che viaggiano sulla rete e che rispettano il protocollo e la signature specificati nell'interfaccia esposta. Questo design architetturale si contrappone all'architettura monolitica dove invece tutti le funzionalità sono strettamente collegate, interdipendenti tra loro e vengono eseguiti come un singolo servizio.

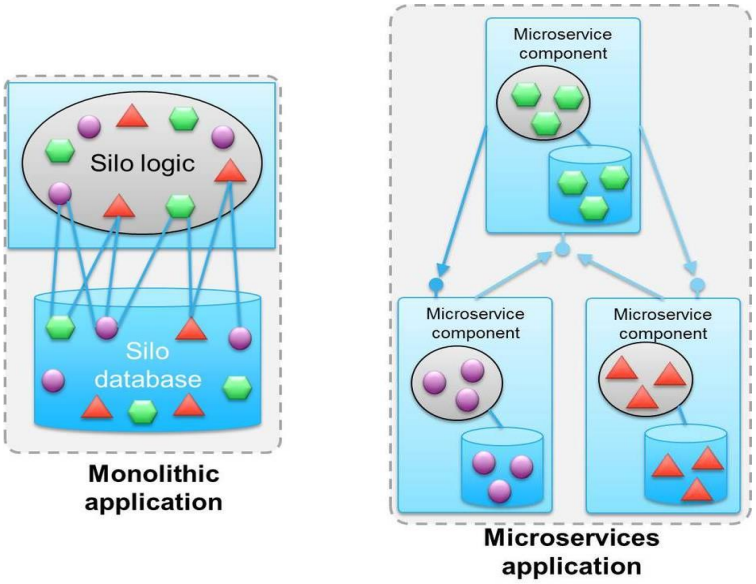


Figura 1.4 – Confronto tra un'applicazione con architettura monolitica e un'applicazione con architettura a micro-servizi.

L'approccio ad architettura monolitica è stato abbandonato nel corso del tempo perché presenta diversi punti deboli che vengono invece risolti dall'approccio a microservizi.

Un'applicazione orientata ai microservizi presenta infatti i seguenti vantaggi: [7][8]

- **Manutenibilità ed estensibilità:** dal momento che l'applicazione è suddivisa in molti microservizi, se si modifica il codice di uno di questi sarà sufficiente fare il deploy solo di quel servizio. Viceversa, una modifica anche piccola al codice di un'applicazione monolitica richiede un nuovo deploy dell'intera applicazione. Questo è un aspetto molto vantaggioso considerando che le modifiche al codice possono essere assai frequenti (correzioni di bug, cambio dei requisiti che porta a modifiche o migliorie del codice, implementazioni di nuove funzionalità richieste…).

- **Resilienza:** l'indipendenza dei microservizi aumenta la resilienza di un'applicazione in caso di errori. In un'architettura monolitica, un errore che riguarda un unico componente potrebbe avere ripercussioni sull'intera applicazione. Con i microservizi invece, le applicazioni possono gestire gli errori di un servizio isolando la specifica funzionalità senza bloccare l'intera applicazione.[7]
- **Libertà tecnologica:** l'indipendenza dei microservizi permette anche di astrarre dalla loro implementazione interna. Ciascun microservizio può quindi essere sviluppato in maniera diversa dagli altri, dando agli sviluppatori la possibilità di scegliere gli strumenti più adeguati e ottimali (linguaggio di programmazione, tecnologie, tipo di storage, ecc.) per ogni specifico problema.
- **Scalabilità:** se un'applicazione con architettura a microservizi sperimenta la necessità di dover scalare orizzontalmente, sarà possibile agire sul singolo servizio che è più stressato in termini di traffico e limitarsi a replicare quello. Viceversa, in un'architettura monolitica si è costretti a replicare l'intera applicazione, comprese le funzionalità che magari non ne avevano la necessità, e questo comporta uno spreco di risorse.

I microservizi del backend

Terminata questa digressione sui microservizi, mirata a motivare la scelta architetturale, possiamo passare ad elencare i microservizi presenti nel progetto Hera SDG.

Smart Contract Generator (SCG)

Questo microservizio è stato sviluppato utilizzando il linguaggio di programmazione Java e il supporto del framework Spring Boot. Espone le proprie API secondo lo stile architetturale REST e dispone di un proprio database Postgresql.

SCG permette di gestire il processo di creazione di un contratto in tre passaggi:

1. Creazione di uno smart contract template: Hera, in qualità di abilitatore della community, crea un template dello Smart Contract, il quale sarà composto dal codice Solidity del contratto, dal codice HTML con il testo human readable del contratto e da un JSON Schema[8]. La denominazione "template" è dovuta al fatto che lo sviluppatore incaricato di scrivere il codice Solidity e HTML esprimerà alcune variabili del Solidity e i relativi input

[8] JSON (JavaScript Object Notation) è un particolare formato di scambio di dati molto utilizzato in rete per via della sua facile integrazione con linguaggi di programmazione orientati ad oggetti e per la minor banda occupata rispetto ad altri formati come XML. Il JSON Schema descrive, in modo dichiarativo e sempre utilizzando il formato JSON, la struttura di altri dati. Può essere quindi usato per validarli.

tags dell'HTML, sotto forma di Mustache[9] tags. Questi tag non hanno un valore associato fintanto che non viene inserito dall'utente finale al momento della compilazione del template (vedi passaggio 3).

2. Testing: il secondo passaggio prevede un testing automatico del template, per verificare la correttezza del codice Solidity sia da un punto di vista sintattico che da un punto di vista della sicurezza e della presenza di eventuali vulnerabilità. Il testing viene effettuato sfruttando i tool Solium[10] e MythX[11].
3. Creazione dello Smart Contract: un utente finale (e.g. un business store nel caso di contratti B2B o end user nel caso di B2C) provvederà a compilare il template fornendo tramite gli input tags HTML i valori delle variabili che erano state lasciate sospese al punto 1 (e.g. le referenze di quel preciso store). I valori verranno inviati allo SCG in formato JSON e questi, dopo averne verificato la validità secondo il JSON Schema, provvederà ad inserirli nel codice Solidity dello smart contract. La connessione tra le variabili HTML e Solidity è realizzata con il tool Mustache. A questo punto il contratto viene salvato sul database dello SCG, in modo da rimanere disponibile per future modifiche o per il successivo deploy su blockchain. Il procedimento appena descritto è rappresentato in Figura 1.5.

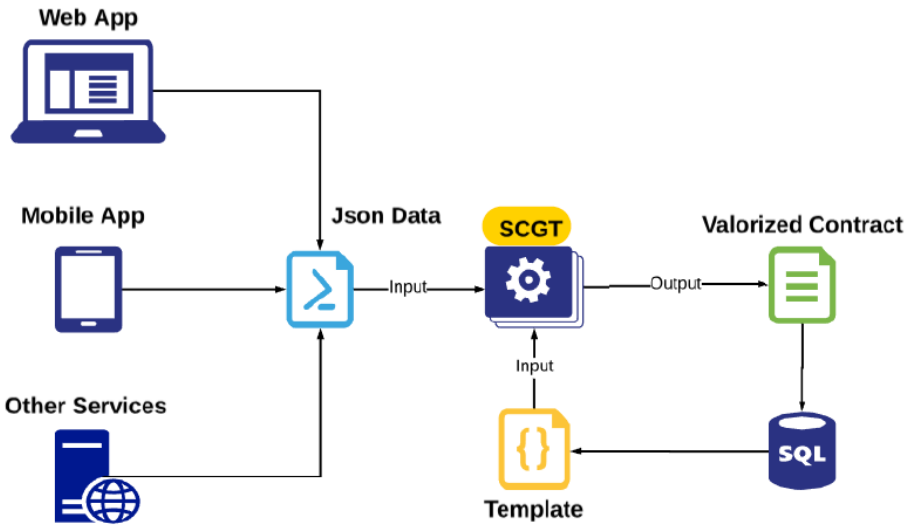


Figura 1.5 – Flusso di creazione di uno smart contract sfruttando lo Smart Contract Generator.

[9] Mustache è un web template system, ossia un tool che permette agli sviluppatori di scrivere codice sorgente o file di configurazione sotto forma di template, lasciando cioè nel codice dei *mustache-tags* privi di valore. Mustache permette di popolarli dinamicamente usando dei valori forniti in seguito.

10 Solium (che recentemente ha cambiato nome in Ethlint) è un linter per il codice Solidity, ossia uno strumento che analizza il codice evidenziando problemi di stile e sicurezza ed aiuta a risolverli.

[11] MythX è un tool in grado di scansionare il codice di uno Smart Contract e di rilevare falle di sicurezza o potenziali vulnerabilità.

Smart Contract Proxy (SCP)

Come SCG, anche questo microservizio è stato sviluppato utilizzando il linguaggio di programmazione Java e il supporto del framework Spring Boot. Anch'esso espone le proprie API secondo lo stile architetturale REST e dispone di un proprio database Postgresql, distinto da quello di SCG per rispettare i principi dell'architettura a microservizi.

Le API esposte da SCP possono essere suddivise nei seguenti sottoinsiemi:

- *Gestione smart contract e deploy su blockchain:* tramite un'API dedicata è possibile inviare a SCP un contratto in formato JSON per richiedere che venga distribuito su blockchain. Il contratto da inviare può essere recuperato tra quelli precedentemente creati con SCG e salvati sul suo database, oppure creato ex-novo. La prima scelta, tuttavia, è consigliata perché garantisce che il contratto sia stato testato.

  Una volta ricevuto il contratto, SCP provvederà ad estrarre il codice Solidity, a trasformarlo in bytecode e a distribuirlo sulla blockchain con l'ausilio di Truffle, un framework di sviluppo che descriverò in sezione 2.2.

  Oltre a distribuire il contratto su blockchain, SCP provvede a salvarne una copia sul proprio database. Con altre API sarà poi possibile recuperare, modificare ed eliminare gli smart contract presenti sul database. Onde evitare equivoci è bene specificare che la procedura di eliminazione rimuove soltanto lo smart contract dal database mentre non è possibile eliminare dalla blockchain uno smart contract già distribuito, a causa della proprietà di immutabilità di cui gode la stessa.

  La procedura di deploy di uno smart contract su Blockchain è riassunta in Figura 1.6.

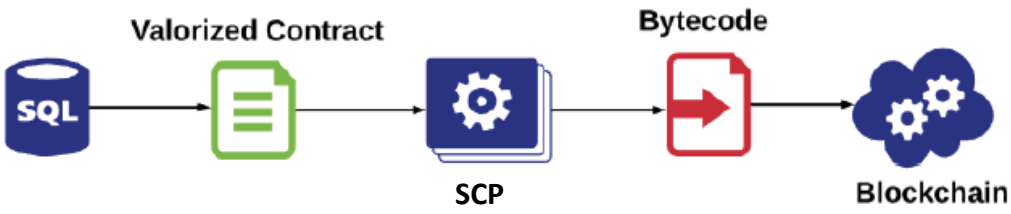


Figura 1.6 – Flusso di deploy di uno smart contract su blockchain sfruttando lo Smart Contract Proxy.

- *Interazione con smart contract già distribuiti:* una volta fatto il deploy di un contratto sulla blockchain sarà possibile invocarne le funzioni. Questo è permesso grazie a un'API a cui fornire in input un oggetto JSON contenente il nome della funzione da invocare, un discriminante che indichi se la suddetta funzione cambia o meno lo stato della blockchain,

i parametri di ingresso e di uscita. SCP provvederà ad interagire con il contratto sfruttando Web3.js, una collezione di librerie ideate per interagire con nodi Ethereum.

SCP, quindi, dà la possibilità ad un client qualsiasi di interagire con la blockchain in maniera trasparente e senza che questi vi si connetta direttamente.

- *Gestione della rete blockchain:* SCP mette infine a disposizione delle API per la gestione delle reti blockchain, quali l'aggiunta di una nuova rete, la modifica, il recupero o l'eliminazione di una rete esistente. Queste API, oltre a modificare il database, agiscono sul file di configurazione truffle-config per modificare il comportamento di Truffle.

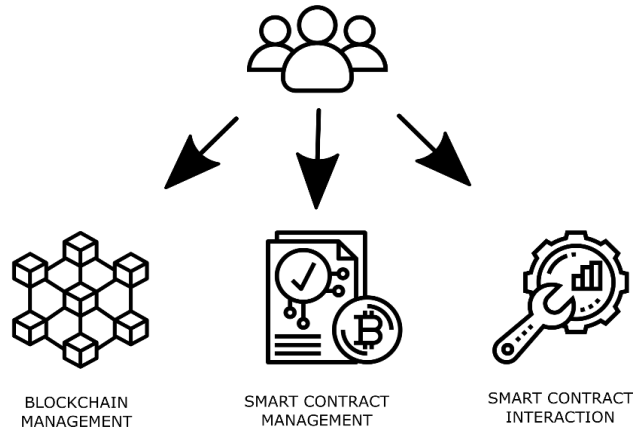


Figura 1.7 – Riassunto delle funzionalità offerte dal microservizio Smart Contract Proxy.

Authentication and Authorization Controller (AAC)

Questo microservizio permette di gestire la registrazione degli utenti e i loro profili, oltre a controllare l'autenticazione e l'autorizzazione per accedere alle API esposte dai microservizi tramite API Gateway. AAC agisce da Identity Provider e da Authorization Server seguendo gli standard OpenID Connect (OIDC) e OAUTH2. OIDC semplifica e velocizza il processo di registrazione e autenticazione degli utenti poiché permette di recuperare i dati necessari alla registrazione da altri Identity Provider presso il quale l'utente è già registrato (e.g. account social quali Google+ e Facebook) e, in seguito, di delegare ad essi anche l'autenticazione.

Quando l'utente viene autenticato, AAC gli rilascia un token di autenticazione secondo lo standard OAUTH2 e questo potrà essere usato per accedere alle API esposte dai microservizi.

Smart Business Manager (SBM)

Lo SBM è un microservizio che ha il compito di permettere l'interfacciamento con l'applicazione wallet "DCASH App" (componente di frontend sviluppato da terzi e presentato in seguito nella sezione 1.2.3). In particolare, fornisce l'integrazione con il wallet per la registrazione dell'utente e il recupero del suo saldo token.

L'API Gateway

Nel Progetto Hera SDG le API messe a disposizione dai microservizi sono esposte mediante un WSO2 API Manager, un API Gateway di tipo open-source che è prodotto dall'omonima azienda[12]. L'API Gateway è un componente che ha l'obiettivo di centralizzare, limitare e controllare l'accesso ai servizi di backend da parte dei client. I clienti dell'applicazione, quindi, non comunicheranno mai direttamente con i microservizi, bensì interagiranno con il gateway, che svolgerà il ruolo di intermediatore. Questi inoltrerà le richieste al microservizio opportuno, raccoglierà la risposta e provvederà a consegnarla al client.

Il gateway, dal punto di vista dei microservizi, non è altro che un cliente qualsiasi. Tuttavia, la sua presenza può portare benefici all'applicazione.

Innanzitutto, da un punto di vista della sicurezza, poiché evita che i microservizi vengano esposti come endpoint pubblici. Solo il gateway viene esposto direttamente ai client e, come recita un noto principio di sicurezza informatica, minore è la superficie d'attacco, maggiore è la sicurezza della tua applicazione.[10]

Inoltre, l'utilizzo di un gateway permette di disaccoppiare il client dallo strato dei microservizi. Senza di esso, i client dovrebbero avere la consapevolezza di come è strutturato il backend dell'applicazione, di quanti e quali microservizi è composto. Un refactoring del backend vorrebbe dire quindi modificare e riadeguare anche le varie applicazioni client, rendendo di conseguenza complessa e disagevole la maintenance del progetto.

Infine, il gateway può essere sfruttato per realizzare funzionalità relative a problematiche trasversali ai microservizi (cross-cutting concerns), quali autenticazione e autorizzazione, load balancing, logging, response caching, ecc. In questo modo le soluzioni vengono implementate un'unica volta e sono disponibili a tutti i microservizi. I team di sviluppo dei singoli microservizi possono così concentrarsi ad implementare le funzionalità di business.

Vi sono anche degli aspetti a cui prestare attenzione quando nella propria architettura si decide di fare uso di un gateway. Bisogna tenere conto del fatto che, centralizzando, stiamo introducendo nel sistema un potenziale single point of failure, poiché in caso di suo crash i microservizi non sono più raggiungibili, e anche un potenziale collo di bottiglia (bottleneck), dal momento che tutte le richieste convergono su di esso. Questi problemi possono essere risolti effettuando un opportuno scaling, replicando quindi il gateway.

---

[12] WSO2 è un fornitore di tecnologia open source fondato nel 2005. Offre una piattaforma di soluzioni per l'integrazione di API, applicazioni e servizi Web in locale e su Internet.
(Wikipedia, https://en.wikipedia.org/wiki/WSO2- consultato il 17 dicembre 2020)

Deploy dei microservizi e breve panoramica su Docker e Kubernetes

Per quanto riguarda il deploy dei microservizi del backend viene sfruttato Docker, la principale piattaforma per la containerizzazione di applicazioni. Sviluppato dall'azienda Docker Inc. come progetto open source, Docker agevola alquanto il deploy di un'applicazione orientata ai microservizi. Esso, infatti, lavora con delle immagini, ossia dei moduli software leggeri, indipendenti ed eseguibili che includono tutto il necessario per la propria esecuzione (codice sorgente, configurazioni, librerie, dipendenze, un proprio filesystem…).[11] Quando un'immagine viene eseguita dà vita ad un container, cioè un processo sulla macchina che è isolato e indipendente dagli altri. L'isolamento è reso possibile dal fatto che la tecnologia Docker fa uso del kernel di Linux e delle sue funzionalità.[13]

Con riferimento a quanto scritto in precedenza sui microservizi, sono evidenti le analogie tra quest'ultimi e i container. L'associazione, quindi, nasce spontanea: possiamo predisporre un container per ciascun microservizio.

La containerizzazione è simile alla tecnologia di virtualizzazione, tuttavia, rispetto alle macchine virtuali, i container sono più leggeri in quanto condividono tutti lo stesso kernel del sistema operativo. Questo garantisce tempi di avvio molto più rapidi e un minor utilizzo di memoria rispetto a una macchina virtuale che invece possiede un proprio sistema operativo.

Inoltre, il fatto che il container disponga internamente di tutte le proprie dipendenze e che non venga eseguito direttamente sul sistema operativo della macchina host bensì da uno strato soprastante (il Docker Engine) permette di standardizzare la sua esecuzione «ottenendo lo stesso comportamento ovunque lo esegui», risolvendo la cosiddetta *dependency hell*[13] ed eliminando il problema del «sul mio laptop funziona».[12]

In conclusione, Docker garantisce un deploy rapido ed efficiente di microservizi.

Tuttavia, presenta dei limiti: seppur sia in grado di gestire i singoli container in modo efficace, l'utilizzo di un numero sempre maggiore di container e applicazioni containerizzate può portare ad una gestione poco pratica utilizzando il solo Docker. È per questo motivo che sono nati dei tool di orchestrazione di container.

Kubernetes (abbreviato K8s) è sicuramente uno dei più diffusi. Sviluppato da Google nel 2014 come software open source, permette di gestire in modo automatico grandi quantità di container

[13] Ad esempio: supponiamo che due componenti A e B del mio sistema facciano affidamento su una libreria condivisa. Supponiamo anche che questa libreria venga aggiornata dalla versione X alla versione Y, ma A richiede la versione X per funzionare correttamente, B la versione Y. A non potrà più funzionare correttamente.

e di distribuirli eventualmente su cluster (locali o cloud) di più nodi computazionali (fisici o virtuali). Kubernetes offre all'applicazione da esso orchestrata numerosi benefici tra cui: [14]

- Resilienza e alta disponibilità: K8s monitora l'esecuzione dei container, riavviando o sostituendo i container che si bloccano, terminando quelli che non rispondo agli health check.
- Scalabilità: K8s è in grado di fare scale-up o scale-down delle repliche di un container, a seconda delle esigenze. Questo garantisce un uso delle risorse e performance sempre ottimali.
- Load Balancing: se il traffico verso un container è alto, Kubernetes è in grado di distribuirlo su più container in modo che il servizio rimanga stabile.
- Rollout e Rollback automatizzati: K8s permette di aggiornare la nostra applicazione senza down time sostituendo gradualmente i container con le nuove versioni degli stessi. Analogamente è possibile ripristinare una versione precedente se per qualche motivo non si desidera mantenere l'ultimo aggiornamento.

Per ulteriori dettagli e approfondimenti sul funzionamento di Kubernetes si faccia riferimento al paragrafo 3.2.1.

1.2.3 Frontend layer

Il frontend è composto da diversi componenti. Alcuni sono componenti realizzati specificamente per il progetto. Altri invece sono componenti generalisti, che erano già esistenti e che vengono integrati con la piattaforma. I componenti specifici sono i seguenti:

Portale registrazione utenti

Questo componente viene utilizzato dai nuovi utenti per crearsi un profilo e partecipare alla community. Tuttavia, per le fasi iniziali del progetto non è previsto lo sviluppo di questo componente. La sua assenza viene sopperita da un processo di inizializzazione in cui ciascun business partner sceglie un elenco di nominativi (nome, cognome, mail). Questi verranno preregistrati presso l'Identity Provider di ciascun partner, dopodiché ciascuno di essi riceverà una e-mail che conferma la registrazione. Tramite un link presente nella mail l'utente potrà scaricare dallo store (Google o Apple) la Wallet App e attivare il proprio wallet con le credenziali ricevute. A questo punto l'utente è abilitato e può partecipare alla community.

Portale gestione tesoreria

Si tratta di un'interfaccia per la gestione e il monitoraggio delle operazioni di tesoreria.

Contract Portal

Questo componente è un'applicazione web realizzata con il linguaggio Javascript che permette di interagire con le API REST esposte dai microservizi SCG e SCP, previa autenticazione presso AAC. Il Contract Portal viene utilizzato sia dall'abilitatore della community che dai business partner.

Il primo lo utilizza per creare e registrare su blockchain lo SC Tesoreria, lo SC B2B Master e per consultare la reportistica dello SC Tesoreria.

I secondi invece lo utilizzano per visualizzare e sottoscrivere i contratti SC Tesoreria e SC B2B Master, in modo da completare la fase di inizializzazione della community. A partire dal SC B2B master possono inoltre creare i template dei contratti B2B e B2C e pubblicarli sul Community Portal, dove potranno essere compilati e sottoscritti dai business store e dagli end user.

Community Portal/App

Si tratta di una web app accessibile, tramite desktop o mobile, e fruibile sia liberamente che effettuando il login.

L'area pubblica viene usata per divulgare informazioni riguardanti la community. È realizzata con l'ausilio di un Content Management System (CMS) ed è gestita da un'utenza amministrativa di Hera, in qualità di soggetto abilitatore della community.

L'area riservata invece è accessibile, previa login, da parte degli end user e dei business store registrati alla community. Quest'area viene utilizzata per tre scopi:

- Visualizzazione e compilazione dei contratti disponibili: business store ed end user potranno visualizzare i template dei contratti (rispettivamente B2B e B2C) che sono stati pubblicati dal proprio partner di riferimento, sceglierne uno e compilarlo. La compilazione prevede un interfacciamento con il microservizio SCG il quale creerà lo smart contract. A questo punto è necessario sottoscriverlo per renderlo operativo. La sottoscrizione del contratto, però, non viene eseguita all'interno del Community Portal ma avviene attraverso la Wallet App.
- Visualizzazione del resoconto dei contratti sottoscritti.
- Vetrina prodotti: gli end user possono visualizzare i prodotti dei vari business partner che sono acquistabili con i propri token. Facendo click su un determinato prodotto, l'utente verrà reindirizzato al sito e-commerce del relativo partner. Qui potrà essere completato l'acquisto, previa conferma della spesa dei token attraverso la Wallet App.

Tra i componenti generalisti invece troviamo:

Siti e-commerce/app dei partner presso i quali spendere i token

Per quanto riguarda Hera, l'utente può utilizzare i token tramutandoli in sconti da usare per gli acquisti effettuati sul sito ufficiale. Nei siti e-commerce di Conad e Camst, invece, i token possono essere usati per acquistare dei buoni sconto da usare presso gli store fisici.
In entrambi i casi, al momento del checkout del carrello, l'utente viene reindirizzato all'App Wallet dove dovrà confermare la spesa dei token. Una volta confermata, ritorna all'indirizzo del carrello per finalizzare l'acquisto.

Wallet App ("DCash App")

Questo componente viene utilizzato dai business store e dagli end user come wallet. Permette diverse funzionalità ma le principali sono sicuramente:

- visualizzare il saldo dei propri token.
- spendere i token.
- sottoscrivere contratti.

Verrà utilizzata a questo scopo l'applicazione "DCash App", un'applicazione già esistente che è stata realizzata dal gruppo SIA[14].

14 SIA - società controllata da CDP Equity - è leader europeo nella progettazione, realizzazione e gestione di infrastrutture e servizi tecnologici dedicati alle Istituzioni Finanziarie, Banche Centrali, Imprese e Pubbliche Amministrazioni, nei segmenti Card & Merchant Solutions, Digital Payment Solutions e Capital Market & Network Solutions.

Abbiamo visto come il Community Portal e anche i siti e-commerce dei partner abbiano la necessità di interfacciarsi con la Wallet App per usufruire delle funzionalità di sottoscrizione di un contratto oppure di spesa dei token. Queste interazioni sono rappresentate in Figura 1.8.

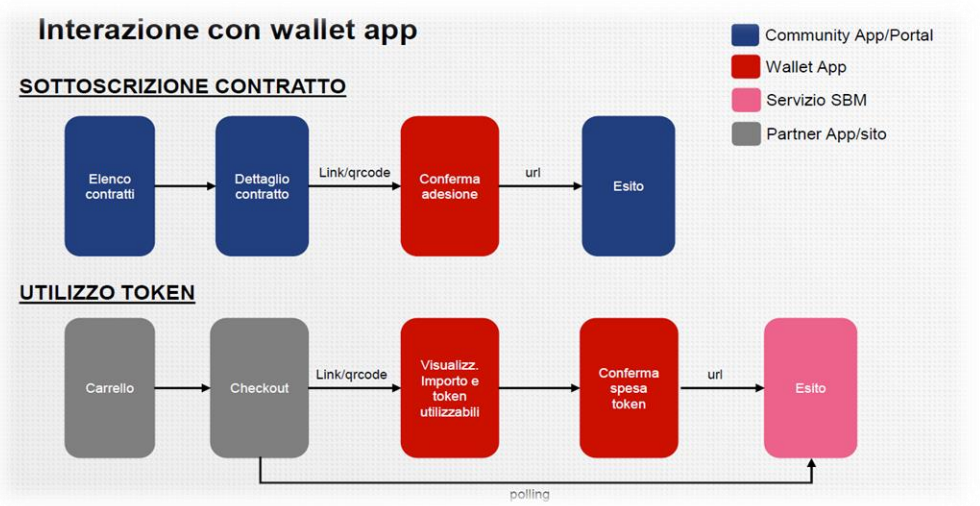


Figura 1.8 – Flussi di interazione con la Wallet App.

Sistemi operazionali e Internet of Things (IoT)

Sono i componenti che forniscono alla piattaforma i dati, relativi ai comportamenti premianti, che verranno usati dagli SC B2C per l'erogazione dei token (e.g. registratore di cassa che fornisce i dati relativi all'acquisto di un prodotto premiante presso un determinato store Conad).

Capitolo 2 – Tecnologie e Framework Big Data

Questo capitolo descrive da un punto di vista teorico le tecnologie e i framework di cui mi sono servito per implementare la pipeline di Big Data Analytics. Si tratta di tecnologie diffuse ed estremamente utilizzate per la realizzazione di pipeline di data processing. Il capitolo è diviso in quattro paragrafi. La trattazione del primo paragrafo riguarda il livello di storage dei dati con la descrizione di Hadoop Distributed File System. Il secondo paragrafo si occupa dell'importazione dei dati descrivendo Apache Kafka e i Kafka Connector. Il terzo paragrafo tratta del framework computazionale Apache Spark. Il quarto paragrafo, infine, ha lo scopo di fornire alcuni cenni teorici sull'argomento del Machine Learning Clustering.

2.1 Hadoop Distributed File System

Prima di entrare nel merito di Hadoop Distributed File System (HDFS), è doveroso spendere alcune parole riguardo il progetto Apache Hadoop di cui esso fa parte. Citando la definizione fornita dalla Apache Software Foundation, Apache Hadoop è «un framework open source che consente il processamento di grandi quantità di dati in modo distribuito, su cluster e sfruttando semplici modelli di programmazione».[15]

Se usato informalmente, il termine Hadoop può essere ambiguo e riferirsi sia al progetto Hadoop in senso stretto, che all'ecosistema Hadoop. Il primo è composto da tre moduli principali: HDFS, che è un filesystem distribuito, MapReduce, un sistema di processamento parallelo, e YARN, un gestore delle risorse del cluster. Quando si parla di ecosistema Hadoop, invece, ci si riferisce, oltre che ai tre moduli di cui sopra, ad un'ampia lista di componenti software tra loro integrabili che sono stati sviluppati nel corso degli anni. Tra questi figurano anche Apache Kafka e Apache Spark, la cui trattazione è demandata ai successivi paragrafi.

Figura 2.1 – Il logo di Hadoop Distributed File System

2.1.1 Architettura

Come detto, HDFS è il file system distribuito su cui si basa il progetto Hadoop. La sua funzione è quella di salvare grandi quantità di dati sotto forma di file, tipicamente di dimensioni dell'ordine dei Gigabyte o Terabyte[16], e di farlo in maniera distribuita su un cluster di più nodi. HDFS è organizzato con un'architettura master-slave: un cluster HDFS, infatti, risulta essere composto da un singolo NameNode e da un numero *n* di DataNode, con *n* tipicamente uguale al numero di nodi del cluster. Tutti i nodi del cluster HDFS sono connessi tra loro e possono comunicare mediante protocollo TCP.[17]

Il NameNode ricopre il ruolo di *master* e coordinatore del cluster. I suoi compiti principali sono:

- memorizzare i metadati del filesystem.
- controllare lo stato di salute dei DataNode: periodicamente i DataNode inviano un heartbeat al NameNode mediante un handshake TCP. Ogni dieci heartbeat poi, i DataNode inviano anche un *block report*, comunicando al NameNode di quali dati si trovano in possesso.
  In questo modo il NameNode potrà avere sempre la situazione del cluster sotto controllo e attuare provvedimenti nel caso in cui lo stato attuale del cluster non rispetti le specifiche desiderate.
- coordinare le operazioni di lettura e scrittura dei dati.

I DataNode, invece, svolgono il ruolo di *slave* e hanno il compito di memorizzare e gestire i dati applicativi veri e propri, interfacciandosi direttamente con i client HDFS per le letture e scritture. Per maggiori dettagli relativi al funzionamento dei processi di lettura e scrittura dei dati si rimanda ai sotto paragrafi 2.1.4 e 2.1.5.

2.1.2 Organizzazione dei dati

Essendo un filesystem, HDFS gestisce dati sotto forma di file. Questi, prima di essere memorizzati vengono sottoposti, nell'ordine, ad un processo di partizionamento in *blocchi*, ad una successiva distribuzione ed infine ad una replicazione.

La dimensione del blocco ha un valore di default pari a 128Mb ma il client HDFS può personalizzarla specificando un valore potenzialmente diverso per ogni file che desidera salvare. Ovviamente può succedere che un file abbia una dimensione non divisibile per la dimensione del blocco stabilita. In questo caso l'ultimo blocco avrà una dimensione uguale al resto della divisione. A titolo di esempio: se si desidera memorizzare un file di 129Mb, questo verrà partizionato in due blocchi rispettivamente di 128Mb e 1Mb[18]. La scelta di non dedicare 128Mb anche all'ultimo blocco permette di non sprecare spazio di archiviazione. Seppur lo spazio che viene risparmiato considerando un unico blocco è poco significativo, questa scelta

architetturale diventa molto vantaggiosa se si tiene presente che HDFS gestisce file di grandi dimensioni, costituiti quindi da un elevato numero di blocchi.

Come detto in precedenza, al partizionamento in blocchi segue un processo di distribuzione: ciascun blocco viene distribuito e memorizzato, se possibile, su un DataNode diverso. Infine, ciascun blocco viene replicato[15] e le repliche distribuite in altrettanti DataNode.

La scelta di come allocare i blocchi tra i vari DataNode spetta al NameNode. Esso, infatti, crea un mapping dove a ciascun blocco associa la lista dei DataNode su cui memorizzare lui e le sue repliche.

Presupposto che le comunicazioni tra server appartenenti allo stesso rack[16] sono più efficienti rispetto alle comunicazioni tra rack differenti, in quanto è disponibile una larghezza di banda maggiore, un'opzione di allocazione efficiente sarebbe quella di salvare tutte le repliche in DataNode dello stesso rack. Questo approccio, tuttavia, non garantirebbe tolleranza al guasto dell'intero rack ed è per questo che il NameNode agisce in modo diverso.

A supporto della strategia di allocazione, infatti, viene utilizzato un algoritmo che è *rack awareness*, in quanto il NameNode conosce l'id del rack a cui appartiene ciascun DataNode.

L'algoritmo rack awareness richiede il salvataggio di due repliche su due DataNode dello stesso rack mentre la terza viene salvata su un DataNode appartenente ad un rack differente (sotto l'ipotesi di un fattore di replicazione pari a tre). In questo modo si riesce a garantire resilienza dei dati minimizzando le scritture inter-rack[16].

Un'altra importante caratteristica di HDFS è che i file da esso gestiti sono del tipo *write-once-read-many*. Si assume, cioè, che una volta che il file è stato creato, scritto e chiuso sul filesystem non possa più essere modificato se non aggiungendovi del contenuto in coda *(append)* o svuotandolo del suo contenuto senza eliminarlo *(truncate)*[16]. Questa assunzione permette di gestire in maniera più semplice i problemi di coerenza tra le varie repliche dei dati, poiché esclude il caso di modifiche arbitrarie al contenuto del file.

---

[15] Il numero di repliche è detto fattore di replicazione. Ha un valore che a default è uguale a tre ma, come per la dimensione del blocco, il client HDFS può configurarlo a livello di file. Dal momento che il namenode non permette di salvare più di una replica dello stesso blocco sullo stesso DataNode, il numero massimo di repliche che è possibile richiedere è pari al numero totale di Datanode.[16]

[16] Tipicamente i server appartenenti a cluster di grandi dimensioni vengono organizzati in rack. Un rack, in informatica e nell'ambito delle telecomunicazioni, è un sistema standard d'installazione fisica di componenti hardware (es. server, switch, router) a scaffale.

2.1.3 Caratteristiche

Le modalità di organizzazione dei dati appena descritte permettono ad HDFS di offrire le seguenti peculiarità:

- **Alta disponibilità dei dati e fault tolerance:** HDFS è stato progettato per eseguire su commodity hardware[17], in condizioni quindi, dove il crash di un server è un evento comune. Per queste ragioni è di primaria importanza assicurare la resilienza dei dati e HDFS la garantisce grazie alla replicazione dei blocchi.

  Nel caso in cui un DataNode dovesse guastarsi, infatti, gli stessi dati saranno ancora disponibili in almeno un altro DataNode. Inoltre, grazie al protocollo di heartbeat citato in precedenza, il NameNode si accorgerà del guasto e provvederà a richiedere la ricreazione di tutte le repliche di blocchi andate perse. In questo modo verrà sempre garantita la presenza di un numero di repliche consistente con il fattore di replicazione.

  Nel caso dovesse fallire un intero rack di nodi, la distribuzione delle repliche condotta in modo *rack awareness*, garantisce ancora una volta resilienza dei dati.

  È possibile, inoltre, garantire disponibilità del servizio anche a fronte di un fallimento del NameNode in quanto HDFS permette di predisporre un NameNode secondario.
- **Alto throughput:** grazie alla distribuzione dei blocchi su più nodi, HDFS garantisce un elevato throughput[18] poiché è possibile parallelizzare operazioni di processamento del file suddividendo il lavoro tra i vari DataNode.

  Il processamento in parallelo permette di ridurre in maniera consistente sia il tempo di lettura che di scrittura di file di grandi dimensioni se confrontato con la medesima operazione condotta su un filesystem tradizionale non distribuito.

  Sempre se confrontato sempre con un filesystem tradizionale, HDFS presenta però una maggiore latenza in quanto la lettura del file non avviene in modo diretto ma con l'intermediazione del NameNode. Nei sistemi di storage per Big Data, tuttavia, è preferibile garantire un throughput elevato piuttosto che una bassa latenza.
- **Scalabilità:** dal momento che HDFS è progettato per eseguire su un sistema distribuito di nodi è immediato garantire scalabilità. Nel momento in cui vi sia la necessità di maggiori

[17] Server economici, facili da ottenere anche in grandi quantità.
[18] Con il termine *throughput* si intende la quantità di dati che è possibile processare/trasmettere in un'unità di tempo. Si differenzia dalla *latenza*, che invece indica il lasso di tempo che intercorre tra l'istante in cui viene fatta una richiesta e l'istante in cui viene ricevuta la prima unità di dati in risposta.

risorse, quali ad esempio maggiori capacità di storage, sarà sufficiente infatti aggiungere al cluster nuovi server con il ruolo di DataNode.

2.1.4 Operazione di scrittura

L'operazione di scrittura di un file su HDFS avviene in due fasi che vedono inizialmente il client interagire con il NameNode e successivamente interfacciarsi con un DataNode.

Nella *prima fase* il client invia una richiesta di scrittura al NameNode il quale, dopo aver controllato che il client possieda i privilegi necessari, decide in quanti blocchi suddividere il file, quante repliche devono essere create per ciascun blocco e in quali DataNode salvarle.

Il NameNode, quindi, invia tutte queste informazioni al client allegando anche un security token che potrà essere usato per autenticarsi presso i DataNode.

A questo punto inizia la *seconda fase*, in cui il client si rivolge ai DataNode per eseguire la scrittura del file. Consultando il mapping blocco-DataNode ricevuto, per ogni blocco il client contatta il primo DataNode della lista, inviandogli sia i dati da scrivere sulla memoria di massa sia la lista degli altri DataNode designati per la replicazione. La replicazione del blocco avviene a cascata, di DataNode in DataNode. Per riportare un esempio, con un fattore di replicazione tre il DataNode contattato dal client avvierà una pipeline di replicazione contattando il secondo DataNode. Questi salverà una replica del blocco e farà lo stesso contattando il terzo DataNode designato. Se anche la terza scrittura si conclude correttamente, la pipeline di replicazione viene ripercorsa in direzione inversa da una serie di acknowledgment, fino a tornare al client.

Avendo avuto la conferma dell'avvenuta replicazione, il client contatta per l'ultima volta il NameNode. Il NameNode aggiorna i propri metadati e dichiara conclusa la scrittura.

In Figura 2.2 viene illustrata la sequenza di interazioni fin qui descritte. Si noti come il client HDFS si avvale di due oggetti Java che fungono da intermediari per la comunicazione con il NameNode (Distributed FileSystem) e con i DataNode (FSDataOutputStream).

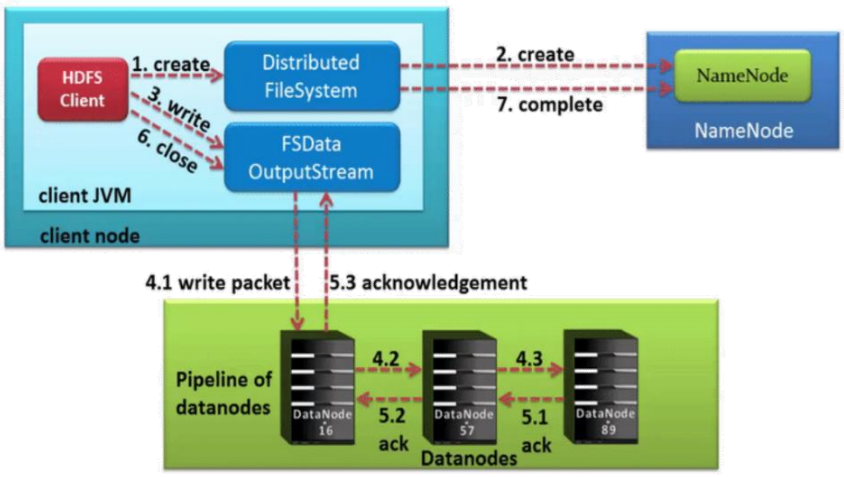


Figura 2.2 – Sequenza delle azioni coinvolte nell'operazione di scrittura di un blocco su HDFS. La stessa sequenza viene ripetuta e svolta in parallelo per tutti i blocchi che il client desidera scrivere.

2.1.5 Operazione di lettura

Come descritto per la scrittura, anche l'operazione di lettura di un file da HDFS avviene in due fasi che vedono il client interagire in prima istanza con il NameNode e successivamente con i DataNode.

Nella *prima fase* il client contatta il NameNode specificando il nome del file che desidera leggere. Il NameNode controlla che il client abbia i privilegi necessari per farlo e, in caso affermativo, risponde fornendo, per ogni blocco da leggere, l'elenco dei DataNode in cui sono salvate le sue varie repliche. Il NameNode, inoltre, invierà al client anche un security token che potrà essere utilizzato per autenticarsi presso i DataNode.

Per ridurre al minimo il consumo di larghezza di banda e la latenza della lettura, HDFS indirizza le richieste di lettura dei blocchi ai DataNode che possiedono la replica più vicina al client. Ad esempio, se dovesse esistere una replica sullo stesso rack del nodo client, verrebbe scelta quella per soddisfare la richiesta di lettura.[16]

Nella *seconda fase* il client invia una richiesta di lettura dei dati contattando, per ogni blocco, il DataNode più vicino su cui esso è presente.

In Figura 2.3 viene illustrata la sequenza di interazioni fin qui descritte. Analogamente all'operazione di scrittura, il client HDFS si avvale di due oggetti Java che fungono da intermediari per la comunicazione con il NameNode (Distributed FileSystem) e con i DataNode (FSDataInputStream).

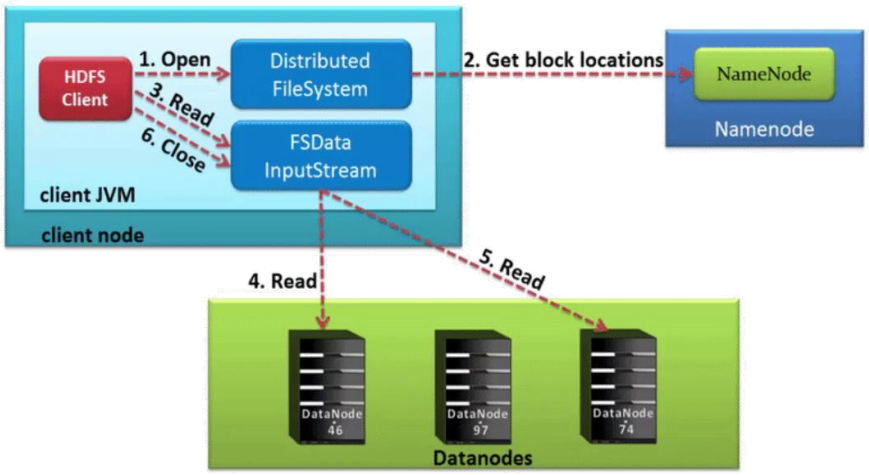


Figura 2.3 – Sequenza delle azioni coinvolte nell'operazione di lettura di un blocco da HDFS. La stessa sequenza viene ripetuta e svolta in parallelo per tutti i blocchi che il client desidera leggere.

2.2 Apache Kafka

Apache Kafka è una piattaforma open source per lo streaming e l'importazione di dati che appartiene all'ecosistema Hadoop. Scritta in Java e Scala e sviluppata dall'Apache Software Foundation, nel tempo è stata adottata da importanti aziende, tra cui LinkedIn, Twitter e Netflix.[19]

Kafka nasce con l'intento di supportare i casi d'uso in cui si ha la necessità di far fluire grandi volumi di dati da una o più sorgenti verso una o più destinazioni. Le sorgenti possono essere le più svariate: sensori IoT, dispositivi mobili, applicazioni software o anche database. Analogamente, le destinazioni possono essere eterogenee nel tipo e nella tecnologia.

In Kafka il singolo dato viene chiamato *evento* (o anche record o messaggio) e rappresenta un "qualcosa che è accaduto" nel dominio specifico in cui ci si trova.[20]

Concettualmente un evento è formato da una chiave accompagnata da un valore, da un timestamp che indica il momento in cui è stato prodotto e, seppur siano opzionali, da alcuni header con metadati.

I record vengono interpretati da Kafka come semplici array di byte. Ne consegue che un record può rappresentare un qualsiasi tipo di dato in un qualsiasi tipo di formato, sia strutturato (e.g. JSON, Avro ecc.) che non strutturato (e.g. PlainText).

Nella terminologia Kafka, le sorgenti di eventi sono dette *produttori*. Essi serializzano i record e li pubblicano sui server Kafka. Da questi potranno essere letti dai *consumatori*, previa un'opportuna de-serializzazione.

- Event key: "Alice"
- Event value: "Made a payment of $200 to Bob"
- Event timestamp: "Jun. 25, 2020 at 2:06 p.m."

Figura 2.4.2 – Un esempio di evento in Kafka

Figura 2.4.1 – Il logo di Apache Kafka

2.2.1 Architettura, principi di funzionamento e proprietà

Kafka è un sistema di messaggistica distribuito. È composto infatti da uno o più nodi server, detti *broker*, che hanno il compito di memorizzare gli eventi pubblicati, di processarli se lo si desidera, e di renderli disponibili ai consumatori interessati.

I messaggi pubblicati in Kafka vengono organizzati e suddivisi in *topic*, così da permettere il processamento da parte dei consumatori sulla base di uno specifico interesse. Una topic è realizzata con semantica *molti a molti* poiché può avere zero o più produttori che inviano messaggi così come zero o più consumatori ad essa interessati. Una topic è a sua volta suddivisa in una o più *partizioni* che vengono distribuite tra i diversi broker del cluster.

Gli eventi che sono destinati alla stessa topic e che hanno la stessa chiave vengono pubblicati nella stessa partizione. Si noti come questa affermazione non impedisca di salvare eventi con diverse chiavi nella stessa partizione. Si noti anche come questa scelta di design favorisca un processamento efficiente lato destinazione: qualora un consumatore sia interessato agli eventi di una specifica topic e ad una particolare chiave potrà reperirli tutti presso lo stesso broker.

Le partizioni vengono inoltre replicate secondo un fattore di replicazione[19] e distribuite tra i broker del cluster. Per ogni partizione viene designato un broker *leader* che sarà il nodo su cui insisteranno le letture e le scritture relative a quella partizione. I broker che invece ospitano le repliche di quella partizione vengono detti *followers*.

È importante evidenziare due aspetti. Innanzitutto, perché il sistema sia efficiente, è necessario che i broker del cluster siano leader di un numero equo di partizioni, in modo da poter bilanciare il carico. Kafka provvede a distribuire equamente le partizioni e le leadership tra i vari broker, tuttavia, è importante che rimangano equi partite anche a seguito di guasti e conseguenti nuove elezioni. Potrebbe altrimenti succedere che un broker leader smetta di funzionare e al suo riavvio venga riconosciuto come semplice follower per tutte le partizioni che possiede. Chiaramente questa situazione sarebbe inefficiente poiché quel nodo verrebbe usato solo come server di backup, senza poter essere sfruttato attivamente per le letture o scritture.

Per questo motivo, Kafka effettua periodicamente un ribilanciamento delle leadership tentando di riassegnare, in caso di riavvio, il titolo di leader al broker che lo aveva perso.[21]

La seconda ma non meno importante considerazione riguarda la necessità di sincronizzazione tra le repliche. Al momento della scrittura di nuovi record nella partizione leader sarà necessario

[19] Il fattore di replicazione indica quante repliche di una partizione devono essere presenti. È configurabile a livello di topic ed è tipicamente pari a tre.

aggiornare anche le sue repliche in modo che dispongano sempre degli stessi messaggi della partizione leader, secondo lo stesso ordine. Nel processo di sincronizzazione i follower agiscono come se fossero dei normali consumatori Kafka.[22]

Per maggiori dettagli sul funzionamento dei consumatori si rimanda al sottoparagrafo 2.2.3, mentre qui sotto, in Figura 2.5 sono riassunti graficamente i concetti fino ad ora espressi.

Per concludere si precisa che la gestione del cluster Kafka e dei problemi tipici dei sistemi distribuiti (sincronizzazione e locazione delle repliche, monitoraggio dello stato di salute, elezioni e rielezioni dei leader) viene demandata a Zookeeper, un servizio integrato con Kafka.

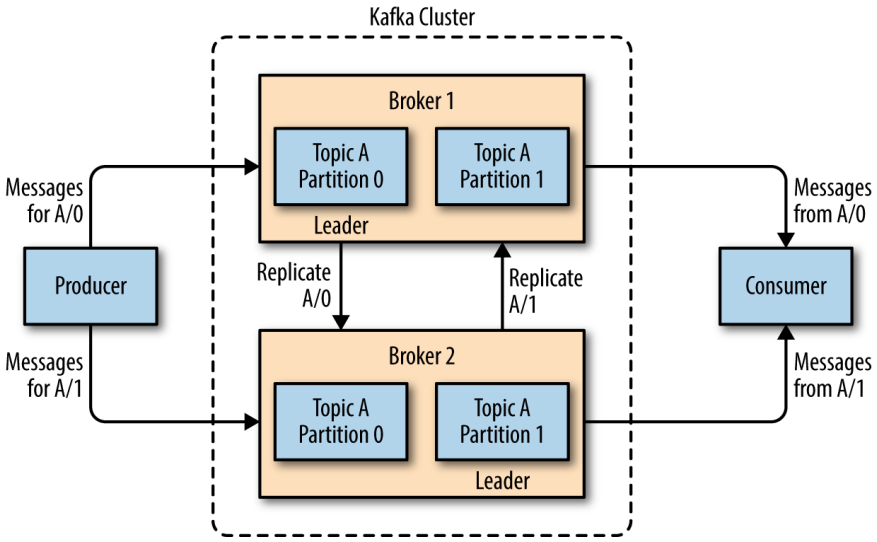


Figura 2.5 – Un cluster Kafka formato da due broker. È presente una topic di nome A composta da due partizioni e avente fattore di replicazione pari a due. Si noti come il produttore (consumatore) si interfacci con il broker leader della topic su cui vuole pubblicare (consumare) il record.

Grazie alle scelte di design sopra descritte e con cui è stato progettato, Kafka offre i seguenti benefici:

- **Decoupling tra produttori e consumatori:** Kafka è un sistema di messaggistica che si interpone tra produttori e consumatori di eventi creando un disaccoppiamento tra queste due entità. Il disaccoppiamento tra entità software è molto citato in letteratura per via dei benefici che comporta. Grazie ad esso sia l'invio che il consumo di un messaggio, ad esempio, possono avvenire in modo asincrono: il produttore può inviare i propri eventi e terminare l'operazione senza dover aspettare il consumo degli stessi. Il consumatore può leggere i messaggi nel momento in cui preferisce.

  Inoltre, produttori e consumatori sono *fully agnostic*: il "come" sono realizzati (tecnologie, linguaggi…) è irrilevante ai fini del funzionamento del sistema. L'unica conoscenza

reciproca che possiedono è il formato degli eventi con cui comunicano. Questo aspetto permette ampia libertà nelle scelte di sviluppo.

- **Alta disponibilità dei dati, fault tolerance e persistenza:** Kafka garantisce la continua disponibilità dei propri dati anche in caso di guasti ai singoli nodi grazie al meccanismo di replicazione delle partizioni in diversi broker del cluster.

  Qualora il broker leader dovesse subire un guasto o non essere più contattabile, i messaggi in esso salvati saranno ancora disponibili presso le altre repliche e uno dei followers verrà eletto come nuovo leader.[23]

  Inoltre, i messaggi che vengono pubblicati in una topic Kafka non vengono eliminati nemmeno dopo essere stati letti[20]. A differenza di altri sistemi di messaging tradizionali, essi vengono memorizzati persistentemente e in modo affidabile per un periodo di *retention* che è possibile specificare a piacimento e a livello di topic. Solo una volta trascorso questo tempo, gli eventi diventati obsoleti vengono eliminati dal sistema.
- **Scalabilità:** la suddivisione delle topic in diverse partizioni e la loro distribuzione in diversi broker del cluster garantisce scalabilità e alto throughput nelle operazioni di lettura e scrittura. Se infatti una specifica topic venisse interamente gestita da un singolo broker, i consumatori e produttori ad essa relativi sarebbero costretti a centralizzare le proprie letture e scritture sullo stesso broker, adottando un comportamento non scalabile.

  La distribuzione delle partizioni, contrariamente, fa sì che il consumo (produzione) di eventi su una singola topic possa essere condotta in parallelo su diversi broker.

  Per esemplificare quanto appena detto si confrontino le seguenti due configurazioni Kafka:
  - Cluster che presenta 1 topic con 1 partizione.
  - Cluster che presenta 1 topic suddivisa in 4 partizioni (e.g. Figura 2.6).

  Nella prima situazione, qualora un consumatore o gruppo di consumatori desiderasse leggere i messaggi presenti sulla topic, le operazioni di lettura insisteranno sull'unico broker leader di quella partizione e verranno di conseguenza sequenzializzate. Analogamente accadrà per la pubblicazione di messaggi da parte di un produttore. Ipotizzando un tempo *t* per leggere un messaggio M dalla topic, la lettura di 4 messaggi richiederà un tempo pari circa a 4 *t*.

Nella seconda situazione invece, illustrata anche in Figura 2.6, le 4 partizioni della topic vengono distribuite tra quattro diversi broker del cluster permettendo di parallelizzare le letture e scritture. Ipotizzando di avere un gruppo di quattro consumatori, interessato a leggere 4 messaggi M1, M2, M3 e M4 appartenenti ciascuno ad una diversa partizione della topic, la lettura dei 4 messaggi avverrà in parallelo richiedendo un tempo complessivo circa uguale a *t*.

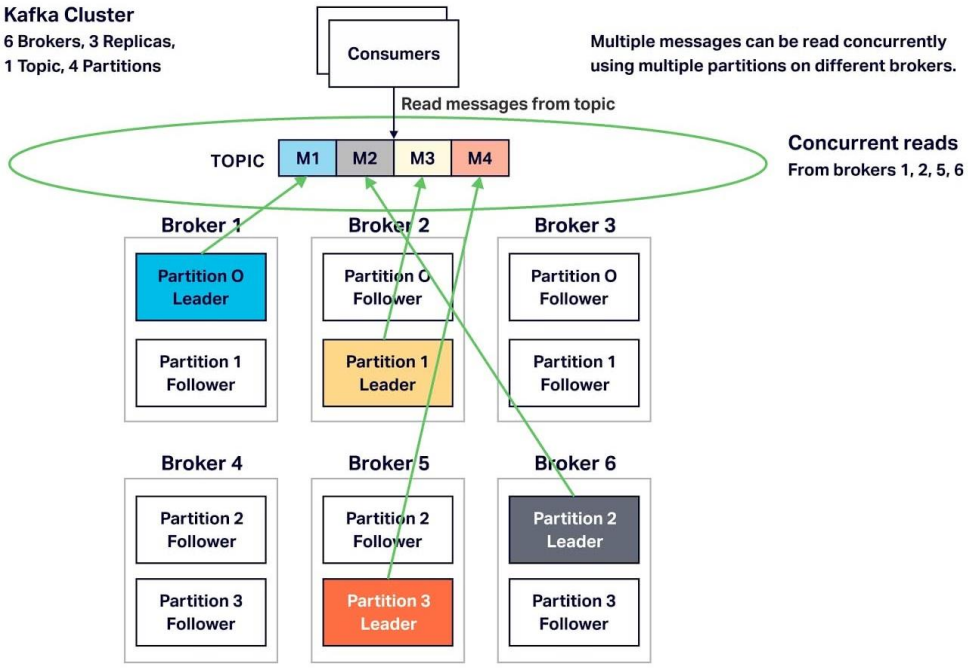


Figura 2.6 – Il partizionamento delle topic e la loro distribuzione su più broker rende possibile una lettura dei record condotta in parallelo.

2.2.2 Modello di produzione dei messaggi

Come accennato in precedenza, il produttore Kafka produce eventi e li pubblica presso il broker leader della partizione designata per quella topic e chiave[20]. Per sapere quale broker è il leader di partizione, il produttore instaura una prima connessione con un *bootstrap server* scelto tra i broker. Esso risponderà con i metadati del cluster, tra cui la lista partizioni-leader.

[20] Un errore compiuto comunemente è quello di programmare il produttore perché invii record sempre con la stessa chiave o con chiave null. In questo modo tutti i messaggi vengono pubblicati nella stessa partizione precludendo la possibilità di sfruttare il parallelismo delle letture e scritture offerto da Kafka.[24]

Il produttore potrà così inviare il record al leader della partizione e questi gli assegnerà un offset[21] univoco. I broker follower, se presenti, possono a questo punto consumare il nuovo messaggio per mantenere la propria partizione sincronizzata con quella del leader.
La scrittura del messaggio viene considerata conclusa correttamente solo dopo che tutti i broker follower, o al minimo un numero configurabile *min.insync.replicas* di broker follower, si sono sincronizzati con il leader. Fino a che la scrittura non è conclusa correttamente, il record non viene reso disponibile per il consumo.[24]

2.2.3 Modello di consumo dei messaggi

In Kafka un consumatore può agire in maniera indipendente oppure come membro di un gruppo di consumatori che leggono i record in parallelo.
Nel primo caso il consumatore invia una richiesta di lettura al broker leader della partizione da cui vuole leggere specificando l'offset da cui iniziare a recuperare i record.
Nel secondo caso, invece, i consumatori appartenenti allo stesso gruppo hanno la necessità di coordinarsi per far sì che i messaggi vengano letti una volta sola. La coordinazione si ottiene dalla garanzia che una partizione non possa essere assegnata a due consumer dello stesso gruppo. Si avrà in questo modo un offset specifico per ogni partizione e consumatore, e le varie letture potranno avvenire in parallelo. Per evitare che alcuni consumatori rimangano inattivi, il numero massimo di consumatori appartenenti allo stesso gruppo è pari al numero di partizioni da cui si desidera leggere.[25]
È possibile anche definire più gruppi di consumatori, a seconda delle esigenze. La scelta di utilizzarne uno solo nasce dall'esigenza di processare i record una sola volta sfruttando il parallelismo e l'alto throughput che il gruppo garantisce. Se si vuole invece beneficiare del parallelismo, ma si dovesse avere l'esigenza di processare più volte gli stessi dati, diventa necessario definire più gruppi in modo che la stessa partizione possa essere assegnata a più consumatori, ciascuno con il proprio offset.[25]
Prima di proseguire è importante notare e soffermarsi su due aspetti:

- è il consumatore ad avere il controllo su quali messaggi leggere.

[21] Per evitare fraintendimenti si noti che è possibile fare riferimento al termine offset in due contesti differenti. Se si parla di offset di una partizione si intende il puntatore all'ultimo record che è stato scritto su quella partizione. Questo permette di assegnare ad ogni record un numero intero univoco che rappresenta la sua posizione nella topic. Se invece di parla di offset nel contesto del consumatore, o *consumer offset*, ci si riferisce alla posizione da cui il consumatore desidera iniziare a leggere i dati dalla partizione.

- è il consumatore che prende l'iniziativa della lettura, seguendo una semantica di tipo *pull*.

Il primo aspetto gli dà la libertà, qualora necessario, di richiedere la lettura di messaggi già letti semplicemente specificando un offset antecedente.

La semantica pull invece comporta due vantaggi.[26] Innanzitutto, il consumatore può scegliere quando richiedere nuovi dati basandosi sul proprio stato attuale di carico. Diversamente, dal momento che Kafka può trovarsi ad interagire con consumatori aventi prestazioni difformi, una semantica push potrebbe causare situazioni in cui a fronte di un data rate elevato alcuni consumatori vengano sovraccaricati di dati che non sono in grado di gestire.

Il secondo vantaggio offerto dalla semantica pull è che il consumatore può processare più record alla volta aggregandoli in batch. Al momento del consumo, infatti, potrà leggere tutti i nuovi record a partire dall'offset specificato in una singola connessione. Con una semantica push, invece, sarebbe il broker a decidere se inviare ogni singolo record non appena questo viene ricevuto oppure se aspettare di accumularne alcuni per poi inviarli in batch. E in questa decisione sarebbe inevitabile un trade off di performance – latenza.

Semantica di consegna

Kafka permette di ottenere una semantica di consegna di tipo *exactly-once.* Ciascun record, quindi, viene consumato esattamente una volta da ciascun consumatore o gruppo di consumatori.

La semantica viene garantita anche in caso di fallimento del consumatore in quanto l'offset di consumo viene salvato in una topic dedicata e aggiornato solo dopo che il processamento del record è stato completato. Inoltre, processamento del record e aggiornamento dell'offset vengono eseguiti all'interno di un'unica transazione. In questo modo, se il consumatore dovesse riscontrare un guasto durante la lettura del record, il suo processamento o durante l'aggiornamento dell'offset, l'intera transazione abortirebbe e l'offset verrebbe ripristinato al suo precedente valore, garantendo letture consistenti anche da parte di un eventuale nuovo consumatore.[27]

Un altro importante aspetto riguardante la semantica di consegna è l'ordinamento temporale, che viene garantito solo a livello di singola partizione. Ciò significa che i messaggi con la stessa chiave verranno letti nello stesso ordine cronologico con cui sono stati pubblicati in quanto vengono salvati nella stessa partizione. L'ordinamento temporale assoluto, invece, non è garantito dal momento che ogni partizione viene letta indipendentemente dalle altre.

2.2.4 Kafka Connect

Come specificato nel paragrafo introduttivo, Kafka viene utilizzata per il seguente caso d'uso: una o più sorgenti di dati pubblicano eventi che vengono consumati da una o più destinazioni. Kafka, quindi, può trovarsi ad interagire con un'ampia eterogeneità di sistemi esterni sia in fase di importazione che di esportazione dei dati (database relazionali, NoSQL, filesystem tradizionali, filesystem distribuiti, data warehouse, sistemi di log o di metric, ecc.). Fortunatamente, nel corso degli anni la comunità di sviluppatori ha realizzato un vasto ecosistema di Kafka Connect, applicazioni che supportano l'integrazione tra Kafka e i sistemi esterni.[28] In questo modo lo sviluppatore non si troverà a dover "reinventare la ruota" ogni volta che deve interfacciarsi con Kafka. Potrà utilizzare il relativo Kafka Connect già sviluppato e configurarlo perché si adatti alle proprie esigenze.

I benefici derivanti dall'utilizzare un Kafka Connect sono dunque evidenti. Primo fra tutti la possibilità di concentrarsi sulla logica specifica del dominio, con conseguente riduzione del time-to-market.

I Kafka Connect sono applicazioni che agiscono come clienti Kafka; dal suo punto di vista, infatti, non sono altro che semplici produttori e consumatori. Possono avere lo scopo di importare dati da una sorgente verso Kafka o viceversa di esportarli da Kafka verso una destinazione. Nel primo caso si parla di *source connector*, nel secondo di *sink connector*.

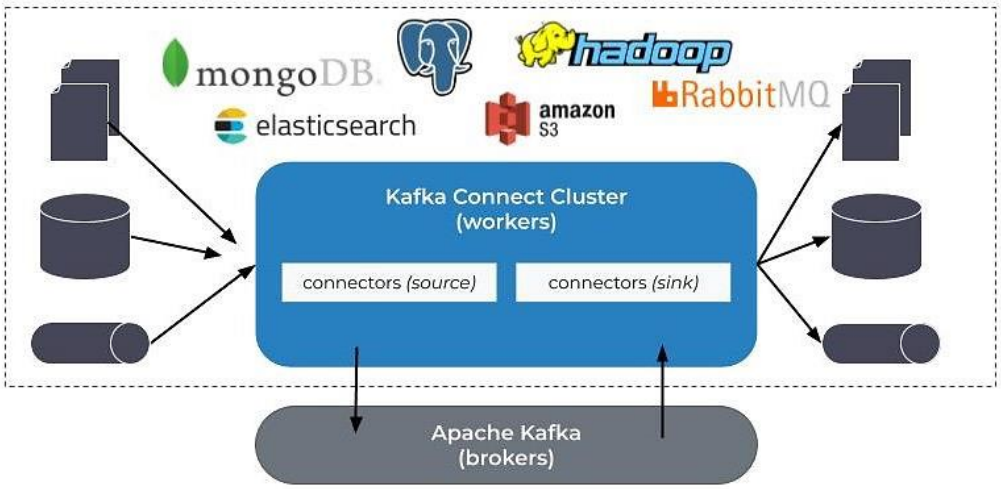


Figura 2.7 – L'ecosistema Kafka Connect è composto da una vasta famiglia di source e sink connector che permettono l'integrazione tra Apache Kafka e i sistemi esterni. In figura è riportata una lista non esaustiva di possibili sistemi esterni.

Componenti logici di un Kafka Connect

Se da un lato è vero che è possibile usare un Kafka Connect esistente senza dover scrivere del codice, dall'altro è indispensabile sapere come è realizzato per poterlo configurare e utilizzare in modo consapevole. Un Kafka Connect è formato dai seguenti componenti logici:[29]

- **Connector plugin:** è l'insieme delle classi utilizzate dal connector[22] per funzionare. Tipicamente si tratta di un JAR ed è l'unica parte del componente *technology-dependent* in quanto fa riferimento allo specifico sistema esterno con cui si desidera interagire. Le classi contenute nel JAR realizzano l'interfacciamento verso lo specifico sistema esterno.
  Nel caso di un source connector che legge da un sistema esterno, il connector plugin creerà un *connect record*, cioè una generica rappresentazione del dato appena letto. Il connect record verrà poi passato al componente Transformer.

- **Transformer:** l'utilizzo dei transformer è opzionale. L'utente può configurare il Kafka Connect perché utilizzi zero o più transformer in pipeline tra loro, a seconda delle proprie esigenze di business. I transformer permettono di compiere semplici trasformazioni a livello di singolo connect record, come per esempio operazioni di filtraggio o casting.

- **Converter:** è il componente utilizzato dal connector per interfacciarsi con il cluster Kafka. Dal momento che un record Kafka è un semplice array di byte, il converter si occupa della serializzazione del connect record prima di pubblicarlo su Kafka, nel caso di un source connector, oppure della sua lettura da Kafka e della de-serializzazione, nel caso di un sink connector. Esistono diversi converter che offrono supporto alla conversione da e verso i più comuni formati di dati (Avro, JSON, Protobuf…).

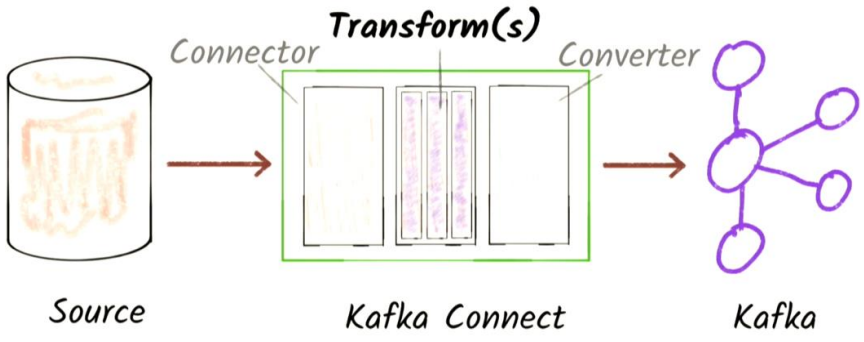


Figura 2.8 – I componenti di un Kafka Connect di tipo source connector. Si noti che per il sink connector i componenti sono analoghi ma il flusso di dati scorre nel verso opposto.

[22] Il termine *connector* viene spesso utilizzato come abbreviazione ambigua per riferirsi a due concetti distinti: *connector instance* e *connector plugin*. La prima è l'istanza logica che si occupa di copiare i dati da/verso Kafka. Il connector plugin invece è l'insieme delle classi utilizzate dalla connector instance per funzionare.

Deploy di un connector

È bene precisare che i Kafka Connect non fanno parte del cluster Kafka ma appartengono ad un cluster logicamente distinto. È necessario quindi eseguire una procedura di deploy dedicata. Il deploy di un Kafka Connect può avvenire in due modalità distinte: *standalone* o *distribuita*. Prima di descriverne le caratteristiche però, è necessario fornire alcune definizioni.
Quando si affronta il deploy di un Kafka Connect le entità in gioco sono le seguenti: [30]

- **Connector instance:** è l'istanza logica che coordina lo streaming dei dati da o verso un determinato sistema esterno coordinando un insieme di *task*. È necessaria una connector instance diversa per ogni tipo di sistema esterno con cui ci si desidera interfacciare.
- **Task:** sono i principali protagonisti del modello Kafka Connect perché hanno il compito di copiare e trascrivere concretamente i dati. Per garantire parallelismo, la connector instance ha la facoltà di suddividere un singolo Job di streaming dati in sotto compiti, e di assegnare ciascuno ad uno dei task che gestisce.
- **Worker:** le connector instance, con i relativi task sono unità logiche di lavoro che hanno la necessità di essere eseguite all'interno di un processo. Questo processo è chiamato worker e può essere pensato come l'entità di più alto livello che costituisce un Kafka Connect. Per questo motivo quando si parla di deploy di un Kafka Connect ci si riferisce al deploy dei worker.

Nella modalità di *deploy standalone* si ha un singolo worker che esegue tutte le connector instance e tutti i task. Tra le due modalità è la più semplice ma non garantisce né tolleranza ai guasti né scalabilità, invece offerte dalla modalità di *deploy distribuita*.
Quest'ultima consiste nell'eseguire un insieme di worker assegnandoli allo stesso *group id*, in modo che appartengano allo stesso Kafka Connect cluster.
I worker appartenenti allo stesso cluster sono in grado di coordinarsi in modo autonomo per distribuire il carico di lavoro (le connector instance e i task) e ribilanciarlo dinamicamente a fronte di cambiamenti nel numero dei worker (crash, scale-up, scale-down). Per svolgere questi compiti, i worker sfruttano il meccanismo dei gruppi di consumatori Kafka descritti nel paragrafo 2.2.3.[30]

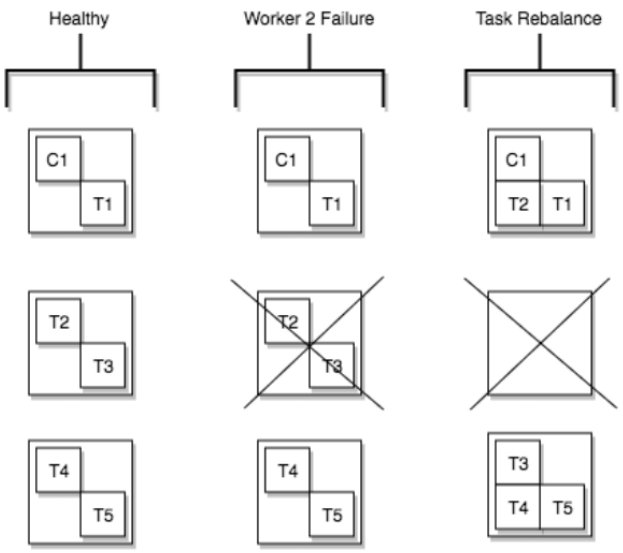


Figura 2.9 – Un esempio di ribilanciamento di carico a seguito del crash di un worker. Il fallimento del worker 2 causa la ridistribuzione dei task T2 e T3 tra gli altri worker attivi.

Sempre in maniera analoga ai consumatori Kafka, i worker eseguiti in modalità distribuita[23] sfruttano delle topic Kafka per memorizzare informazioni come il proprio stato o gli offset di consumo. Queste informazioni, risiedendo in un sistema fault tolerance come Kafka, garantiscono una conseguente fault tolerance anche ai worker distribuiti.

Nella modalità distribuita, i worker espongono delle API tramite interfaccia REST. Possono essere usate dallo sviluppatore per richiedere il deploy di nuove connector instance oppure per modificare, elencare, eliminare e monitorare quelle già esistenti. Vengono inoltre usate anche dagli stessi worker per coordinarsi.

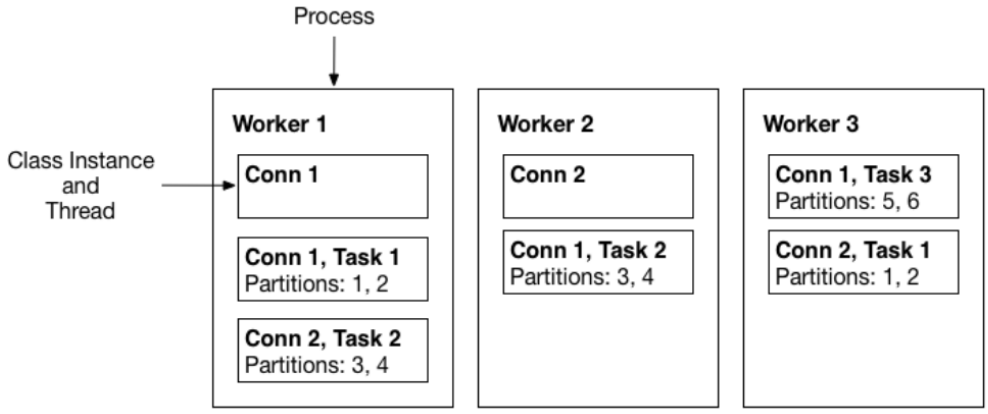


*Figura 2.10 – Un Kafka Connect cluster formato da 3 **worker** e due **connector instance** che gestiscono rispettivamente tre e due **task**. A ciascun task vengono assegnate delle partizioni da cui leggere (sink connector) o su cui scrivere (source connector). Si noti come il carico, rappresentato dalle connector instance e dai task, sia bilanciato tra i vari worker del cluster.*

[23] I worker eseguiti in modalità standalone, invece, memorizzano i propri offset di consumo e le informazioni di stato solo localmente. Non sono quindi in grado di sopravvivere ad un eventuale crash del worker.

2.3 Apache Spark

Apache Spark nasce nel 2009 come progetto interno all'Università di Berkeley, California, e successivamente viene donato all'Apache Software Foundation dove diventa un progetto open source appartenente all'ecosistema Hadoop.[34]

Apache Spark è un framework per l'analisi e il processamento distribuito di dati su larga scala. Viene anche definito, dalla documentazione ufficiale, come «unified analytics engine».[35]

Il termine *unified* si riferisce al fatto che Spark è un framework general purpose in grado di unificare sotto il suo utilizzo una vasta gamma di operazioni di analisi e di processamento che spaziano dal batch processing allo streaming processing, dalle più semplici operazioni di mining e trasformazioni dei dati alle più complicate procedure di Machine Learning e graph processing. Spark, infatti, mette a disposizione diverse librerie, tutte basate sullo *Spark core*[24], che possono essere sfruttate a seconda delle esigenze. Le più note sono:

- **MLlib:** una libreria che offre supporto e API di alto livello per realizzare e implementare pipeline di Machine Learning sui dati.
- **Streaming:** una libreria che permette di condurre analisi e processamento real-time sui dati.
- **Spark SQL and DataFrame:** un modulo che consente di effettuare query e processamento su dati strutturati.
- **GraphX:** una libreria che permette di eseguire graph processing, un processamento che analizza i dati considerando le relazioni reciproche. Il graph processing è particolarmente utile in alcuni domini. Un esempio sono i social network (e.g. Facebook) dove i vari profili utenti sono interconnessi da relazioni di amicizia.

Inoltre, Spark è un framework *language agnostic* in quanto offre API di sviluppo in diversi linguaggi di programmazione, tra cui Python, Java, Scala e R.

[24] Lo Spark core è il fondamento del framework. Esso è responsabile della gestione della memoria, dell'interazione con i sistemi di storage esterni ed è colui che garantisce le proprietà principali offerte da Spark. Mette a disposizione una serie di API che nascondo la complessità del framework e ne permettono un utilizzo più ad alto livello.[33]

Spark è anche in grado di interfacciarsi con tantissime sorgenti di dati diverse, tra le quali Kafka e HDFS descritte nei paragrafi precedenti. Può essere eseguito in modalità standalone oppure appoggiarsi al supporto di cluster manager come YARN, MESOS o Kubernetes.

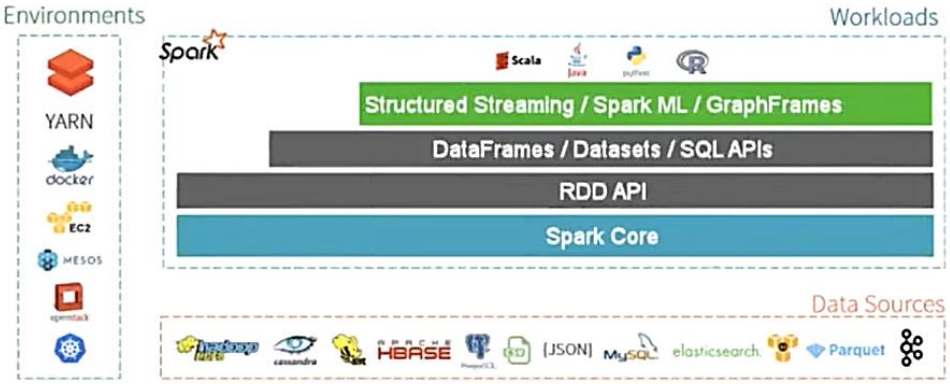


Figura 2.11– Spark come framework unificato di processamento e analisi di dati.

2.3.1 Architettura

Come citato in precedenza, l'obiettivo di Spark è quello di processare dati in modalità distribuita e concorrente e per raggiungerlo sfrutta un'architettura di tipo master-slave. Un'applicazione Spark, infatti, è composta da un processo chiamato *driver program*, o semplicemente *driver*, e da uno o più processi detti *executor*.[36]

Il driver esegue la funzione main dell'applicazione e crea lo SparkContext, un oggetto configurato per connettersi con il *cluster manager* e richiedere la creazione di uno o più executor. Gli executor vengono allocati sui nodi *worker* e su di essi viene distribuito il carico computazionale del processamento dei dati, che può così avvenire in parallelo.

Il driver svolge il ruolo di coordinatore in quanto si premura di inviare il codice dell'applicazione da eseguire agli executor e, tramite lo SparkContext, distribuisce i task[25] da compiere. Il risultato finale delle computazioni, quando richiesto, viene poi restituito al driver.

È possibile fare alcune considerazioni riguardo l'architettura appena descritta. Innanzitutto, ogni applicazione Spark dispone dei propri processi driver ed executor. Da un lato questo garantisce isolamento tra le varie applicazioni. Permette quindi una maggior semplicità a livello di driver, che deve pianificare i task di una sola applicazione, ed evita il rischio di interferenze tra le esecuzioni lato worker, in quanto i task di diverse applicazioni vengono eseguiti in JVM

[25] Si faccia riferimento al sotto paragrafo 2.3.2 per la definizione di task.

differenti. Tuttavia, questa scelta di design implica l'impossibilità di condividere dati tra diverse applicazioni Spark, se non mediante l'utilizzo di un sistema di memorizzazione esterno.[36]
La seconda considerazione è che driver ed executor devono comunicare intensivamente durante il ciclo di vita di un'applicazione. Seppur non sia strettamente obbligatorio, è preferibile effettuare il deploy del driver vicino ai nodi worker, preferibilmente sulla stessa LAN.[36]
In Figura 2.12 viene illustrata l'architettura fin qui descritta.

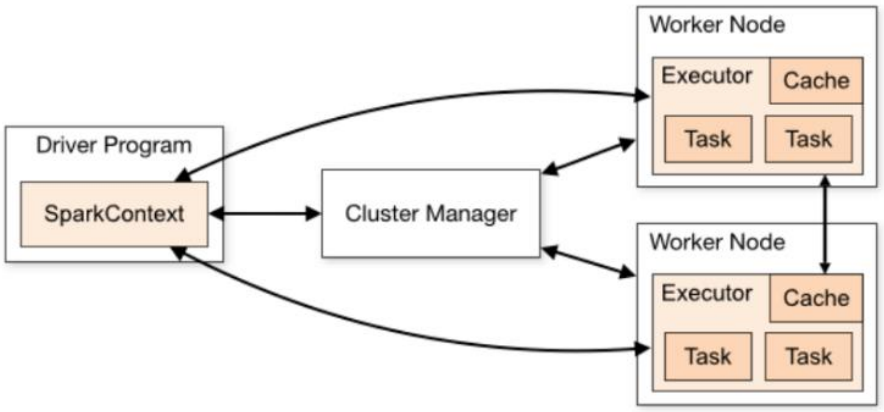


Figura 2.12– Architettura master-slave di Spark. Il driver program si interfaccia, grazie allo SparkContext, con il cluster manager per richiedere l'istanziazione di uno o più executor. Agli executor vengono poi assegnati i task da eseguire.

2.3.2 Terminologia e concetti

Prima di descrivere le peculiarità offerte da Spark è indispensabile fornire alcune definizioni e soffermarsi sui concetti fondanti del framework.

RDD e DataFrame

I Resilient Distributed Dataset (RDD) sono la principale astrazione con cui Spark permette di lavorare. Come dice il nome, sono delle collezioni di elementi partizionate e distribuite tra i vari executor in modo che possano essere processate in parallelo. Il termine *resilient* invece fa riferimento al fatto che sono in grado di sopravvivere a eventi inattesi, come il crash dell'executor su cui si trova una partizione.
Questa proprietà viene garantita dal fatto che Spark è in grado di tenere traccia del lignaggio di ogni RDD e così facendo, qualora dovesse andare persa una singola partizione o l'intero RDD, sarà in grado di garantirne il ripristino chiedendo che venga rieseguita la sequenza di computazioni contenute nel lignaggio. Si noti come tutto questo può funzionare a patto che la sorgente dei dati da cui ha origine il lignaggio sia anch'essa resiliente.

È possibile generare un RDD a partire da una sorgente dati (HDFS, database relazionali, file ecc.), oppure parallelizzando una collezione di dati presente sul driver.[37] In ogni caso l'RDD creato è immutabile. Ciò significa che ogni successiva operazione ad esso applicata genererà un nuovo RDD.[34]

A partire dalla versione Spark 1.6 sono stati introdotti i DataFrame, una nuova astrazione costruita a partire dagli RDD. A differenza di quest'ultimi, però, i DataFrame sono collezioni di dati strutturate, organizzate in colonne e aventi uno schema. Permettono quindi di lavorare più facilmente con insiemi di dati tabellari e relazionali.

Possono essere generati da sorgenti di dati esterne oppure creati a partire da RDD esistenti, offrono API di alto livello, il che li rende più semplici da utilizzare e programmare, e soprattutto sono più efficienti rispetto agli RDD. Le operazioni condotte sui DataFrame, infatti, vengono ottimizzate dietro alle quinte con il supporto del modulo Spark SQL.[38]

Trasformazioni e Azioni

In Spark è possibile eseguire due tipi di operazioni sui dati: trasformazioni e azioni.

Le trasformazioni sono operazioni che vengono eseguite in modo *lazy*. Questo significa che una trasformazione genera un nuovo RDD solo da un punto di vista logico in quanto non viene eseguita nessuna computazione concreta. La trasformazione, o la catena di trasformazioni, applicata alla collezione di dati viene registrata ma attuata solo quando viene richiesta l'esecuzione di un'azione. Le azioni, per contro, sono operazioni che eseguono una computazione su una collezione di dati e restituiscono immediatamente un risultato al driver.

L'esecuzione delle trasformazioni in modalità *lazy* consente a Spark di migliorare le prestazioni del processamento perché dà la possibilità di condurre le ottimizzazioni con maggiore consapevolezza e visione d'insieme. Infatti, temporeggiando sull'esecuzione delle trasformazioni, si viene a delineare l'intero piano d'esecuzione. Osservandolo, Spark potrà individuare l'ordine più efficiente in cui conviene eseguirlo e sulla base di questo modellare il DAG[26] e pianificare i task.

Alcuni esempi di trasformazioni sono le operazioni di groupBy, join e map. Alcuni esempi di azioni, invece, le operazioni collect, count e reduce.

[26] Il Directed Acyclic Graph (DAG) è il grafico che rappresenta l'ordine di esecuzione delle trasformazioni ed azioni in Spark. Nel DAG, ogni entità rappresenta un RDD intermedio. Gli RDD sono collegati tra loro da frecce rappresentanti una trasformazione o un'azione.

Job, Stage e Task

Il processamento dei dati in Spark viene organizzato e suddiviso in più entità logiche, che ora verranno descritte seguendo un approccio top-down.

Nel momento in cui viene richiesta l'esecuzione di un'azione, il driver crea e la associa automaticamente un *job*, ossia l'insieme di tutte le computazioni necessarie a soddisfare quell'azione e le trasformazioni precedenti, se presenti.

Un job può essere poi diviso in *stage*, sottogruppi di una o più computazioni. Il criterio per determinare la divisione in stage è il seguente: ogni volta che è necessario uno scambio di dati tra executor, lo stage corrente termina e ne inizia uno nuovo.[34] Lo scambio di dati tra processi viene detto *shuffle* ed è un'operazione abbastanza onerosa dal punto di vista computazionale in quanto coinvolge operazioni di I/O, di serializzazione e di network I/O.[37]

La singola computazione, infine, viene detta *task*, ed è la più piccola unità di lavoro definibile in Spark. I vari task vengono eseguiti in parallelo dagli executor del cluster.

È possibile riportare un semplice esempio per fissare i concetti appena descritti.[34] Supponiamo di voler contare il numero di record contenuti in un DataFrame: l'azione count, invocata sul DataFrame, scatena la computazione e ad essa viene associato un job, composto da due stage. Nel primo stage i record del DataFrame verranno distribuiti tra i vari executor e a ciascuno verrà assegnato il task di contare i record di cui si trova in possesso.

Si avranno così i conteggi parziali generati dai vari worker e sarà necessario aggregarli. Verrà quindi effettuato uno shuffle in modo da condividere con il worker incaricato dell'aggregazione le informazioni riguardanti le varie somme parziali. Il primo stage, quindi, termina ed ha inizio il secondo, composto dal singolo task di sommare i conteggi parziali. Infine, il risultato finale verrà trasmesso al driver.

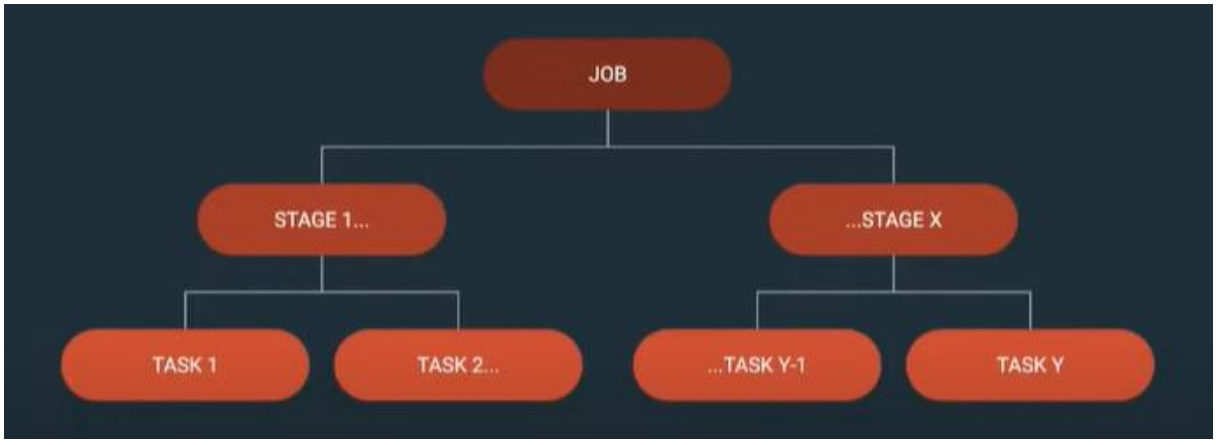


Figura 2.13 – La suddivisione di un job in stage e in task.

Persistenza dei RDD

Quando si esegue un'azione, se non specificato diversamente, viene sempre eseguita l'intera catena di trasformazioni necessarie ad ottenere l'RDD finale su cui poi eseguire l'azione. Questo comportamento risulta inefficiente qualora si abbia la necessità di ripetere la stessa azione più volte, come nel caso di algoritmi iterativi.

Spark, quindi, offre la possibilità di memorizzare un RDD in cache o in memoria sul disco, in modo da poter garantire accessi più rapidi nel corso dei processamenti futuri. Quando si richiede la memorizzazione di un RDD, infatti, ogni nodo salva le partizioni che ha calcolato e le riutilizza per le successive operazioni.[37]

È possibile usare i metodi *cache* o *persist* per richiedere questa funzionalità. Il primo realizza la persistenza sulla RAM, mentre al secondo metodo è possibile passare come argomento uno *storage level* per specificare il tipo di persistenza desiderato. È bene specificare che anche la persistenza di un RDD viene valutata in modalità lazy e realizzata non appena viene eseguita un'azione sul RDD in questione.

Oltre al salvataggio in RAM già citato, e che è anche il comportamento di default, è possibile richiedere un salvataggio parzialmente su RAM e disco oppure interamente su disco.

È possibile, inoltre, specificare una fattore di replicazione per le partizioni da salvare. Come già detto in precedenza infatti, Spark garantisce fault-tolerance dei propri dati ricalcolando l'RDD o la singola partizione andata persa. Sfruttando la persistenza con replicazione però, a fronte di un maggior utilizzo di memoria, la fault tolerance viene garantita senza la necessità di ulteriori computazioni, risparmiando tempo e risorse di calcolo.

Per quanto riguarda la liberazione di risorse, infine, Spark monitora l'utilizzo di cache sui vari nodi ed elimina le vecchie partizioni seguendo una politica LRU (*least-recently-used*). È possibile anche richiedere la liberazione di memoria manualmente utilizzando il metodo *unpersist*.[37]

2.3.3 Proprietà

A questo punto della trattazione, dopo aver descritto i concetti principali del framework, è possibile comprendere a pieno le principali caratteristiche offerte da Spark e apprezzare le qualità che gli hanno permesso di affermarsi come strumento di processamento nel settore Big Data. Di seguito viene riportato un elenco volto a riassumerle:

- **Velocità:** il processamento in parallelo, l'utilizzo intensivo della cache e la gestione attiva del DAG, rendono Spark un framework veloce ed efficiente per il processamento di dati su larga scala. Se confrontato con Hadoop MapReduce, il framework computazionale del progetto Hadoop, Spark garantisce processamenti fino a cento volte più veloci.[33]
- **Scalabilità:** grazie all'architettura master slave con cui è realizzato, risulta semplice realizzare applicazioni Spark scalabili. A fronte di un maggior carico di lavoro, infatti, è sufficiente aumentare il numero di executor.
- **Fault tolerance:** la resilienza dei dati derivanti dal processamento viene garantita dall'utilizzo del lignaggio o dalla possibilità di salvare le partizioni in modalità replicata.
- **Versatilità:** grazie alle librerie costruite sullo Spark core, Spark si propone come framework adatto a tanti tipi di processamento diversi. Inoltre, la possibilità di interfacciamento con tantissime sorgenti dati e cluster manager differenti ne facilitano l'impiego in tantissimi casi d'uso.
- **Facilità d'uso:** la vasta famiglia di linguaggi di programmazione che sono supportati nativamente, uniti alle API di alto livello offerte, fanno sì che Spark possa essere appreso facilmente velocizzando il processo di sviluppo delle applicazioni.

2.4 Machine Learning Clustering

Gli algoritmi utilizzati nel campo del Machine Learning possono essere raggruppati in due categorie: tecniche di *apprendimento supervisionato* e tecniche di *apprendimento non supervisionato*.

Nella prima categoria, il termine "supervisionato" indica che il dataset di riferimento viene arricchito con ulteriori informazioni fornite da figure esperte del dominio oppure sfruttando dati provenienti dallo storico. Ad ogni record del dataset, quindi, viene aggiunta una o più *label* che permettono di classificarlo come appartenente ad una certa categoria.

A partire dal dataset arricchito, anche detto *training* set, gli algoritmi di apprendimento supervisionato hanno lo scopo di determinare una funzione in grado di associare ad un certo input (e.g. il vettore con gli attributi del record) un determinato output (e.g. la *label*).[39] I modelli così creati potranno quindi essere usati per inferire della nuova conoscenza su esempi e record mai visti prima. Un esempio di tecnica di apprendimento supervisionato è la classificazione.

Nelle tecniche di apprendimento non supervisionato, invece, i dataset non vengono arricchiti. Lo scopo è quello di apprendere e dedurre nuova conoscenza semplicemente osservando i dati di cui si dispone e i pattern ricorrenti.[40]

Un esempio di tecnica non supervisionata è la *cluster analysis.* Essa consiste nel suddividere i record del dataset in un certo numero di gruppi, o *cluster*, basandosi sulla loro similarità. Si noti come, seppur lo scopo della cluster analysis assomigli molto a quello della classificazione, è l'approccio che viene usato per raggiungerlo, dettato dall'appartenenza alle due diverse famiglie di apprendimento, a differenziare queste due tecniche.

La similitudine tra due record deve quindi essere un valore misurabile e può essere definita in diversi modi, anche se la metrica solitamente usata è la distanza euclidea.

Ciascun record, infatti, può essere visto come un punto dello spazio euclideo dove i suoi attributi, se raggruppati in un unico vettore, ne rappresentano le coordinate. Questo implica che ad ogni attributo numerico verrà associata una dimensione dello spazio. Se il numero di attributi che vengono considerati è elevato si rischia di incorrere nella cosiddetta "curse of dimensionality"[41] e ottenere risultati scarsi, a fronte di elevati sforzi computazionali. All'aumentare della dimensionalità, infatti, aumenta anche il volume dello spazio euclideo e i punti risultano sempre più sparsi. Questo fa sì che le varie distanze tra coppie di punti differiscano sempre meno e che i singoli attributi utilizzati perdano rilievo.

È importante quindi, prima di approcciare un algoritmo di clustering, individuare gli attributi significativi che si vogliono utilizzare. Potrebbe sembrare controintuitivo, ma conviene

scegliere attributi che sono tra loro scarsamente correlati. Scegliere attributi correlati infatti, fa sì che i singoli attributi non influiscano con il peso che effettivamente meritano.[42]
Un altro accorgimento che è bene adottare prima di eseguire l'algoritmo di clustering è normalizzare gli attributi perché abbiano tutti lo stesso range. In questo modo la distanza euclidea potrà essere calcolata in modo più corretto, poiché le variazioni che accadono in diverse dimensionalità diventano comparabili.[43]

2.4.1 Customer segmentation

La cluster analysis trova applicazione in diversi campi. Tra questi figura anche il settore del marketing dove può essere utilizzata per effettuare una *customer segmentation*. Con questo termine si intende la suddivisione del mercato in determinati gruppi di clienti, o anche detti segmenti di clienti, che condividono caratteristiche simili.
La customer segmentation può essere condotta considerando diversi tipi di attributi. Tra i più comunemente utilizzati vi sono: [44]

- informazioni di carattere demografico, come genere, età, stato di famiglia, reddito, mestiere e livello di educazione.
- informazioni geografiche, come residenza e città presso cui il cliente ha fatto acquisti o usufruito di servizi.
- informazioni psicografiche, come classe sociale, stili di vita ecc.
- informazioni relative ai comportamenti, come acquisti, abitudini di consumo e interazione con il prodotto/servizio.

Attraverso la customer segmentation è possibile comprendere maggiormente il mercato a cui si affaccia il proprio business e approfondire la conoscenza dei clienti che vi partecipano.
Una volta individuati gruppi di clienti simili è possibile intraprendere specifici piani di azione con lo scopo di migliorare il proprio business. Alcuni esempi sono il lancio di campagne pubblicitarie mirate o la scelta del prezzo più appropriato per un prodotto.

2.4.2 L'algoritmo K-Means e le sue varianti

Esistono diversi algoritmi di clustering ma l'algoritmo K-Means è sicuramente uno tra i più utlizzati. Partendo da una dataset di record casualmente distribuiti, esso è in grado di suddividerli in un numero *k* di cluster specificabile a piacimento, assegnando ogni record ad un cluster. L'algoritmo funziona nel seguente modo: [45]

Presi in input:

- N oggetti caratterizzati da vettori di dimensione d.
- Un intero k che rappresenta il numero di cluster desiderati.
- K vettori di dimensione d che rappresentano i centroidi iniziali di ciascun cluster.

L'algoritmo itera lungo i seguenti step:

- Step 1) Per ogni record calcola la sua distanza euclidea dai centroidi.
- Step 2) Assegna ogni oggetto al cluster il cui centroide ha la distanza euclidea più bassa.
- Step 3) Aggiorna il centroide di ogni cluster calcolandolo come punto medio dei record ad esso assegnati.

Gli step 1, 2 e 3 vengono ripetuti fintanto che non si verifica un criterio di convergenza. Questo potrebbe coincidere con un numero massimo di iterazioni oppure avvenire quando i nuovi centroidi non cambiano più posizione rispetto all'iterazione precedente (oppure cambiano ma solo di un valore inferiore ad una determinata soglia).

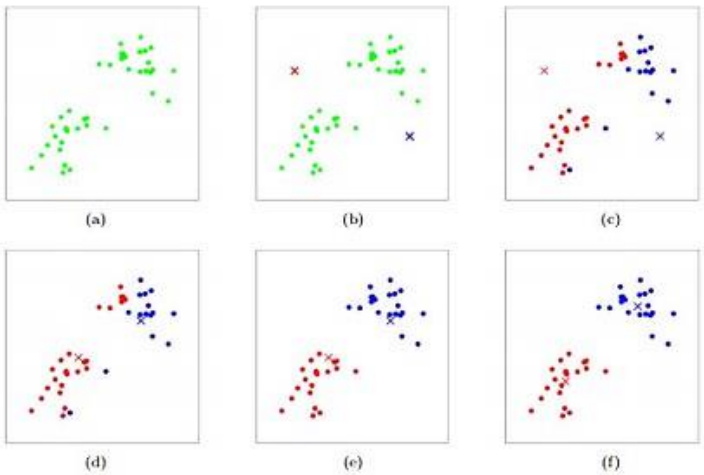


Figura 2.14 – I punti circolari rappresentano i dati mentre le x i centroidi. La sotto figura (a) illustra il dataset originale, (b) l'assegnazione iniziale e casuale dei centroidi. I passaggi (c)-(f) mostrano l'iterazione lungo gli step 1, 2 e 3 fino alla convergenza.

È importante evidenziare come l'esecuzione dell'algoritmo K-Means non sia deterministica bensì dipenda dal tipo di inizializzazione eseguita. In altre parole, dipende dal numero *k* di cluster richiesti e anche da come vengono scelti i centroidi iniziali.

L'algoritmo K-Means, inoltre, non garantisce il raggiungimento di un ottimo globale ma solo di un ottimo locale. La garanzia, cioè, è quella di raggiungere la configurazione di cluster migliore possibile[27] secondo le condizioni iniziali fornite.

In una sua variante, chiamata K-Means++, l'algoritmo viene inizializzato scegliendo i centroidi di partenza tra i punti del dataset e in modo che siano il più sparsi possibile. Questa strategia di inizializzazione, seppur più costosa rispetto a quella dell'algoritmo standard che inizializza i centroidi in modo casuale, porta a individuare dei centroidi che con grande probabilità apparterranno a cluster differenti e questo permette all'algoritmo K-Means++ di convergere alla configurazione di cluster ottima in tempi sensibilmente più brevi.[46]

Per quanto riguarda la scelta del parametro *k*, invece, esistono diversi metodi in grado di determinare il suo valore ottimale. Uno di questi è il cosiddetto *Elbow Method*.

Si tratta di una tecnica euristica che consiste nell'eseguire più volte l'algoritmo K-Means con un numero variabile di cluster e salvare per ciascuna esecuzione una metrica indicativa della qualità della configurazione di cluster ottenuta. Si procede poi a graficare i valori della metrica al variare del numero k di cluster. Si sceglie infine il *k* ottimo in corrispondenza del valore dopo il quale la metrica non migliora più in modo significativo. Quanto appena descritto è individuabile graficamente nel punto in cui la curva ricorda un gomito, da cui appunto il nome.

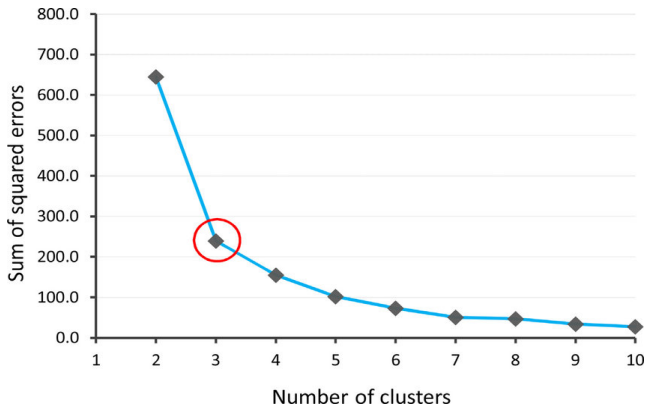


Figura 2.15 – Un esempio di Elbow Method applicato utilizzando la SSE (Sum of Squared Errors) come metrica per determinare la bontà della configurazione di cluster. In rosso viene cerchiato il numero ottimale di cluster, k=3, dopo il quale la metrica non migliora più significativamente.

[27] La qualità di una configurazione di cluster è misurabile con diverse metriche che tengono in considerazione le distanze intra-cluster e le distanze inter-cluster. Si ottiene una configurazione di qualità maggiore quanto più la distanza intra-cluster viene minimizzata e la distanza inter-cluster massimizzata. In altri termini, una configurazione di cluster è di buona qualità quando i punti appartenenti allo stesso cluster sono molto vicini tra loro e allo stesso tempo distanti dai punti appartenenti ad altri cluster.

Prima di concludere il capitolo è importante fare alcune considerazioni sulla scalabilità dell'algoritmo K-Means. Si noti come il suo costo sia direttamente proporzionale al numero di record che compongono il dataset di riferimento. Se i record sono N, infatti, ad ogni iterazione sarà necessario calcolare *k* N distanze euclidee.

Tipicamente, quando ci si trova a lavorare con dataset molto estesi è molto probabile che anche il numero di cluster presenti sia elevato. In queste situazioni, quindi, l'esecuzione dell'algoritmo sarà molto costosa sia in termini di risorse che di tempo impiegato. In altri termini, l'algoritmo K-Means nella sua versione standard è scarsamente scalabile.

Fortunatamente esiste un'altra versione dell'algoritmo, chiamata K-Means | |, che nasce per superare questo limite servendosi di una computazione parallela e distribuita. Una sua implementazione viene messa a disposizione dalla libreria MLlib di Apache Spark.

Ad ogni executor Spark verranno inviati i centroidi stabiliti in fase di inizializzazione e verrà assegnata una partizione di dati su cui lavorare. In parallelo, ciascun executor calcolerà le distanze euclidee dei soli punti ad esso assegnati e determinerà per ciascuno il cluster di appartenenza. A questo punto verrà eseguito uno shuffle per far sì che i record vengano raggruppati per cluster. Ogni executor, quindi, possiederà tutti i record di un determinato cluster e potrà calcolarne il nuovo centroide dando inizio ad una nuova iterazione.[48]

Servendosi dell'algoritmo K-Means | | è possibile quindi eseguire il K-Means clustering garantendo scalabilità e distribuendo il carico tra i vari executor.

Capitolo 3 – Implementazione di una Pipeline di Big Data Analytics

In questo capitolo viene descritta la parte progettuale della tesi, ossia l'implementazione di una pipeline di Big Data analytics che supporti e arricchisca il progetto Hera SDG.
L'obiettivo è quello di realizzare uno strumento per il monitoraggio e la gestione della community basato sui dati da essa generati. In questo modo, l'intero progetto Hera SDG potrà giovare di un management consapevole e sarà possibile attuare decisioni *data driven* in un'ottica di continua evoluzione e miglioramento del sistema.
Il capitolo è diviso in quattro paragrafi. Nel primo viene fornita una panoramica sul progetto, trattandone l'architettura e i requisiti che ne hanno guidato il design. Nel secondo vengono descritte Kubernetes e Helm, due tecnologie che ho utilizzato in fase di sviluppo. Il terzo è dedicato alla descrizione dei vari componenti della pipeline, soffermandosi sui dettagli implementativi. Nel quarto paragrafo, infine, vengono riportate alcune ottimizzazioni riguardanti il deploy della pipeline e alcune misurazioni delle performance ottenute.

3.1 Il progetto

Il progetto Hera SDG, e in particolare la community ad esso associata, possono generare grandi quantità di dati se si considera che l'attuale bacino di utenza Hera, Conad e Camst si assesta nell'ordine dei milioni di utenti e migliaia di partner store. Questi numeri, inoltre, possono potenzialmente aumentare qualora nuovi attori decidessero di unirsi al partenariato, incoraggiati dal Goal 17 dell'Agenda per lo Sviluppo Sostenibile ONU.[28]
I dati che è possibile raccogliere per ciascun utente e partner store aderente alla community sono innumerevoli. Quella che segue è una lista non esaustiva di alcuni esempi: si possono registrare dati di carattere demografico e geografico, dati relativi alla circolazione dei token, dati riguardanti i comportamenti sostenibili che vengono messi in pratica, dati circa i premi che vengono acquistati e ancora dati di interazione dell'utente con la Community App e con il market store.
Come introdotto in apertura di capitolo, l'obiettivo del progetto è sviluppare una pipeline di Big Data Analytics che acquisisca tutti questi dati e li accompagni in un processo di

[28] *cfr. p.2.*

memorizzazione, processamento, training di modelli di Machine Learning ed infine visualizzazione.

Soprattutto grazie agli ultimi due step, la pipeline giocherà un ruolo fondamentale nel monitoraggio continuo della community e sarà un importante strumento di supporto decisionale. Eventuali business manager o figure preposte all'amministrazione del progetto Hera SDG, infatti, potranno ricavare valore a partire dai dati disponibili e intraprendere azioni e iniziative per migliorarlo. Potrebbero, ad esempio, decidere di ridurre l'incentivo premiale di un certo comportamento dopo aver notato che molti utenti lo stanno perseguendo e magari aumentare gli incentivi di comportamenti meno seguiti, per stimolarli. Sarà possibile, inoltre, fissare dei *Key Performance Indicator* (KPI) per la community, ossia delle metriche con cui valutare in maniera oggettiva l'andamento del progetto Hera SDG e il raggiungimento degli obiettivi posti. Alcuni KPI significativi potrebbero essere il volume di utenti e partner store aderenti alla comunità, la quantità di materiale e rifiuti recuperati grazie alla raccolta differenziata, la quantità di CO_2 in meno emessa in atmosfera perseguendo determinati comportamenti sostenibili o ancora un indicatore di raggiungimento dei singoli SDG da parte della comunità.

È importante sottolineare che una pipeline di Big Data processing e analytics, in quanto tale, ha un grande potenziale poiché può essere utilizzata come fondamento su cui poter sviluppare diverse soluzioni e componenti.

Con l'ausilio del Machine Learning e dei dati di cui si dispone, ad esempio, si potrebbero allenare dei modelli in grado di predire quanto un nuovo comportamento potrà coinvolgere la comunità di utenti, basandosi sui dati di interazione con comportamenti simili già attivi.

Un'altra possibilità è quella di sviluppare modelli capaci di evidenziare le *association rules* tra comportamenti sostenuti e premi scelti, con l'obiettivo di individuare relazioni e pattern che altrimenti sarebbero difficilmente individuabili.

Nel mio progetto ho deciso di implementare un modello di clustering che permette di segmentare gli utenti della community basandosi sulle loro informazioni demografiche (sesso, età), geografiche (provincia di residenza), sulla quantità di token guadagnati e spesi e sull'engagement con i vari partner.

Tuttavia, è bene precisare che questo componente analitico è proposto e realizzato come mero strumento di supporto decisionale. Seppur in un miglioramento futuro sarebbe possibile

automatizzare il processo decisionale sviluppando un componente di analisi prescrittiva[29], il componente di analytics che ho realizzato si limita a compiere un'analisi descrittiva. Qualsiasi tipo di interpretazione dei dati e decisioni, quindi, vengono rimandate ad un business manager o a figure preposte alla gestione della community.

A questo punto della trattazione, dopo aver discorso ad alto livello del progetto, è possibile descrivere l'architettura della pipeline e fornire una panoramica dei componenti limitandomi, per il momento, a citare le tecnologie utilizzate per realizzarli. Alla loro specifica implementazione è dedicato l'intero paragrafo 3.3.

3.1.1 Architettura

I dati che vengono originati dalle sorgenti sono grezzi, anche detti *raw data*, e per poterne ricavare valore è necessario sottoporli ad una serie di operazioni sequenziali che ne migliorino la qualità e ne permettano la comprensione, l'analisi e la visualizzazione.
Tipicamente, i passaggi principali che si trovano in una qualsiasi data pipeline sono i seguenti:

- **Data Ingestion:** i dati vengono raccolti dalle varie sorgenti e convogliati all'ingresso della pipeline.
- **Pre-processing:** i dati subiscono un primo processamento che solitamente avviene in modalità streaming agendo dato per dato. In questa fase i dati vengono "puliti" filtrando (o gestendo):
    - record malformati.
    - record con valori mancanti.
    - record senza schema.
    - record outlier.

  In questa fase è possibile anche arricchire i dati con informazioni aggiuntive ("Data Enrichment") o prepararli per operazioni di analisi da compiere in futuro. Certi modelli di ML, ad esempio, performano meglio se vengono allenati su dati standardizzati e normalizzati.
- **Storage:** i dati vengono memorizzati in opportuni sistemi di storage.

[29] Nel campo della Data Analysis si distinguono tre tipi di analisi: *descrittiva*, *predittiva* e *prescrittiva*. La prima è tipica di quei modelli che si limitano a descrivere una situazione attuale rispondendo alle domande "cosa è successo?" e "perché è successo?". L'analisi predittiva invece si concentra sul predire l'evolversi futuro di una certa situazione rispondendo alla domanda "cosa succederà?". Infine, l'analisi prescrittiva è delle tre quella che si spinge più lontano in quanto suggerisce (ed eventualmente mette in pratica) le decisioni e le azioni da dover intraprendere sulla base delle analisi compiute.[51]

- **Processing & ML Analytics:** in questa fase è possibile condurre un processamento dei dati più articolato rispetto alle operazioni di pre-processing, compiendo operazioni come aggregazione o join relazionali. Inoltre, è anche possibile sfruttare tecniche di Machine Learning per compiere analisi descrittive, predittive e/o prescrittive sui dati.
- **Visualization:** i dati e gli insight prodotti dallo step precedente vengono visualizzati realizzando delle dashboard o dei report grafici.

Ispirata dalle fasi appena descritte, la pipeline di Big Data Analytics che ho realizzato presenta la seguente architettura:

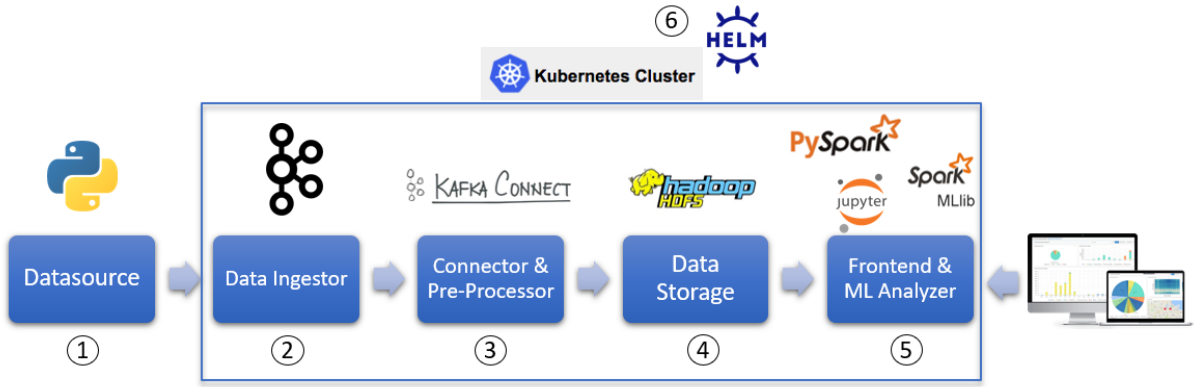


Figura 3.1 – Architettura della pipeline di Big Data Analytics realizzata per il progetto Hera SDG.
Si notino le tecnologie scelte per la realizzazione di ciascun componente: (1) Python Kafka Producer (2) Kafka (3) Kafka HDFS Sink Connector (4) Hadoop HDFS (5) Jupyter Lab & Spark.

1. Datasource: all'interno del progetto Hera SDG i data source possibili sono molti ed eterogenei. Tra questi figurano i sistemi informativi che registrano i nuovi utenti, i dispositivi IoT che tracciano i comportamenti virtuosi (e.g. registratori di cassa Conad/Camst, applicazione MyHera) e i market store dove vengono acquistati i premi (Community App e siti dei partner).

   Nel mio progetto ho simulato la presenza di un datasource realizzando un Python Kafka producer che pubblicasse i dati presso il componente "Data Ingestor", previa serializzazione in formato JSON. In particolare, pubblica tre tipi di eventi che simulano rispettivamente:

   - la registrazione di un nuovo utente.
   - il compimento di un comportamento virtuoso.
   - l'acquisto di un premio.

   Si noti come, a differenza degli altri componenti, il datasource non faccia parte del cluster gestito da Kubernetes, in quanto logicamente non appartiene alla pipeline.

2. Data Ingestor: i dati prodotti dal datasource vengono pubblicati nel componente di Data Ingestion che è un cluster Kafka. Gli eventi vengono suddivisi in tre topic (utenti, comportamenti, premi) opportunamente partizionate per garantire parallelismo e vengono resi disponibili ai consumatori interessati.

3. Connector & Pre-Processor: questo componente permette l'interfacciamento tra Kafka e il sistema di storage HDFS ed è realizzato come un cluster di Kafka HDFS Sink Connector eseguiti in modalità distribuita. Una volta pubblicati sul componente "Data Ingestor", infatti, i record vengono consumati dai vari worker e poi, ad intervalli temporali regolari, scritti su HDFS sotto forma di file.
   I connector eseguono anche un pre-processamento a livello di singolo messaggio filtrando i record con valore nullo. Inoltre, sono in grado di gestire eventuali messaggi pubblicati che non rispettino le specifiche sfruttando la tecnica della *Dead Letter Queue.* Per maggiori informazioni si rimanda al paragrafo 3.3.4.

4. Data Storage: il componente di Data Storage è un cluster HDFS. I record pre-processati correttamente vengono qui salvati in tre directory distinte (una per ogni topic). Il componente "Data Storage" si ritrova quindi a possedere lo storico di tutti i dati della community.

5. Frontend & ML Analyzer: questo componente è stato realizzato come Jupyter Notebook e svolge una duplice funzione.
   La prima consiste nel leggere i dati da HDFS, prepararli e condurre un'analisi di Machine Learning Clustering. Vengono quindi:
   - rimossi eventuali record duplicati
   - eseguite funzioni di aggregazione quali somme dei token spesi e guadagnati.
   - eseguiti join relazionali tra le informazioni utente, comportamenti e premi in base alla chiave *id_utente.*
   - calcolati alcuni indicatori di engagement con i vari partner.

   Successivamente i dati vengono normalizzati e viene eseguita la segmentazione degli utenti mediante un clustering condotto con l'algoritmo K-Means | |.
   Sia il processamento che la ML Analytics vengono eseguiti sfruttando Spark come framework di computazione. In particolare, sono state usate le API offerte da PySpark e dalla libreria MLlib.
   La seconda funzione svolta dal componente "Frontend & ML Analyzer" è quella di offrire una visualizzazione dei dati e degli insight prodotti dall'analisi di clustering. Per fare ciò

sono state utilizzate alcune librerie Python di visualizzazione come la libreria pandas_profiling e la libreria Bokeh.

6. Cluster Manager: il deploy dei componenti 2, 3, 4, 5 della pipeline è stato fatto raggruppandoli all'interno di un unico cluster gestito con Kubernetes. Tali componenti, quindi, sono stati trattati come risorse Kubernetes e distribuiti sul cluster sfruttando il tool Helm. Alla descrizione di queste due tecnologie è dedicato il paragrafo 3.2.

3.1.2 Requisiti alla base delle scelte tecnologiche

Dopo aver citato le tecnologie impiegate per realizzare ciascun componente della pipeline, questo sotto paragrafo vuole raccogliere la lista dei requisiti fondamentali che ho individuato in avvio di progetto e che mi hanno guidato nella fase di design architetturale motivando la scelta di una particolare tecnologia piuttosto che un'altra.
Similmente a quanto suggerito dall'ingegneria del software, la mia progettazione è partita da un'analisi dei requisiti, da «cosa» la mia pipeline di Big Data Analytics dovesse fare, essere e garantire, e solo in seguito ha spostato il focus sul «come» realizzarlo.
I requisiti fondamentali che ho individuato sono i seguenti, qui riportati in ordine alfabetico:

1. **Adattabilità**
2. **Batch processing**
3. **Estensibilità**
4. **Open Source**
5. **Resilienza**
6. **Scalabilità**
7. **Velocità**
8. **Visualizzazione user-friendly**

1. *La caratteristica di **adattabilità** è atta a rendere la mia pipeline robusta ai possibili cambiamenti nei requisiti che possono (sicuramente) verificarsi in futuro.*
   Una pipeline predisposta al cambiamento può essere adattata con minore sforzo in termini di tempo e costi. In questa prospettiva, la scelta di Apache Kafka come tecnologia per la Data Ingestion garantisce alla mia pipeline estrema adattabilità sia alla sorgente che alla foce del flusso dei dati. Grazie al disaccoppiamento produttore-consumatore e al vasto ecosistema di Connector esistenti, infatti, è possibile cambiare, rimuovere o aggiungere nuove sorgenti e destinazioni di dati con il minimo sforzo.

Ad esempio, un domani si potrebbe avere la necessità di utilizzare un sistema di storage alternativo a HDFS. Per adattarsi a questo cambiamento, dal momento che i record depositati in una topic Kafka possono essere consumati da più consumatori, è sufficiente aggiungere al sistema il Kafka Sink Connector specifico per il nuovo storage. Come illustrato in Figura 3.2, il flusso di dati potrà dare origine ad una nuova pipeline lasciando immutata quella esistente.[53]

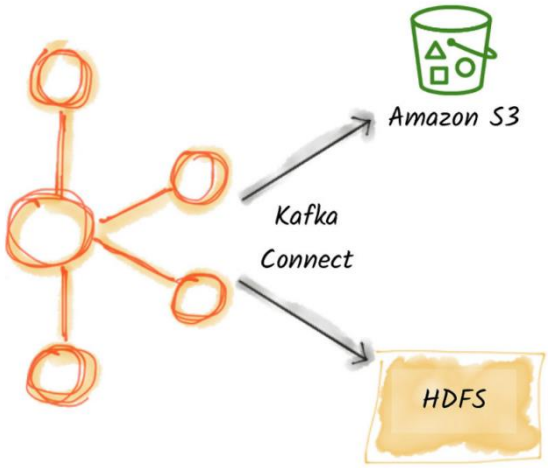


Figura 3.2 – L'utilizzo di Kafka permette di biforcare la pipeline all'altezza del componente di Data Ingestion, offrendo adattabilità al sistema.

Potrebbe altresì cambiare il formato con cui vengono prodotti i record. Anche questo non rappresenterebbe un problema in quanto Kafka è in grado di interfacciarsi con tutti i formati di dati più diffusi (JSON, Avro, CSV, ecc.).

Il requisito di adattabilità, infine, viene soddisfatto anche dalla presenza di Spark, framework di computazione "unificato". Una pipeline realizzata con Spark, infatti, è già predisposta a diversi tipi di processamento (e.g. graph processing o stream processing).

2. *Le esigenze del progetto Hera SDG portano alla scelta di un* ***batch-processing***[30] *rispetto ad uno stream processing.*

[30] Batch vs stream processing: nello stream processing si lavora con dati in tempo reale, il cui flusso è continuo e il processamento viene applicato immediatamente dato per dato, in modo da ricavare informazioni di cui si ha necessità in tempi brevi. Nel batch processing, diversamente, si accumulano batch di dati per un certo intervallo di tempo e solo in seguito si procede a processarli. Inoltre, mentre lo stream processing può eseguire solo operazioni semplici sui flussi di dati, a causa della velocità di generazione degli stessi, la seconda può effettuare analisi più approfondite e complesse, avendo più tempo a sua disposizione per la produzione e presentazione degli insights.[52]

Non essendovi decisioni da prendere o azioni da compiere nell'immediato che dipendano dall'analisi real-time dei dati raccolti, infatti, è possibile far confluire i dati nel componente di Data Ingestion, leggerli in batch, memorizzarli e analizzarli su richiesta.

3. *I cambiamenti nei requisiti potrebbero portare non solo a modificare ma anche ad **estendere** la pipeline con nuovi componenti.*

   L'architettura orientata ai microservizi e l'utilizzo di K8s, che a differenza di altri cluster manager (e.g. Yarn) containerizza i componenti del sistema, garantisce un elevato isolamento, agevolando la manutenibilità e l'estensibilità della pipeline. Banalmente, dal momento che ciascun componente containerizzato dispone internamente delle proprie dipendenze, vengono evitati problemi quali il *dipendency hell*.

   L'estensibilità è richiesta anche a livello del componente di "Frontend & ML Analyzer", dove è desiderabile poter aggiungere nuovi modelli di ML.

4. *La pipeline dovrà essere realizzata prediligendo tecnologie **open-source.***

   Le tecnologie open-source sono utilizzate in maniera diffusa. Di conseguenza risulta più semplice ottenere supporto da parte della comunità degli sviluppatori mediante forum o tutorial presenti in rete. Utilizzare una tecnologia open source, inoltre, permette di avere accesso al codice sorgente. Kafka, HDFS, Spark e Kubernetes sono tecnologie open source.

5. *I dati sono risorse preziose perché a partire da essi è possibile ricavare valore. Non è accettabile costruire un sistema in cui ci sia la possibilità che questi vadano persi. È fondamentale garantire **resilienza**.*

   La scelta di Kafka, HDFS e Spark permette di ottenerla in ogni step della pipeline grazie alla replicazione delle partizioni Kafka, alla replicazione dei file HDFS e ai RDD Spark descritti nel capitolo 2.

6. *La pipeline deve essere in grado processare e analizzare grandi quantità di dati e farsi trovare pronta nel caso in cui il progetto Hera SDG dovesse espandersi e aumentare il carico di lavoro richiesto.*

   La natura distribuita di Kafka, HDFS e Spark predispone alla realizzazione di un processamento parallelo e distribuito, e di conseguenza **scalabile.** Il resto del gioco è compiuto da Kubernetes che, in qualità di orchestratore di container, è in grado di mettere in pratica un auto-scaling basato sul carico attuale delle singole entità.

   L'auto-scaling prevede anche uno scale-down in caso di bassi carichi e questo è fondamentale per garantire una gestione efficiente delle risorse del cluster, soprattutto nel caso del componente "Frontend & ML Analyzer". Dal momento che Spark utilizza

intensivamente le risorse (e.g. RAM) e che l'analisi dei dati viene eseguita solo su richiesta, infatti, è fondamentale fare scale-down degli executor Spark quando non sono necessari.

7. *Per rendere più proficua la fase di supporto decisionale rivolta ai business manager è necessario un componente frontend che sia **user friendly** ma allo stesso tempo estensibile e programmabile per rifarsi al requisito 4.*

   Jupyter Lab è uno strumento perfetto per realizzare un frontend con questi requisiti in quanto abbina l'opportunità di utilizzare le ricche librerie di visualizzazione Python con la possibilità di visualizzare il codice sorgente ed eventualmente modificarlo sulla base delle proprie esigenze.

8. *La pipeline deve garantire **velocità** di esecuzione in ogni suo step.*

   Questo requisito viene garantito grazie all'alto throughput in lettura e scrittura offerto da Kafka e HDFS e all'utilizzo intensivo della RAM nel processamento con Spark.

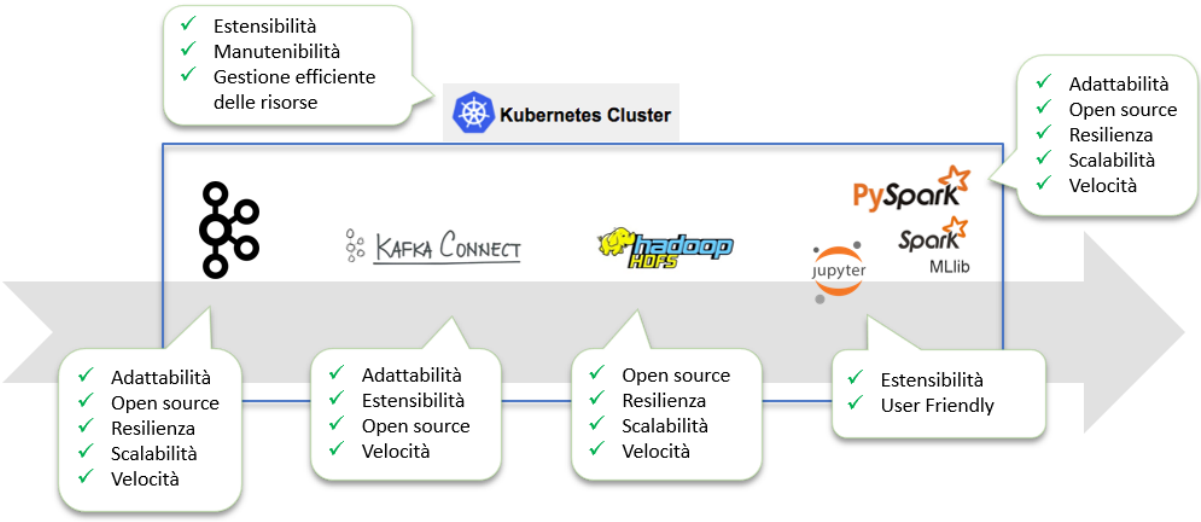


Figura 3.3 – I requisiti che mi hanno guidato nella scelta delle tecnologie per realizzare la pipeline.

3.2 Tool implementativi

In questo paragrafo vengono descritti i due strumenti che hanno agevolato il processo di sviluppo del mio progetto. Il primo è Kubernetes, che ho utilizzato come cluster manager mentre il secondo è Helm, un package manager tramite il quale è possibile recuperare e distribuire software su un cluster Kubernetes.

3.2.1 Kubernetes come Cluster Manager

L'utilizzo di Kubernetes come cluster manager garantisce alle applicazioni una serie di benefici, molti dei quali sono stati enunciati nel sotto paragrafo "Deploy dei microservizi e breve panoramica su Docker e Kubernetes" in 1.2.2.
In un cluster Kubernetes è possibile creare e interagire con diverse risorse, anche detti oggetti Kubernetes. Ecco le principali:

Pod

Il Pod è sicuramente la risorsa più importante in quanto tutte le applicazioni Kubernetes vengono trattate sotto forma di Pod. Nella documentazione ufficiale il Pod viene definito come «un insieme di container in esecuzione su cluster che condividono uno storage, risorse di rete (e.g. un indirizzo IP) e presentano delle specifiche su come devono essere eseguiti».[54] E ancora un Pod è «la più piccola unità di deploy applicativa che è possibile creare e gestire in Kubernetes».[54]
I Pod, quindi, rappresentano dei processi in esecuzione sul cluster e, a seconda dell'immagine che viene utilizzata dai suoi container, possono realizzare una qualunque applicazione software. Nonostante la definizione lo proponga come «insieme di container», tipicamente un Pod contiene un solo container, in modo da garantire massimo isolamento. Agisce quindi da wrapper incapsulando il container in un'astrazione gestibile da Kubernetes.
Il modello di Pod multi-container, invece, è meno diffuso e viene adottato solo nel caso in cui si abbiano più container, logicamente appartenenti alla stessa applicazione, con la necessità di una stretta interazione.
È importante sottolineare che i Pod hanno un ciclo di vita limitato. Quando viene fatto lo scale-down o il rollback di un'applicazione, oppure in caso di guasto, un Pod può cessare di esistere e al Pod eventualmente creato in sostituzione verrà assegnato un nuovo indirizzo IP.
L'instabilità di indirizzo genera ovvi problemi di comunicazione tra i Pod ed è per questo motivo che sono nati i servizi.

Servizi

I servizi sono un'astrazione Kubernetes con la quale è possibile «esporre un'applicazione che esegue in uno o più Pod».[55] Kubernetes, infatti, fornisce ad ogni servizio un indirizzo di rete stabile che può essere utilizzato per accedere ai Pod da esso esposti.

I servizi possono specificare quali Pod esporre usando degli opportuni *selector*. Ad ogni Pod o insieme di Pod, infatti, lo sviluppatore può associare delle label del tipo [key]:[value] a cui il servizio potrà riferirsi.

Supponiamo, ad esempio, di voler esporre mediante un servizio tutti i Pod che compongono il frontend di un'applicazione. È sufficiente assegnare a ciascun Pod una stessa label, (e.g. *label* "app:frontend") e successivamente specificare come selector del servizio lo stesso valore (*selector* "app:frontend").

I servizi, a loro volta, possono essere esposti in diversi modi ai clienti[31] che vi desiderano interagire. Il metodo di esposizione è detto *service type* e i più frequenti sono:[55]

- **Cluster IP**: se non specificato diversamente, è il tipo di default. Come si può intuire dal nome, ad un servizio di tipo Cluster IP viene assegnato un indirizzo IP accessibile solo dall'interno del cluster. Se un cluster IP espone un gruppo di Pod, inoltra le richieste facendo load balancing tra i vari Pod con una politica Round Robin.

- **NodePort:** i servizi di tipo NodePort vengono esposti usando una porta definita staticamente (*nodeport*) che viene aperta su ogni nodo del cluster. Questo tipo di servizio, quindi, risulta accessibile anche dall'esterno del cluster contattando l'indirizzo *nodeIP:nodeport*.

  Dietro alle quinte Kubernetes crea un servizio Cluster IP a cui inoltrare il traffico in arrivo sulle porte esposte, quindi il servizio NodePort può anche essere visto come un'estensione del servizio Cluster IP.

- **LoadBalancer:** anche i servizi di tipo LoadBalancer sono accessibili dall'esterno del cluster ma per funzionare necessitano di un load balancer che intercetti tutte le richieste dei client. Tutti i maggiori cloud provider (AWS, Google Cloud Platform, Azure, ecc.) forniscono una propria implementazione di load balancer che può essere usata per questo scopo.

[31] I clienti di un servizio possono essere applicazioni esterne al cluster o anche Pod interni al cluster.

Dietro alle quinte Kubernetes crea un servizio NodePort a cui inoltrare il traffico in arrivo sul load balancer, quindi il servizio LoadBalancer può essere anche visto come un'estensione del servizio NodePort.

Si noti come un servizio di tipo LoadBalancer sia più efficiente e sicuro rispetto ad un servizio di tipo NodePort. Infatti, a differenza di quest'ultimo dove il cliente specificava il nodo da contattare, ora le richieste insistono sul load balancer prima di essere inoltrate ad uno dei nodi. Questo permette di bilanciarle tra i vari nodi del cluster ed evita di esporre direttamente le porte dei nodi a connessioni provenienti dall'esterno.

- **Headless service:** un headless service viene utilizzato quando non si vuole bilanciare le richieste tra i vari Pod ma si desidera comunicare con un Pod specifico.

Può essere utile, a questo punto della trattazione, soffermarsi su alcuni esempi grafici in modo da visualizzare i concetti fin qui esposti.

Nelle immagini che seguono viene raffigurato un cluster Kubernetes composto da due nodi in cui un Pod di nome *pod-python* viene contattato nelle seguenti situazioni: [64]

- assenza di servizi (Figura 3.4).
- presenza un servizio di tipo Cluster IP (Figura 3.5).
- presenza di un servizio di tipo NodePort (Figura 3.6).
- presenza di un servizio di tipo LoadBalancer (Figura 3.7).

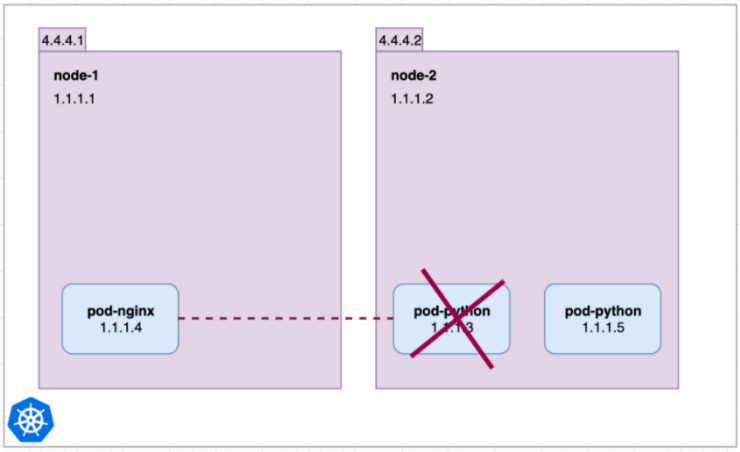


*Figura 3.4 – **Assenza di servizi.** Il crash del pod-python fa sì che al suo riavvio presenti un nuovo indirizzo IP. pod-nginx, quindi, non è più in grado di contattare pod-python.*

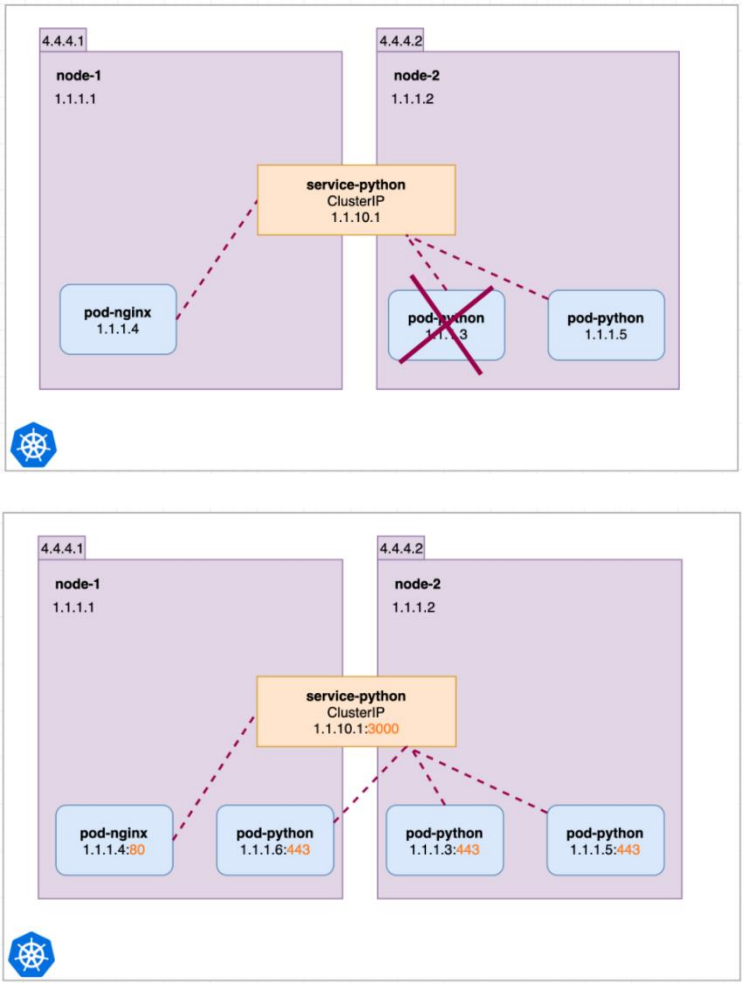


*Figura 3.5 – **Presenza di un servizio ClusterIP** che espone pod-python all'interno del cluster.*

(a) Il crash del pod-python fa sì che al suo riavvio presenti un nuovo indirizzo IP. Tuttavia, pod-nginx è ancora in grado di contattare pod-python poiché utilizza l'indirizzo IP del servizio ClusterIP, che è un indirizzo stabile.

(b) Estensione dell'esempio (a) ma questa volta si hanno tre repliche del pod-python. Il servizio Cluster IP è associato espone tutte e tre facendo load balancing.

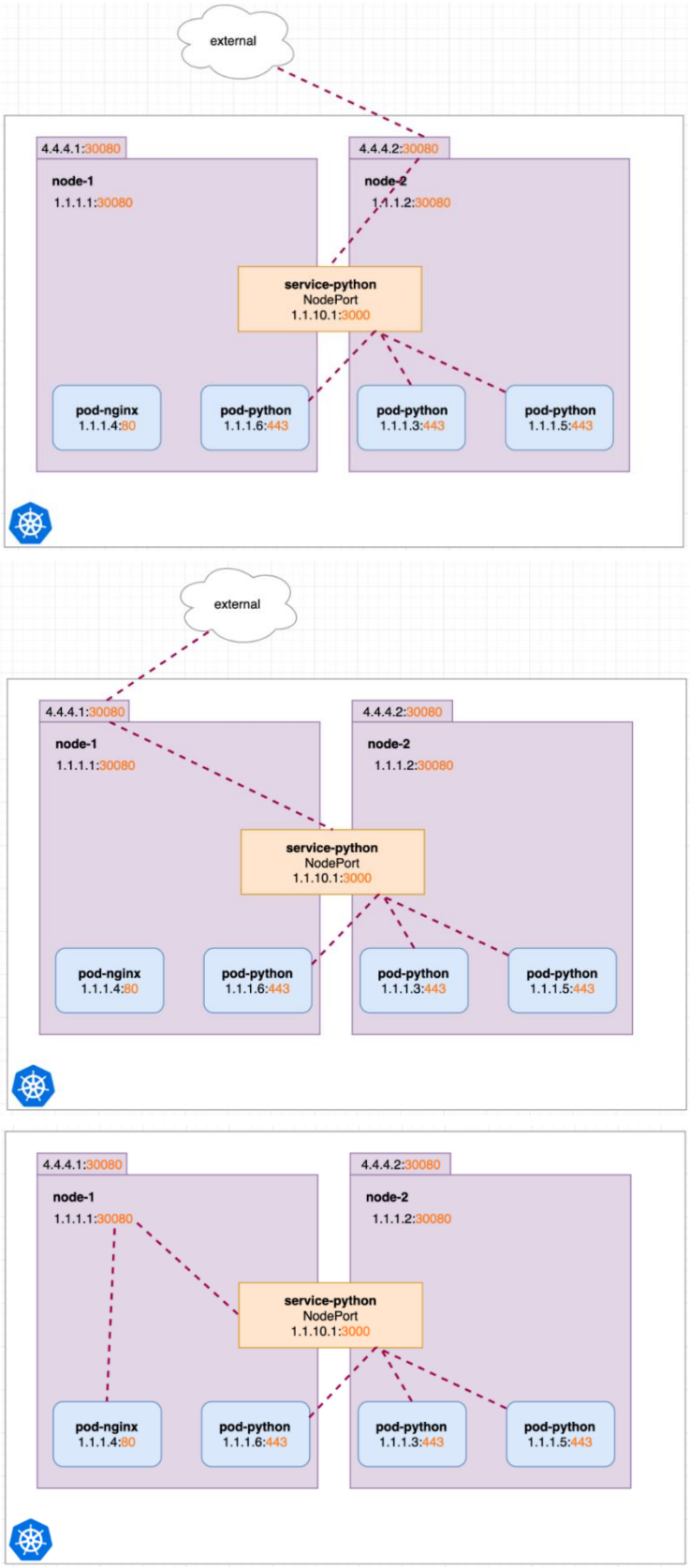


*Figura 3.6 – **Presenza di un servizio NodePort** che espone le repliche di pod-python anche all'esterno del cluster. Queste sono raggiungibili:*

(a) Contattando dall'esterno il nodo 1 all'indirizzo <external-IP-node-1>:<node-port>. Nell'esempio 4.4.4.1:30080.

(b) Contattando dall'esterno il nodo 2 all'indirizzo <external-IP-node-2>:<node-port>. Nell'esempio 4.4.4.2:30080.

(c) Contattando dall'interno un qualsiasi nodo all'indirizzo <internal-IP-node>:<node-port>.

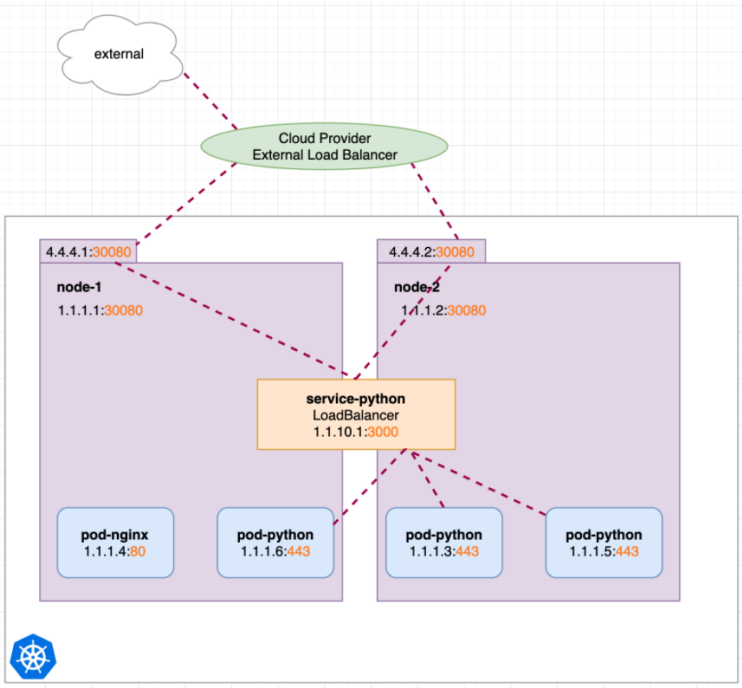


*Figura 3.7 – **Presenza di un servizio LoadBalancer** che espone le repliche di pod-python anche all'esterno del cluster mediante un load balancer del cloud provider. Questi provvede a bilanciare le richieste tra i vari nodi del cluster.*

Pod Controller: Deployment e StatefulSet

Tipicamente un Pod non viene mai creato e gestito direttamente dallo sviluppatore. In Kubernetes è buona prassi definire delle astrazioni di più alto livello, chiamate Pod Controller, che si frappongono tra sviluppatore e Pod e con le quali è possibile interfacciarsi per una gestione semplificata di gruppi di Pod analoghi (o anche di un singolo Pod).

Mediante un file in formato YAML, lo sviluppatore fornisce ai Pod Controller un *pod template* nel quale vengono esplicitate le caratteristiche che il Pod deve avere (e.g. immagine da utilizzare, labels da applicarvi, indicazioni per il deploy, ecc.) e il numero di repliche desiderate per quel Pod.

Una volta create le repliche, i Pod Controller permettono di gestirne lo scaling, il rollout ed il ripristino in caso di fallimento.[54] Qualora un nodo dovesse subire un guasto, ad esempio, Kubernetes verificherà se su quel nodo vi erano dei Pod appartenenti ad un Pod Controller, e nel caso richiederà un nuovo scheduling degli stessi su un altro nodo del cluster.[54]

Questo comportamento è reso possibile grazie al lavoro del Control Plane, che verrà meglio descritto nella sezione "Architettura", presente nel seguito della trattazione.

I principali tipi di Pod Controller sono:

- **Deployment:** un Deployment viene utilizzato per gestire un insieme di Pod che realizzano un'applicazione *stateless*. Tutti i Pod appartenenti ad un Deployment, infatti, sono intercambiabili e possono essere utilizzati indistintamente per servire le richieste di un cliente. Un esempio di applicazione stateless facilmente distribuibile come Deployment è un web server (e.g. Tomcat).
- **StatefulSet:** uno StatefulSet è adatto alla gestione di Pod che realizzano un'applicazione *stateful*, in cui cioè si ha la necessità di trattare dati persistenti (e.g. lo stato di interazione con il cliente o altri dati applicativi).

  Diversamente dal Deployment, i Pod di uno StatefulSet non sono intercambiabili nel servire le richieste. Ogni Pod, infatti, gestisce una propria unità di storage persistente che lo differenzia dagli altri Pod del gruppo. Di conseguenza i clienti di uno StatefulSet possono avere la necessità di contattare una specifica replica del Pod e, per permetterlo, ogni Pod disporrà di un proprio headless-service.

  Per rendere persistente l'associazione Pod-storage nonostante la natura effimera del primo, Kubernetes assegna a ciascun Pod dello StatefulSet un hostname stabile che viene mantenuto anche a seguito di guasti e successivi rescheduling.[57]

  L'hostname stabile fa sì che il Pod riavviato possa riassociarsi con la stessa unità di storage gestita dal suo predecessore. Si dice anche che il Pod abbia una *sticky identity*, in quanto non perde mai la propria "identità", ossia i dati persistenti da lui gestiti.

  Tuttavia, è importante sottolineare che uno StatefulSet si limita ad offrire quanto appena descritto e non garantisce altri servizi di persistenza (e.g. replicazione dei dati, consistenza tra le repliche, ecc.). Se questi aspetti fossero richiesti, sarebbero a carico dell'applicazione ospitata nei Pod.

  Alcuni esempi di applicazioni realizzabili come StatefulSet sono Kafka, HDFS e altri sistemi di storage (MySQL, MongoDB, ecc.).

Storage: Volume e Persistent Volume

I file e i dati contenuti all'interno di un container sono effimeri e vengono cancellati quando il container termina. Inoltre, container appartenenti allo stesso Pod potrebbero avere la necessità di condividere gli stessi dati. Per ovviare entrambi i problemi esistono i **Volumi**, unità di storage che possono essere definite e rese accessibili dai container appartenenti allo stesso Pod.[58]

I volumi hanno un ciclo di vita pari a quello del Pod che li ospita: resistono quindi al crash e al riavvio di un singolo container ma quando il Pod viene rimosso, per un qualsiasi motivo, anche il volume viene eliminato con tutto il suo contenuto.

Per garantire una persistenza più durevole si utilizzano i **Persistent Volume** (PV). Similmente ai volumi descritti in precedenza sono unità di storage ma hanno un ciclo di vita pari a quello del cluster e quindi indipendente dal Pod che li utilizza.[59] Possono essere creati staticamente dall'amministratore del cluster oppure forniti dinamicamente appoggiandosi a servizi di cloud storage.

Un Pod che desidera usare un PV sfrutta un'astrazione Kubernetes chiamata **Persistent Volume Claim** (PVC). Essa rappresenta una richiesta di storage da parte di un cliente ed esula il Pod dal designare il PV specifico che vuole utilizzare.

Attraverso una PVC, infatti, il Pod si limita a specificare le caratteristiche di storage desiderate, come la quantità di memoria e le modalità di accesso[32]. Successivamente, è compito della PVC ricercare nel cluster eventuali PV che soddisfino le richieste e che non siano già associati ad altre PVC.

È bene sottolineare come sia i Deployment che gli StatefulSet possano utilizzare i PV. Nel caso di un Deployment, però, tutti i Pod condivideranno lo stesso PVC mentre nel caso di uno StatefulSet ogni Pod avrà il proprio PVC e quindi la propria unità di storage.[60]

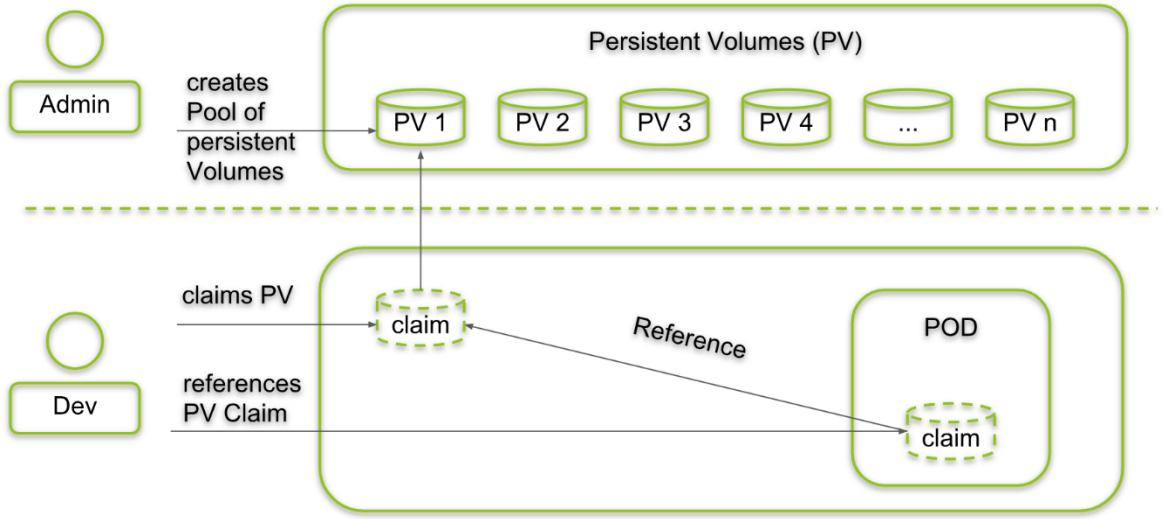


Figura 3.8 – Lo sviluppatore inserisce nel Pod un riferimento ad una Persistent Volume Claim, esplicitando le caratteristiche di storage desiderate (e.g. memoria di tipo file-system con capacità di almeno 10 Gb). A questo punto la PVC ricerca un Persistent Volume adeguato tra quelli presenti nel cluster e vi si associa.

[32] I Persistent Volume supportano le seguenti modalità di accesso:[59]
- ReadWriteOnce: il volume può essere acceduto in lettura e scrittura da un singolo nodo.
- ReadOnlyMany: il volume può essere acceduto in sola lettura da nodi multipli.
- ReadWriteMany: il volume può essere acceduto in sola lettura e scrittura da nodi multipli.

Architettura

Per poter utilizzare Kubernetes è necessario predisporre un cluster adeguato. **Un cluster Kubernetes** è composto da uno o più nodi worker, su cui eseguono i Pod delle applicazioni, e da un Control Plane, che monitora i nodi worker e gestisce i Pod del sistema.

Più precisamente, il Control Plane si occupa di prendere decisioni riguardanti il cluster, come scegliere in quale nodo far eseguire un nuovo Pod o richiedere la creazione di un nuovo Pod quando il numero di repliche gestite da un Pod Controller è inferiore a quelle richieste.[61]

In Figura 3.9 viene illustrata l'architettura di un cluster Kubernetes che mi accingo a descrivere.

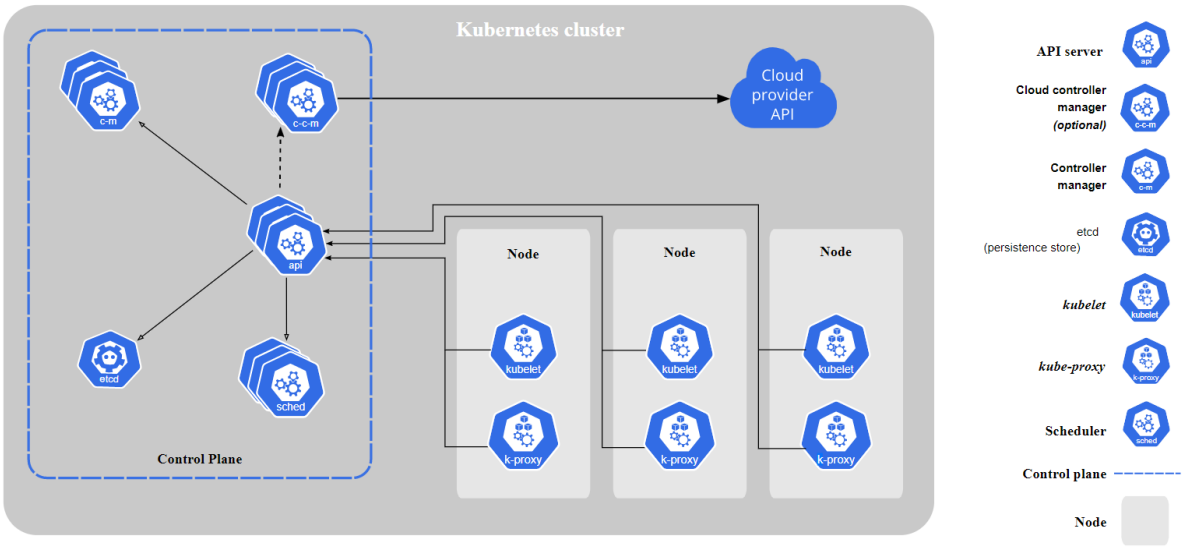


Figura 3.9 – Architettura di un cluster Kubernetes.

Il Control Plane è composto da quattro processi software che possono eseguire tutti nello stesso nodo oppure essere replicati e distribuiti su più nodi del cluster, in modo da garantire tolleranza ai guasti ed elevata disponibilità. La seconda opzione è chiaramente preferita in un contesto di produzione. I componenti del Control Plane sono i seguenti:[61][62]

- **API server:** è il frontend che espone tutte le API di K8s con cui il developer, le risorse e i processi Kubernetes possono interfacciarsi.
  Tutte le richieste passano attraverso l'API server che funge da gateway. Ogni richiesta viene validata autenticando il richiedente e verificando che abbia i privilegi necessari. In caso affermativo, la richiesta viene inoltrata al processo competente.
- **etcd:** è uno storage distribuito e resiliente che memorizza tutte le informazioni relative ai nodi del cluster sotto forma di dati key-value.
  Alcuni esempi di informazioni memorizzate nell'etcd sono le risorse disponibili per ogni nodo, lo stato di salute dei nodi, lo stato di salute dei singoli Pod, ecc. I dati applicativi,

invece, non vengono memorizzati nell'etcd ma nei Volumi o nei Persistent Volume descritti in precedenza.

- **Scheduler:** è il processo responsabile per lo scheduling dei Pod tra i vari nodi. Monitora continuamente il cluster per rilevare la presenza di nuovi Pod ancora non assegnati ad alcun nodo e per ciascuno sceglie il nodo su cui farli eseguire.

  Nella scelta del nodo influiscono diversi fattori, tra cui le risorse disponibili, le risorse richieste dal Pod e le specifiche di affinità e anti-affinità fornite dallo sviluppatore nel file YAML della risorsa.[61]
- **Controller:** è il processo che si assicura che lo stato attuale del cluster coincida in ogni istante con lo stato desiderato. Kubernetes, infatti, lavora seguendo un paradigma dichiarativo: il programmatore fornisce delle specifiche che descrivono lo *stato desiderato* delle risorse ed il controller monitora costantemente lo *stato attuale* assicurandosi che combaci con quanto richiesto.

  È compito del controller, ad esempio, monitorare i guasti dei nodi e dei Pod, per assicurarsi che il numero di repliche richieste per un certo Pod Controller sia sempre consistente con il numero di Pod in esecuzione. Nel caso ci fosse una divergenza, il controller richiederà la creazione di nuovi Pod.

Continuando la descrizione architetturale, su ogni nodo worker devono essere installati tre processi:[61][62]

- **Container-runtime:** è il supporto software che permette l'esecuzione dei container. Kubernetes supporta diversi container-runtime ma il più diffuso è sicuramente Docker.
- **Kubelet:** è un agente che gestisce il nodo su cui esegue per conto del Control Plane. Periodicamente, infatti, gli invia dei report con lo stato del nodo e dei relativi Pod e viene istruito sulle azioni da eseguire, quali creazione/eliminazione di Pod.

  Si noti come la richiesta di creazione di un Pod passi sempre attraverso l'API server. Da qui viene demandata allo Scheduler e successivamente al Kubelet presente sul nodo designato.
- **Kube-proxy:** è il processo che gestisce le regole di rete che permettono la comunicazione con i Pod allocati in quel nodo da parte di connessioni provenienti dall'interno o dall'esterno del cluster.

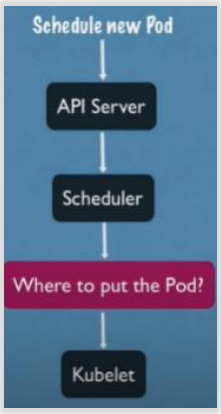


Figura 3.10 – Il flusso della richiesta di creazione di un nuovo Pod.

Creazione di una risorsa

La creazione di una risorsa Kubernetes avviene in due passaggi. Inizialmente è necessario scrivere un file in formato YAML in cui descriverla. Un esempio di file descrittore di un Deployment è visionabile in Figura 3.11.

application/deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Figura 3.11 – Un esempio di file YAML in cui viene descritto un Deployment Kubernetes.

È possibile notare come nella descrizione della risorsa si possano includere diverse informazioni, organizzate per valori annidati. Alcune di queste sono obbligatorie:[63]

- **apiVersion:** la versione delle API Kubernetes che si stanno usando per richiedere la creazione dell'oggetto.
- **kind:** il tipo di risorsa da creare.
- **metadata:** sono dati utili per identificare l'oggetto. Tra questi troviamo il nome ed eventualmente delle labels.
- **spec:** è una delle sezioni più importanti in quanto contiene tutte le specifiche che descrivono lo stato desiderato per quella risorsa.

  Il file in Figura descrive un Deployment e quindi troviamo informazioni come:
  - **replicas:** numero di repliche desiderate.
  - **template:** il pod template con cui creare il Pod e le sue repliche. È formato da diverse specifiche come quanti container con le relative immagine eseguire, informazioni di networking (e.g. porte da aprire), comandi da eseguire all'avvio dei container, dispositivi di storage (e.g. PVC), ecc.

Un altro campo importante è *status*, seppur non venga scritto dallo sviluppatore. Viene infatti aggiunto automaticamente da Kubernetes alla creazione della risorsa e contiene lo stato corrente dell'oggetto. I campi *spec* e *status* vengono usati dal controller del Control Plane che, come descritto in precedenza, gestisce attivamente le risorse per far sì che il loro stato corrente coincida sempre con lo stato desiderato.[63]

Una volta scritto il file descrittore, è possibile interagire con l'API server di Kubernetes mediante un tool chiamato kubectl e richiedere la creazione della risorsa.

Il comando da eseguire è il seguente:

```
kubectl apply -f <filename>
```

3.2.2 Helm

Per semplificare le procedure di deploy su cluster ho sfruttato Helm, uno strumento open-source per recuperare, condividere e utilizzare software realizzato per Kubernetes.[66]
In altri termini, Helm agisce da package manager per Kubernetes e assiste lo sviluppatore semplificando e velocizzando il processo di definizione, deploy e gestione delle applicazioni Kubernetes.
In Helm sono importanti tre concetti:[67]

- Il *chart*, ossia la collezione dei file YAML che descrivono le risorse Kubernetes necessarie a creare un'istanza di una certa applicazione.
- I *valori di configurazione*, con cui è possibile personalizzare le risorse del chart. Vengono infatti associati ad un certo chart per creare una release.
- La *release*, ossia l'istanza di chart che è in esecuzione sul cluster e che è stata creata combinando un chart con dei valori di configurazione.

Prima di procedere con la trattazione, è possibile fornire un semplice esempio utile ad intuire le potenzialità di Helm.
Supponiamo di voler distribuire un'istanza di Kafka su un cluster Kubernetes. Sarà necessario scrivere tutti i componenti applicativi sotto forma di risorse Kubernetes. Bisognerà quindi definire uno StatefulSet per rappresentare i broker, i relativi Headless service per esporre ciascun broker, risorse di storage quali PV e PVC, eventuali ConfigMap per i valori di configurazione, ecc.
Ciascuna di queste risorse richiede la scrittura del relativo file descrittore e, una volta distribuite sul cluster Kubernetes, è altresì necessario testarle per assicurarsi che interagiscano correttamente. Il procedimento può risultare lungo, tedioso e anche ripetitivo qualora si dovessero distribuire applicazioni simili su cluster diversi.
Helm sopperisce a queste necessità mettendo a disposizione un ricco hub da cui è possibile recuperare i file delle risorse Kubernetes per l'applicazione desiderata. Questi file sono stati realizzati e testati dalla comunità open source e sono quindi pronti per l'uso. Inoltre, sono stati scritti utilizzando un sistema di templating che permette allo sviluppatore di personalizzare la release che desidera creare.
Come illustrato in Figura 3.12, infatti, nei template compare l'oggetto built-in Values con cui è possibile accedere ai valori di configurazione. Questi possono derivare da quattro fonti diverse, qui presentate in ordine di priorità crescente:[68]

- Il file *values.yaml* presente all'interno del chart, in cui sono presenti i valori di configurazione di default.
- Se il chart in questione è un sub-chart, il file *values.yaml* proprio dei chart da cui discende.
- Il file che lo sviluppatore fornisce al comando 'helm install' o 'helm upgrade'.
- Eventuali valori individuali passati con il flag --set al comando 'helm install'.

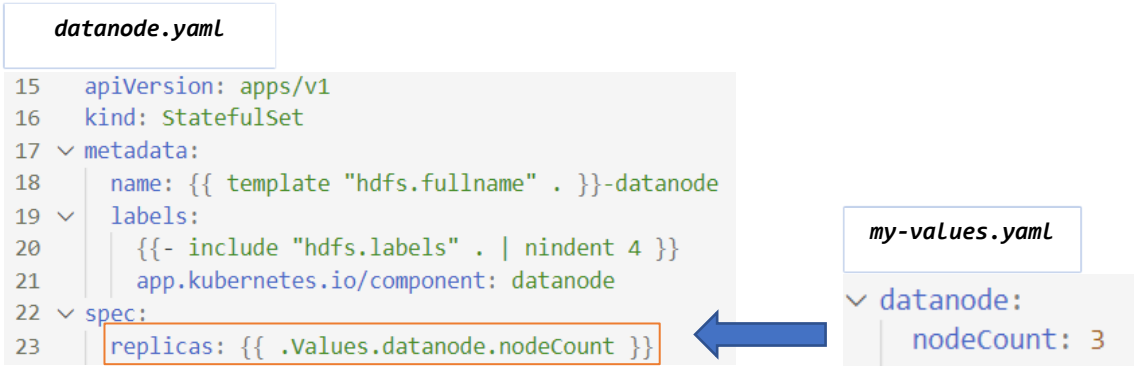


Figura 3.12 – Il sistema di templating usato in Helm permette di configurare la release. Sulla sinistra è riportato un template di un DataNode HDFS (datanode.yaml) realizzato come StatefulSet. Il numero di repliche viene scelto e fornito dallo sviluppatore tramite file di configurazione (my-values.yaml)

Vediamo ora quali sono le operazioni necessarie per eseguire il deploy di una release con Helm.

1. Innanzitutto, si ricerca un chart per l'applicazione desiderata. La ricerca si può condurre navigando con un browser sull'Hub Helm[33] oppure sfruttando il comando *helm* da CLI:

```
helm search hub [KEYWORD]
```

2. La repository contenente il chart trovato va aggiunta alla lista delle proprie repository locali.

```
helm repo add [NAME] [URL]
```

3. A questo punto è possibile procedere con l'installazione, eventualmente personalizzandola, e creare la release sul cluster Kubernetes.

```
helm install -f my-values.yaml <RELEASE-NAME> <CHART-NAME>
```

Nel mio progetto ho utilizzato Helm per il deploy dei componenti della pipeline interni al cluster Kubernetes: il "Data Ingestor", il "Connector & Pre-processor", il componente di storage e il "Frontend & ML Analyzer". Il prossimo paragrafo è dedicato alla descrizione del loro deploy e della loro implementazione.

[33] https://artifacthub.io/

3.3 I Componenti: implementazione e deploy

3.3.1 Il Cluster DISI

Per eseguire il deploy della pipeline mi sono state fornite due VM appartenenti al cluster di facoltà del dipartimento DISI. Le macchine presentano le seguenti caratteristiche:

Name	VCPUS	RAM	Total Disk	Root Disk	Ephemeral Disk	Public	OS
piazza-1	2	4 GB	40 GB	40 GB	0 GB	Yes	Ubuntu 20.04
piazza-2	2	4 GB	40 GB	40 GB	0 GB	Yes	Ubuntu 20.04

Tabella 3.1 –Specifiche tecniche dei nodi del cluster.

Tramite *ssh* ho potuto collegarmi alle macchine e configurarle perché formassero un cluster Kubernetes. A questo fine ho usato il tool *kubeadm*.
Preliminarmente però, su ciascun nodo è stato necessario installare:

- Docker come container-runtime.
- l'agente Kubelet.
- il tool *kubectl* per potersi interfacciarsi con l'API Server da linea di comando.

Successivamente, sfruttando *kubeadm,* è stato possibile inizializzare il Control Plane con tutti i suoi componenti. Infine, è stata installata una Pod Network per permettere la comunicazione tra i Pod e sono stati creati i Persistent Volume necessari ai vari componenti della pipeline.

È bene sottolineare che solitamente il nodo master Kubernetes si limita ad eseguire i processi del Control Plane e non esegue mai del carico applicativo. Nel mio caso però, dal momento che il cluster è composto di soli due nodi, ho ritenuto opportuno renderli entrambi dei nodi worker. Di conseguenza anche il nodo piazza-1, su cui eseguono i componenti del Control Plane, è stato predisposto per poter eseguire i vari Pod.
Per maggiori dettagli sul processo di configurazione e inizializzazione del cluster si faccia riferimento alla documentazione presente sulla repository GitHub del progetto[34].

[34] https://github.com/LorenzoPiazza/HeraSDG-BigDataAnalyticsPipeline

3.3.2 Datasource (Python Kafka Producer)

Il componente Datasource che ho realizzato per generare gli eventi della community Hera SDG è un Python Kafka Producer. Come citato in precedenza, si tratta di un componente *mock* che simula il verificarsi di tre tipi di eventi: la registrazione di un nuovo utente, il compimento di un comportamento virtuoso e l'acquisto di un premio.
Gli eventi presentano il seguente formato:

id_utente	Sesso	Data di Nascita	Eta	Provincia
CO_305	M	1987-10-18	33	BO

Evento utente

id_utente	Partner_erogante	comportamento	reward(tk)
CO_214	CONAD	Recupero bottiglie di plastica	0.9

Evento comportamento

id_utente	Partner_erogante	premio	prezzo(tk)
CA_8	HERA	Sconto in bolletta 10€ Hera	10

Evento premio

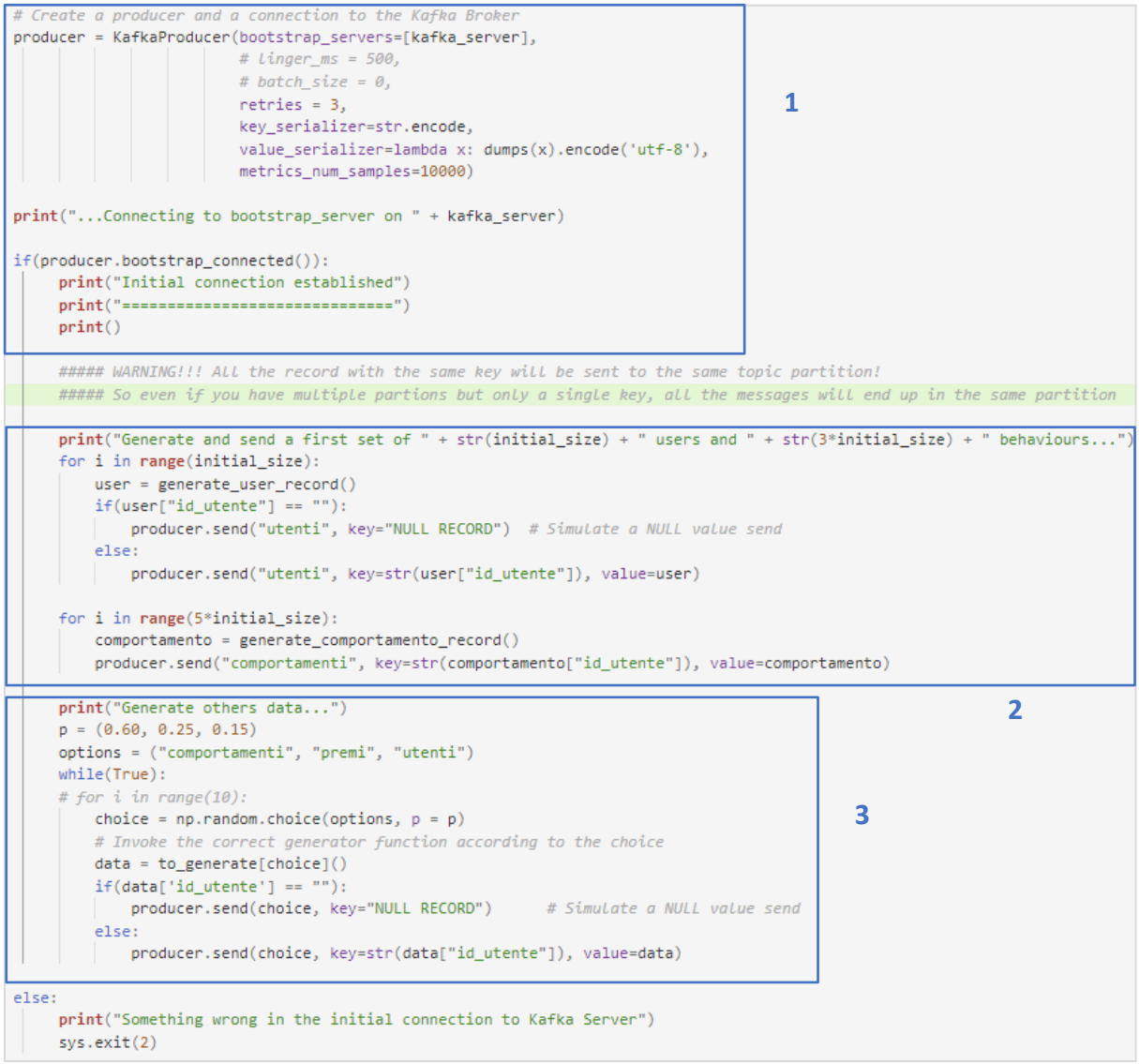
Figura 3.13 – Il formato dei dati generati dal mock datasource.

In tutti gli eventi è presente il campo `id_utente`, una stringa che rappresenta l'identificativo univoco dell'utente a cui si riferisce il record.
L'identificativo è formato in modo da esplicitare l'appartenenza dell'utente ad un certo partner. È composto, infatti, dalle due lettere iniziali del partner presso cui l'utente ha compiuto la registrazione, seguite da un numero progressivo comune a tutta la community (e.g. HE_243 si riferisce all'utente n° 243 della community ed informa che quell'utente si è registrato presso HERA).
Gli eventi vengono generati con dei valori pseudo-casuali, ottenuti grazie all'ausilio della libreria Numpy e della libreria Faker. Quest'ultima, in particolare, permette di fissare dei vincoli nella generazione, in modo da evitare di ottenere dei valori privi di senso o non adatti al contesto.
Ad esempio, permette di specificare un insieme di valori, ciascuno con la propria probabilità, dai quali estrarre in maniera casuale. Inoltre, offre alcune funzioni utili a generare dati e generalità che vengono comunemente usati. Tra queste, la funzione `fake.date_of_birth(minimum_age, maximum_age)` che permette di generare una data di nascita casuale compresa in un certo intervallo di età.

Il componente Datasource può essere eseguito da riga di comando con la seguente sintassi:

```
python data_source.py <bootstrap-server> <initial-data-size>
```

Il codice della funzione *main* del file `data_source.py` (Figura 3.14) può essere suddiviso in tre passaggi che ora mi accingo a descrivere:




```python
# Create a producer and a connection to the Kafka Broker
producer = KafkaProducer(bootstrap_servers=[kafka_server],
                         # linger_ms = 500,
                         # batch_size = 0,
                         retries = 3,
                         key_serializer=str.encode,
                         value_serializer=lambda x: dumps(x).encode('utf-8'),
                         metrics_num_samples=10000)

print("...Connecting to bootstrap_server on " + kafka_server)

if(producer.bootstrap_connected()):
    print("Initial connection established")
    print("==============================")
    print()

    ##### WARNING!!! All the record with the same key will be sent to the same topic partition!
    ##### So even if you have multiple partions but only a single key, all the messages will end up in the same partition

    print("Generate and send a first set of " + str(initial_size) + " users and " + str(3*initial_size) + " behaviours...")
    for i in range(initial_size):
        user = generate_user_record()
        if(user["id_utente"] == ""):
            producer.send("utenti", key="NULL RECORD")  # Simulate a NULL value send
        else:
            producer.send("utenti", key=str(user["id_utente"]), value=user)

    for i in range(5*initial_size):
        comportamento = generate_comportamento_record()
        producer.send("comportamenti", key=str(comportamento["id_utente"]), value=comportamento)

    print("Generate others data...")
    p = (0.60, 0.25, 0.15)
    options = ("comportamenti", "premi", "utenti")
    while(True):
    # for i in range(10):
        choice = np.random.choice(options, p = p)
        # Invoke the correct generator function according to the choice
        data = to_generate[choice]()
        if(data['id_utente'] == ""):
            producer.send(choice, key="NULL RECORD")      # Simulate a NULL value send
        else:
            producer.send(choice, key=str(data["id_utente"]), value=data)
else:
    print("Something wrong in the initial connection to Kafka Server")
    sys.exit(2)
```

Figura 3.14 – Il codice della funzione main del Datasource.

1. *Creazione del Kafka Producer e connessione con il bootstrap server.*

   Per la creazione del Kafka Producer è necessario specificare il `bootstrap_server` da contattare alla prima connessione. Può essere l'indirizzo IP di uno qualunque dei broker del cluster Kafka e viene fornito come parametro da riga di comando.

   Quando viene contattato per instaurare una connessione, il bootstrap server invia in risposta un insieme di metadati utili al produttore per gestire gli invii successivi. Tra questi troviamo

gli indirizzi di tutti i broker presenti nel cluster, la lista delle topic, delle partizioni e dei broker leader di ciascuna partizione.

In Figura 3.15 è illustrato un esempio dei metadati che vengono inviati al produttore in risposta alla prima connessione.

```
/ # kafkacat -b 137.204.57.224:30001 -L
Metadata for all topics (from broker -1: 137.204.57.224:30001/bootstrap):
 2 brokers:
  broker 0 at 127.0.0.1:30001
  broker 1 at 127.0.0.1:30002 (controller)
 9 topics:
  topic "test" with 2 partitions:
    partition 0, leader 0, replicas: 0,1, isrs: 0,1
    partition 1, leader 1, replicas: 1,0, isrs: 1,0
```

⋮

Figura 3.15 – Metadati che il bootstrap server invia in risposta al produttore. Quelli presenti in questo esempio sono stati ottenuti sfruttando kafkacat, un tool di debugging per Kafka.

Altri parametri rilevanti nella creazione del producer sono `retries`, ossia il numero di tentativi che vengono fatti in caso di fallimento nell'invio di un record, `key_serializer` e `value_serializer`, rispettivamente le funzioni con cui serializzare la chiave e il valore di ciascun record prima dell'invio. Nel mio progetto ho deciso di serializzare le chiavi come stringhe e i valori come dati JSON.

2. *Generazione e invio di un primo gruppo di utenti e di comportamenti.*

   In questa fase il Datasource genera ed invia un numero `initial_data_size` di utenti e `5*initial_data_size` comportamenti. Questa scelta permette di simulare la presenza di un gruppo di utenti registrati alla community che hanno già compiuto dei comportamenti e che quindi dispongono di un certo numero di token da poter spendere in premi.

   Similmente a quanto avviene nella realtà, infatti, quando il mio Datasource genera un evento "premio", tenta di assegnargli un `id_utente` che disponga di abbastanza token necessari al suo acquisto. Per realizzare questo comportamento il Datasource mantiene traccia del bilancio dei token di ciascun utente.

3. *Generazione ed invio di ulteriori eventi.*

   Da questo momento in poi il Datasource entra in un loop infinito nel quale genera continuamente uno dei tre eventi, secondo probabilità prestabilite, e lo invia sulla relativa topic Kafka con la funzione `send`.

   La funzione send è asincrona in quanto, come riporta la documentazione ufficiale, «aggiunge il record ad un buffer di altri record in attesa di essere inviati e ritorna

immediatamente».[69] In questo modo il Datasource può aggregare più record da inviare in un unico batch (di dimensione configurabile) garantendo operazioni di invio più efficienti.
Con riferimento alla Figura 3.14, sezione 3, si può notare che il produttore usa il campo `id_utente` come chiave per i record. In questo modo tutti record dello stesso evento e riferiti allo stesso utente verranno pubblicati nella stessa partizione. Inoltre, si sfruttano tutte le partizioni presenti, requisito necessario per garantire parallelismo nelle scritture e nelle letture da e verso Kafka.
Il Datasource, infine, simula anche l'invio di alcuni record con valore NULL. Questi verranno filtrati dal componente "Connector & Pre-processor" che legge i dati da Kafka e li scrive su HDFS.

3.3.3 Data Ingestor (Kafka cluster)

Il componente di Data Ingestion è un cluster Kafka formato da due broker. Il deploy sul cluster Kubernetes è stato realizzato usando l'Helm chart *bitnami/kafka* e personalizzando i template con dei valori di configurazione che ho fornito tramite file.
Il comando Helm che ho utilizzato per installare la release è il seguente:

```
helm install -f my-kafka-values.yaml my-kafka bitnami/kafka
```

dove:

- *my-kafka-values.yaml* è il file con i valori di configurazione.
- *my-kafka* è il nome da assegnare alla release che viene installata.
- *bitnami/kafka* è il nome del chart da utilizzare.

Considerato che il codice sorgente dei template del chart è molto vasto, in questa trattazione vi si farà riferimento limitandosi ad enunciare le risorse che compongono la release e a riportare i valori di configurazione più significativi che ho fornito.
Per maggiori dettagli riguardo al codice sorgente dei template, e per capire in quali sezioni si inseriscano i suddetti valori, si faccia riferimento alla sezione "template" della pagina web ufficiale del chart.[35]

Fatte queste premesse è possibile iniziare a descrivere la release che ho installato.
Essa è composta da due StatefulSet, dei quali uno gestisce i broker Kafka mentre l'altro gestisce un'istanza Zookeeper, sfruttata per la coordinazione tra i broker.

---

[35] https://artifacthub.io/packages/helm/bitnami/kafka

I valori di configurazione più importanti che ho fornito tramite il file *my-kafka-values.yaml* sono i seguenti:

Figura 3.16 – Alcuni frammenti del file di configurazione my-kafka-values.yaml in cui sono presenti i principali valori usati per personalizzare la release del componente "Data Ingestor".

Come si può notare dalla sezione "Broker Configurations", lo StatefulSet Kafka gestisce un numero `replicaCount=2` di Pod, ciascuno dei quali rappresenta un broker. Ogni broker dispone di un identificativo stabile composto dal nome della release e da un numero progressivo a partire da `minBrokerId=0`. Grazie alla stabilità dell'identificativo, i broker possiedono la *sticky identity* descritta nel paragrafo 3.2.1 e possono associarsi persistentemente, tramite una PVC, ad un Persistent Volume.

Nella sezione "Persistence Configurations" si può notare come ogni broker faccia richiesta di un PV con modalità di accesso `ReadWriteOnce` e una dimensione di almeno `10Gi`.

Per permettere al Datasource di comunicare con i broker, essendo questi un componente esterno al cluster, ho configurato un servizio NodePort che espone le porte `30001` e `30002` su entrambi i nodi del cluster. La prima è configurata per inoltrare il traffico al broker *my-kafka-0* mentre la seconda per inoltrarlo al broker *my-kafka-1*.
Nella release, inoltre, sono presenti dei servizi ClusterIP e Headless che permettono la comunicazione con i broker anche dall'interno del cluster.

Le Figure 3.17 e 3.18 mostrano l'output dei comandi `kubectl get pods` e `kubectl get svc` con i quali, interfacciandosi con il cluster Kubernetes, vengono mostrati i Pod e i servizi relativi alla release installata.

```
$ kubectl get pods -l app.kubernetes.io/component=kafka
NAME         READY   STATUS    RESTARTS   AGE
my-kafka-0   1/1     Running   0          40h
my-kafka-1   1/1     Running   0          40h
```

Figura 3.17 – I due Pod, broker del cluster Kafka.

```
$ kubectl get svc -l app.kubernetes.io/component=kafka
NAME                  TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)              AGE
my-kafka              ClusterIP   10.100.232.127   <none>        9092/TCP             41h
my-kafka-0-external   NodePort    10.98.165.115    <none>        9094:30001/TCP       41h
my-kafka-1-external   NodePort    10.109.115.249   <none>        9094:30002/TCP       41h
my-kafka-headless     ClusterIP   None             <none>        9092/TCP,9093/TCP    41h
```

Figura 3.18 – I servizi ClusterIP, NodePort e Headless.

Un altro aspetto importante da evidenziare è che la release è predisposta per creare in modo automatico le topic richieste. Al momento dell'installazione, infatti, viene eseguito un Job Kubernetes con questo compito. Come si vede in Figura 3.16, sezione "Topic Configurations", ho deciso di creare tre topic (utenti, comportamenti e premi) con due partizioni ciascuna e fattore di replicazione pari a due.
La scelta del numero di partizioni e di repliche è motivata dal voler garantire il massimo parallelismo e la massima fault tolerance possibile con il numero di nodi a disposizione, che nel mio caso sono due.
Ho deciso anche di settare una `retention.ms` pari a due giorni, periodo dopo il quale i messaggi più vecchi vengono eliminati dalle topic. In questo modo la memoria dei broker Kafka non viene inutilmente occupata da messaggi che, dopo due giorni dalla loro pubblicazione, si assume siano già stati consumati dal componente "Connector & Pre-processor" della pipeline.

Come descritto nel capitolo 2, Kafka si occupa di distribuire le partizioni delle topic e le loro repliche tra i vari broker del cluster. Garantisce anche che ciascun broker sia leader di un numero equo di partizioni.

Nel mio progetto ho usato il tool *kafkacat* per verificare che effettivamente venisse applicato il bilanciamento:

```
kafkacat -b <bootstrap-server> -t <topic> -L
```

Il comando, eseguito rispettivamente per le tre topic, ha restituito i seguenti metadati:

```
topic "comportamenti" with 2 partitions:
  partition 0, leader 1, replicas: 0,1, isrs: 1,0
  partition 1, leader 0, replicas: 0,1, isrs: 0,1
```

```
topic "premi" with 2 partitions:
  partition 0, leader 0, replicas: 0,1, isrs: 0,1
  partition 1, leader 1, replicas: 0,1, isrs: 1,0
```

```
topic "utenti" with 2 partitions:
  partition 0, leader 1, replicas: 0,1, isrs: 1,0
  partition 1, leader 0, replicas: 0,1, isrs: 0,1
```

Figura 3.19 – Informazioni riguardanti le 3 topic Kafka.
La notazione vuole che il termine "leader" sia seguito dall'id del broker leader di quella partizione.
Il termine "replicas", invece, dagli id dei broker che ospitano almeno una replica di quella partizione.

Si può notare come le partizioni e le repliche siano equi-partite tra i due broker del mio cluster: ciascun broker, infatti, è leader di tre partizioni su sei.
In forma grafica la situazione descritta dagli output precedenti è la seguente:

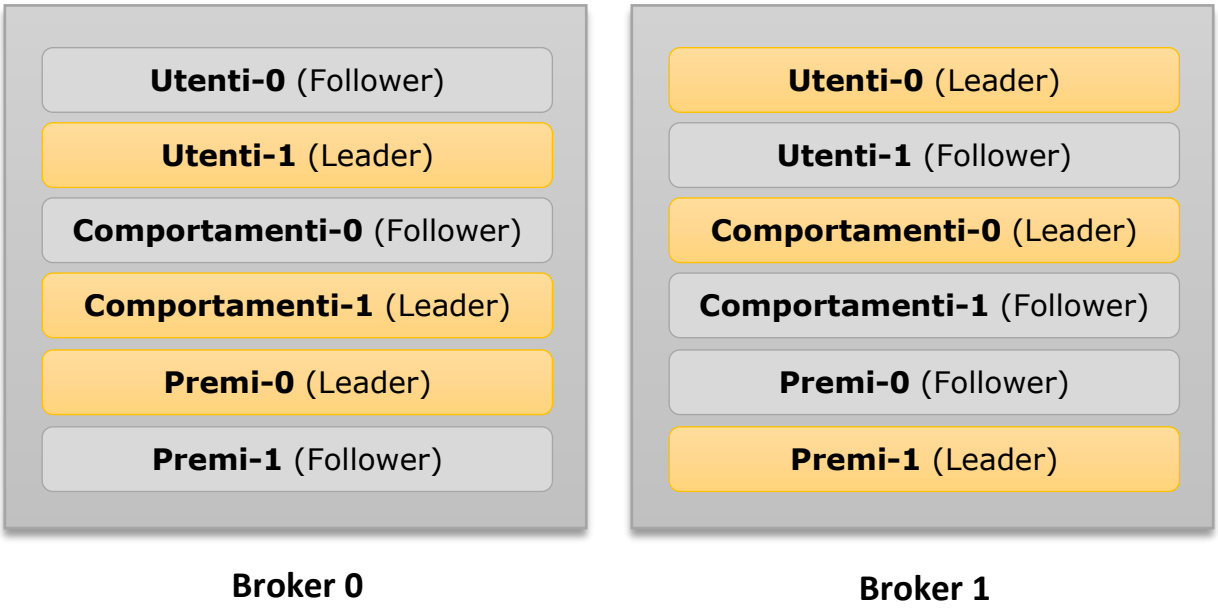


Figura 3.20 – La distribuzione delle topic tra i due broker del cluster Kafka.

3.3.4 Connector & Pre-Processor (Kafka HDFS Sink Connector)

Per eseguire il deploy del componente "Connector & Pre-Processor" ho sfruttato il campo *extraDeploy* presente nel bitnami/kafka chart, lo stesso chart utilizzato per il Data Ingestor.
All'interno del file *my-kafka-values.yaml,* infatti, è possibile descrivere una lista aggiuntiva di risorse Kubernetes che si desidera installare in aggiunta a quelle predefinite.
Queste risorse verranno installate sempre all'esecuzione del comando `helm install` presentato nello scorso paragrafo. La release my-kafka può essere così arricchita dagli oggetti Kubernetes necessari a realizzare il componente Connector.
In particolare, le risorse extra di cui viene fatto il deploy sono:

- un Deployment che gestisce un insieme di Kafka HDFS Sink Connector eseguiti in modalità distribuita.
- Un servizio di tipo Cluster IP per esporli.
- Una ConfigMap[36] chiamata *my-kafka-connect-config* con il file di configurazione dei worker.
- Una ConfigMap chiamata *my-kafka-connect-script* contenente il codice che deve essere eseguito all'interno del Pod per avviare un worker in modalità distribuita e creare una connector instance.

La scelta di distribuire il componente "Connector & Pre-Processor" come Deployment e non come StatefulSet è stata fatta considerando che i connect worker sono processi *stateless*. Essi, infatti, non hanno la necessità di mantenere una propria identità tra i riavvii e nemmeno di salvare i dati che leggono in maniera persistente.
Come descritto nel capitolo 2.2.4, i worker che eseguono in modalità distribuita memorizzano il proprio stato di consumatori all'interno di topic Kafka create a questo scopo. Quindi, delegano l'onere della persistenza dei propri dati a Kafka, sgravandosi da questa responsabilità.

Nelle due pagine che seguono viene riportato il file YAML descrittore del Deployment. Sono stati evidenziati i punti salienti, in modo da potervisi riferire nella trattazione che subito segue.

[36] Le ConfigMap sono un tipo di risorsa Kubernetes che permette di esternalizzare la configurazione di un determinato Pod. Un'altra loro funzione, quando montate come volumi, è quella di *iniettare* all'interno del Pod un file con un contenuto a piacere dello sviluppatore. Si possono così iniettare dei file di configurazione o anche dei file contenenti codice eseguibile.

```
extraDeploy:
  - |
    apiVersion: apps/v1
    kind: Deployment
    metadata:
      name: {{ include "kafka.fullname" . }}-connect
      labels: {{- include "common.labels.standard" . | nindent 4 }}
        app.kubernetes.io/component: connector
    spec:
      replicas: 1
      selector:
        matchLabels: {{- include "common.labels.matchLabels" . | nindent 6 }}
          app.kubernetes.io/component: connector
      template:
        metadata:
          labels: {{- include "common.labels.standard" . | nindent 8 }}
            app.kubernetes.io/component: connector
        spec:
          affinity:
            podAntiAffinity:
              preferredDuringSchedulingIgnoredDuringExecution:
              - weight: 1
                podAffinityTerm:
                  labelSelector:
                    matchExpressions:
                    - key: app.kubernetes.io/component
                      operator: In
                      values:
                      - connector
                  topologyKey: "kubernetes.io/hostname"
            podAffinity:
              preferredDuringSchedulingIgnoredDuringExecution:
              - weight: 1
                podAffinityTerm:
                  labelSelector:
                    matchExpressions:
                    - key: app.kubernetes.io/component
                      operator: In
                      values:
                      - kafka
                      - datanode
                  topologyKey: "kubernetes.io/hostname"
          # An initial container to install some useful packages:
          initContainers:
          - name: installer
            image: python:3
            command: ["/bin/sh", "-c"]
            args: ["pip install -t /additionalSoftware kafka-connect-healthcheck"]
            volumeMounts:
```

```
            - name: extra-installation
              mountPath: "/additionalSoftware"
          containers:
            - name: connect
              image: confluentinc/cp-kafka-connect-base:6.1.0
              imagePullPolicy: IfNotPresent
              resources:
                 requests:
                   memory: "256Mi"
                   cpu: “200m”
              command: ["/bin/sh", "-c"]
              args:
                - confluent-hub install --no-prompt confluentinc/kafka-connect-
hdfs3:1.1.1 && /opt/bitnami/kafka/scripts/launch-hdfs-sink-connector.sh;
              ports:
                - name: connector
                  containerPort: 8083
              env:
                - name: CONNECT_REST_ADVERTISED_HOST_NAME
                  valueFrom:
                    fieldRef:
                      fieldPath: status.podIP
              volumeMounts:
                - name: configuration
                  mountPath: /opt/bitnami/kafka/config
                - name: create-connector-script
                  mountPath: /opt/bitnami/kafka/scripts
                - name: extra-installation
                  mountPath: "/additionalSoftware"
              livenessProbe:
                httpGet:
                  path: /
                  port: 18083
                failureThreshold: 1
                periodSeconds: 20
              startupProbe:
                httpGet:
                  path: /
                  port: 18083
                failureThreshold: 15
                periodSeconds: 20
          volumes:
            - name: configuration
              configMap:
                name: {{ include "kafka.fullname" . }}-connect-config
            - name: create-connector-script
              configMap:
                name: {{ include "kafka.fullname" . }}-connect-script
                defaultMode: 0777
            - name: extra-installation
```

```
            emptyDir: {}
```

Il Deployment appena descritto gestisce un numero `replicas: 1` di Pod, ciascuno rappresentante un Connect worker che esegue in modalità distribuita. Al momento dell'installazione ho scelto di creare un solo Connect worker e di affidare la gestione del suo scaling ad un Horizontal Pod Autoscaler descritto nel paragrafo 3.4.

I Pod eseguono l'immagine `confluentinc/cp-kafka-connect-base:6.1.0` ma in fase di esecuzione la arricchiscono installando i plugin necessari a realizzare un Confluent HDFS3 Sink Connector. Il comando eseguito per questo scopo è `confluent-hub install --no-prompt confluentinc/kafka-connect-hdfs3:1.1.1`).[37]

I Pod dispongono di due volumi di tipo ConfigMap che servono ad iniettare all'interno del Pod i file *worker.properties* e *launch-hdfs-sink-connector.sh*.
Quest'ultimo script è molto importante: viene eseguito come comando di avvio del Pod e contiene il codice per avviare il worker.
Il contenuto dello script può essere suddiviso in tre sezioni:

[37] Una soluzione più efficiente di quella presentata sarebbe creare direttamente un'immagine con già i plugin HDFS3 Sink Connector installati e fare utilizzare direttamente quella ai Pod del Deployment, evitando che ogni Pod debba installarseli all'avvio. Tuttavia, al momento della realizzazione di questa tesi, i plugin HDFS3 Sink Connector presentano una licenza di utilizzo gratuita per 30 giorni a partire dall'installazione.
La soluzione che ho adottato, quindi, mi ha permesso di far ripartire il computo dei 30 giorni ogni volta che un nuovo Pod worker viene eseguito.
È ovvio che questa soluzione rappresenti un escamotage valido solo in un contesto di testing. Per un eventuale passaggio del progetto alla fase di produzione saranno necessarie ulteriori valutazioni a riguardo.

Per maggiori informazioni sui plugin e sulla licenza si faccia riferimento al seguente link: https://www.confluent.io/hub/confluentinc/kafka-connect-hdfs3
URL visitato il: 04 maggio '21.

1. Nel primo step (Figura 3.21) viene avviato in background e in modalità distribuita un processo worker fornendogli il file di configurazione *worker.properties*.
   Questo file contiene alcuni importanti parametri come:
   - `group.id`: id del gruppo. Lo stesso id viene fornito a tutti i worker in modo tale che appartengano allo stesso gruppo di consumatori.
   - Numero di partizioni e fattore di replicazione delle topic utilizzate dai worker per salvare i propri offset di consumazione, il proprio stato e le proprie configurazioni.
   - `key.converter`: la classe con cui viene eseguita la deserializzazione della chiave dei record letti.
   - `value.converter`: la classe con cui viene eseguita la deserializzazione del valore dei record letti.

```
launch-hdfs-sink-connector.sh: |-
  #!/bin/sh
  # 1. Launch (in background) Kafka Connect Worker in distributed mode
  echo -e "\n\n=============\nLaunching the Kafka Connect WORKER in Distributed mode...\n=============\n"
  /bin/connect-distributed /opt/bitnami/kafka/config/worker.properties &
```

Figura 3.21 – Codice dello script launch-hdfs-sink-connector.sh (parte 1).

2. Nel secondo step (Figura 3.22) si attende che il worker sia pronto.
   Il check viene realizzato con un ciclo *while* all'interno del quale vengono inviate richieste HTTP all'interfaccia REST da esso esposta, fintanto che non viene ritornato il valore HTTP_CODE=200.

```
  # 2.Wait for Kafka connect REST listener
  echo -e "\n\n=============\nWaiting for Kafka Connect to start listening on localhost ⏳\n=============\n"
  while [ $(curl -s -o /dev/null -w %{http_code} http://localhost:8083/connectors) -ne 200 ] ; do
    echo -e "\t" $(date) " Kafka Connect listener HTTP state: " $(curl -s -o /dev/null -w %{http_code} http://localhost:8083/connectors)
    sleep 5
  done
  echo -e $(date) "\n\n--------------\n\o/ Kafka Connect is ready! Listener HTTP state: " $(curl -s -o /dev/null -w %{http_code} http://
```

Figura 3.22 – Codice dello script launch-hdfs-sink-connector.sh (parte 2).

3. Nel terzo ed ultimo step (Figura 3.23) viene fatta una chiamata POST al worker per richiedere la creazione di una connector instance.
   Come dati del payload (parametro `--data` della chiamata `curl`) vengono passate le proprietà desiderate per configurare la connector instance.

```
# 3. Create HDFS sink connector with a REST call using curl
echo -e "\n\n=============\nCreating the specific CONNECTOR with a REST call...\n=============\n"
curl -X PUT \
  -H "Content-Type: application/json" \
  --data '{
            "_comment": "--- The HDFS SINK connector class ---",
            "connector.class"                       : "io.confluent.connect.hdfs3.Hdfs3SinkConnector",

            "_comment": " --- HDFS SINK-specific configuration below here  --- ",
            "tasks.max"                             : "6",
            "topics"                                : "utenti, comportamenti, premi",
            "hdfs.url"                              : "hdfs://my-hdfs-namenodes:8020",
            "hadoop.conf.dir"                       : "/etc/hadoop/",
            "hadoop.home"                           : "/opt/hadoop-3.1.3/share/hadoop/common",
            "logs.dir"                              : "/tmp",
            "confluent.topic.bootstrap.servers"     : "{{ include "kafka.fullname" . }}-0.{{ include "kafka.fullname" . }}-headless.

            "_comment": " --- Partition Storage Strategy  --- ",
            "partitioner.class"                     : "io.confluent.connect.storage.partitioner.TimeBasedPartitioner",
            "path.format"                           : "YYYY/MM/dd/HH",
            "partition.duration.ms"                 : "600000",
            "locale"                                : "fr-FR",
            "timezone"                              : "Europe/Paris",
            "flush.size"                            : "2000000",
            "rotate.schedule.interval.ms"           : "600000",
            "format.class"                          : "io.confluent.connect.hdfs3.json.JsonFormat",
            "topics.dir"                            : "/HeraSDG/raw_data",
            "filename.offset.zero.pad.width"        : "6",

            "_comment": " --- Error Handling: dead letter queue  --- ",
            "errors.tolerance": "all",
            "errors.deadletterqueue.topic.name":"dlq_hdfs-sink",
            "errors.deadletterqueue.topic.replication.factor": 1,

            "_comment": " --- Metrics: see metrics.reporters configuration values  --- ",

            "_comment": " --- Single Message Transformation: Filtering  --- ",
            "transforms"                            : "dropNullRecords",
            "transforms.dropNullRecords.type"       : "org.apache.kafka.connect.transforms.Filter",
            "transforms.dropNullRecords.predicate"  : "isNullRecord",

            "predicates"                            : "isNullRecord",
            "predicates.isNullRecord.type"          : "org.apache.kafka.connect.transforms.predicates.RecordIsTombstone"
        }' \
  http://localhost:8083/connectors/hdfs3-sink/config
```

1

3

4

2

Figura 3.23 – Codice dello script launch-hdfs-sink-connector.sh (parte 3).

Con riferimento ai riquadri evidenziati in Figura 3.23, i valori di configurazione più importanti possono essere suddivisi nelle seguenti sezioni:

1. HDFS Sink Connector Configuration

La connector instance è stata configurata per poter creare al massimo `tasks.max: 6` task e per leggere dalle `topics: utenti, comportamenti, premi`.

Dal momento che ciascuna topic presenta due partizioni, la creazione di sei task garantisce il massimo parallelismo, potendo assegnare a ciascuna partizione un task.

2. Single Message Transformation - Il pre-processing dei record

Come descritto nel capitolo 2.2.4, i Kafka Connector permettono di eseguire delle trasformazioni a livello di singolo messaggio (Single Message Transformation). Queste permettono al mio componente di effettuare un pre-processing dei record prima di scriverli sul componente di storage.

Nello specifico, ho predisposto il connector perché applichi una trasformazione chiamata `dropNullRecords` che consiste nel filtrare i record con valore NULL che dovessero essere stati pubblicati per errore nel componente di Data Ingestion.
La sintassi di dichiarazione della Single Message Transformation vuole che si specifichi la classe che implementa il metodo di trasformazione, in questo caso la classe `org.apache.kafka.connect.transform.Filter`, e altri parametri da essa richiesti.
Per le trasformazioni di filtraggio, ad esempio, è necessario specificare il predicato da usare per determinare se un record va filtrato oppure no. In questo caso il predicato è del tipo `org.apache.kafka.connect.transforms.predicates.RecordIsTombstone`.

3. Partition Storage Strategy - La scrittura sul Data Storage

In questa sezione di codice, il connector viene istruito su come scrivere i file nel componente "Data Storage" HDFS. In particolare, viene utilizzato un `TimeBasedPartitioner` con `partition.duration.ms=600000`. Ciascun task, quindi, legge i record dalla partizione Kafka che gli è stata assegnata, e ogni 10 min comanda un flush sul componente di Data Storage scrivendo un file JSON.
È importante sottolineare che dalla scelta del parametro `partition.duration.ms` dipende la dimensione dei file che effettivamente vengono scritti su HDFS.
Dal momento che HDFS è un filesystem ottimizzato per gestire file di medio-grandi dimensioni (almeno pari a 64-128 Mb, cioè alla dimensione di un blocco) sarà fondamentale un tuning di questo parametro prima di dichiarare la pipeline *production-ready*.
Il valore di `partition.duration.ms` è inversamente correlato con il data-rate di pubblicazione dei Datasource e con la dimensione dei singoli record che vengono pubblicati. Minore è il data-rate di pubblicazione (analogamente la dimensione dei singoli record), maggiore dovrà essere la `partition.duration.ms` per poter garantire la scrittura di file che non siano troppo piccoli dal punto di vista di HDFS.

4. Dead Letter Queue - Gestione degli errori

Per gestire eventuali errori di lettura e processamento dei record viene utilizzata una Dead Letter Queue (DLQ). Potrebbe infatti capitare che su Kafka vengano pubblicati erroneamente dei record con un diverso formato da quello che il connector si aspetta. In condizioni normali, il task che legge un simile record scatenerebbe un errore e fallirebbe. Con la presenza di una DLQ invece, il task può evitare il fallimento: la DLQ, infatti, è una topic aggiuntiva, nel mio caso chiamata `dlq-hdfs-sink`, nella quale il task andrà a scrivere tutti i messaggi che non è in grado di processare. Da questa topic poi, potranno essere letti in futuro da altri Connector che invece sono in grado di gestirli.

3.3.5 Data Storage (HDFS cluster)

Il componente di Data Storage è un cluster HDFS, composto da due DataNode e un NameNode. Il deploy sul cluster Kubernetes è stato realizzato usando l'Helm chart *gaffer/hdfs* e personalizzando i template con dei valori di configurazione che ho fornito tramite file.
Il comando Helm che ho utilizzato per installare la release è il seguente:

```
helm install -f my-hdfs-values.yaml my-hdfs gaffer/hdfs
```

dove:

- *my-hdfs-values.yaml* è il file con i valori di configurazione.
- *my-hdfs* è il nome da assegnare alla release che viene installata.
- *gaffer/hdfs* è il nome del chart da utilizzare.

La release installata è composta da due StatefulSet, gestori rispettivamente del NameNode e dei DataNode, e da un Deployment chiamato *my-hdfs-shell*. Quest'ultimo mette a disposizione dello sviluppatore un Pod al quale è possibile connettersi per eseguire, qualora se ne avesse la necessità, dei comandi di gestione del Filesystem.
Oltre ai sopra citati Pod Controller, vengono installati dei servizi ClusterIP e Headless Service. Questi sono sufficienti a permettere la comunicazione con il componente di Data Storage dal momento che non c'è l'esigenza di contattarlo dall'esterno del cluster.

I più importanti valori di configurazione che ho fornito tramite il file *my-hdfs-values.yaml* sono i seguenti:

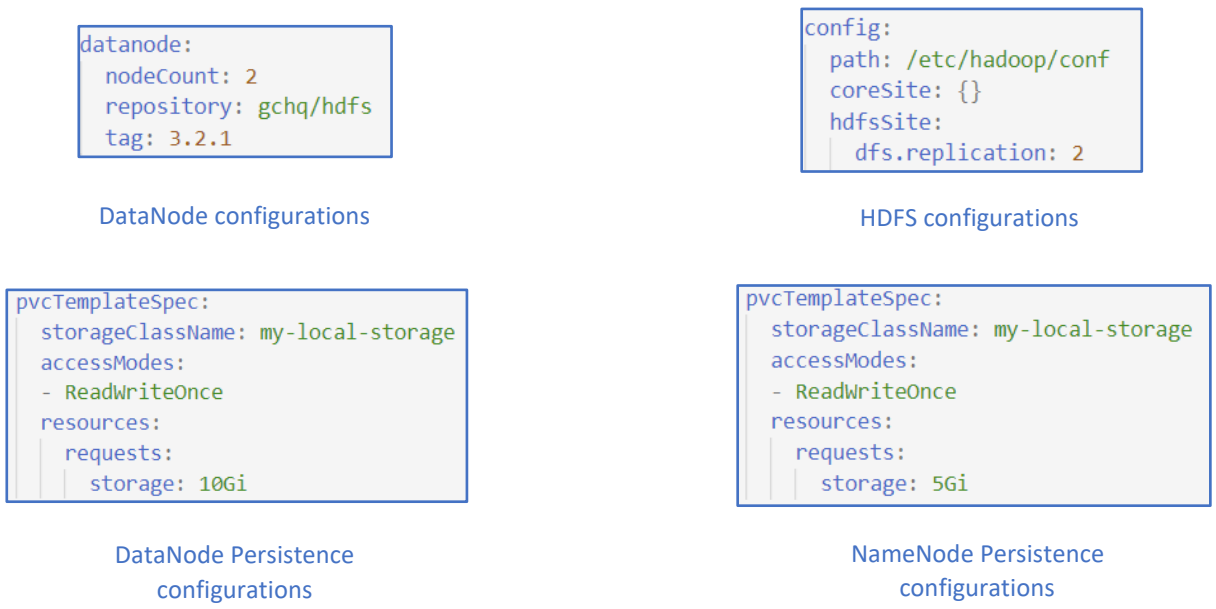


Figura 3.24 – Alcuni frammenti del file di configurazione my-hdfs-values.yaml in cui sono presenti i principali valori usati per personalizzare la release del componente "Data Storage".

Si può notare come lo StatefulSet riguardante i DataNode gestisca un numero `nodeCount=2` di Pod, ciascuno rappresentante un DataNode. Questi eseguono l'immagine `gchq/hdfs:3.2.1` e dispongono di un PV con modalità di accesso `ReadWriteOnce` e capacità di `10Gi`.

Nella sezione "HDFS Configurations" viene specificato `dfs.replication=2`, con il quale si richiede un fattore di replicazione dei blocchi pari a 2. La dimensione dei blocchi è stata lasciata invariata e quindi è pari a 128Mb.

Infine, si noti come il NameNode disponga di un PV di capacità pari a `5Gi`, quindi inferiore rispetto a quella dei DataNode, in quanto non ha la necessità di salvare i dati applicativi.

Nello sviluppo del componente di Data Storage si è rivelata utile la possibilità di utilizzare la web UI HDFS, accessibile alla porta `9870` del Pod NameNode. Questa offre un supporto grafico con cui monitorare lo stato dei DataNode, consultare informazioni riguardo l'occupazione attuale di memoria, visualizzare l'elenco dei valori di configurazione HDFS ed esplorare il filesystem.

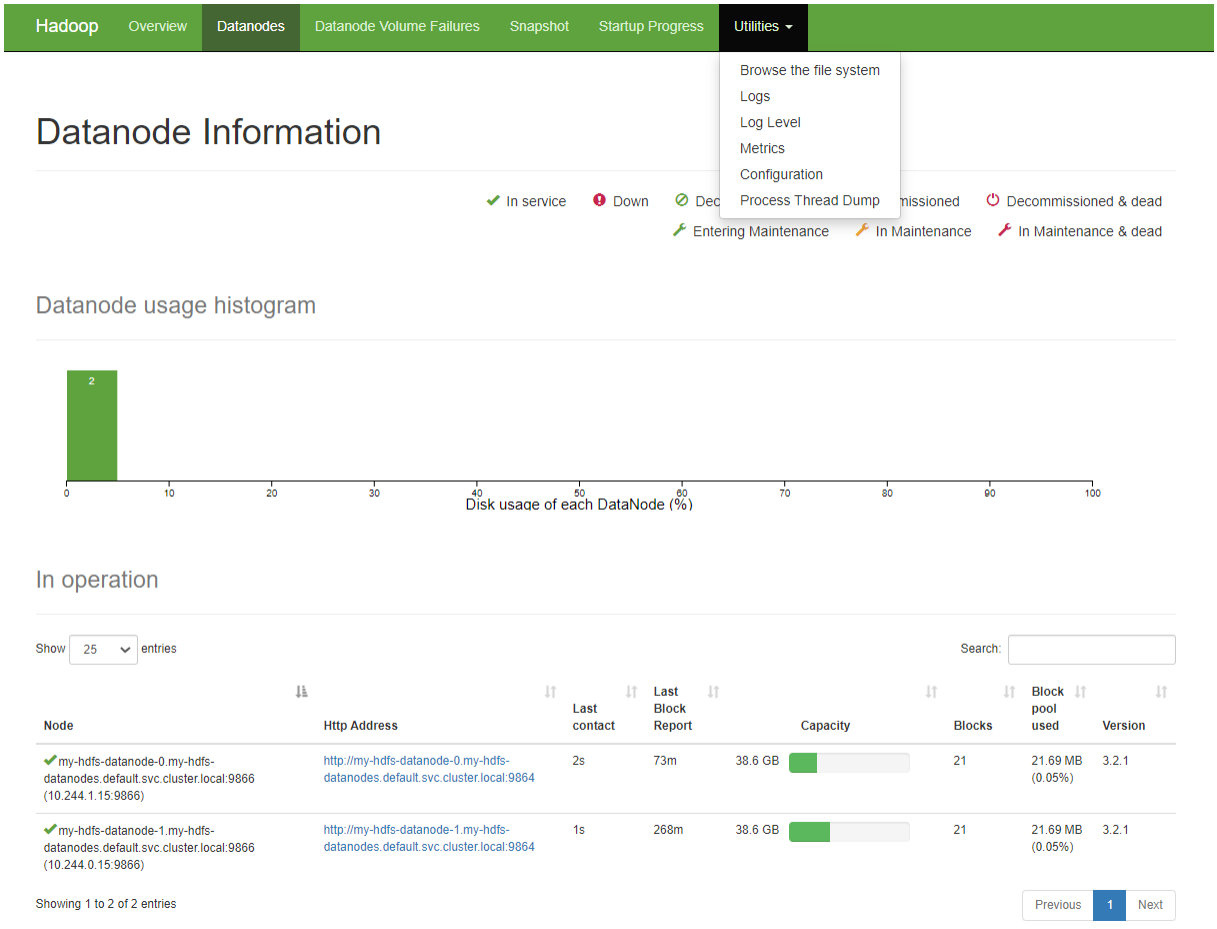


Figura 3.25 – Una schermata della web UI fornita da HDFS.

3.3.6 Frontend & ML Analyzer (Jupyter Lab & Spark)

Il deploy di questo componente sul cluster Kubernetes è stato realizzato usando l'Helm chart *gradiant/jupyter* e personalizzando i template con dei valori di configurazione che ho fornito tramite file.
Il comando Helm che ho utilizzato per installare la release è il seguente:

```
helm install -f my-jupyter-values.yaml my-jupyter gradiant/jupyter
```

dove:

- *my-jupyter-values.yaml* è il file con i valori di configurazione.
- *my-jupyter* è il nome da assegnare alla release che viene installata.
- *gradiant/jupyter* è il nome del chart da utilizzare.

La release installata risulta composta da uno StatefulSet che gestisce un unico Pod, chiamato my-jupyter-0. Questo Pod, all'avvio, inizializza un ambiente Jupyter Lab ed espone il Frontend sulla porta `8888`.
Jupyter Lab è un ambiente di sviluppo interattivo e web-based nel quale è possibile realizzare, tra le altre cose, dei Jupyter Notebook. Questi ultimi sono particolarmente diffusi in ambito data science e machine learning poiché sono dei file in cui lo sviluppatore ha la facoltà di mescolare codice di programmazione con elementi grafici e sezioni di linguaggio Markdown.

Il Pod my-jupyter-0 dispone di un PV di dimensione `1Gi` nel quale è possibile salvare in maniera persistente i Notebook che vengono prodotti.

Inoltre, esegue l'immagine `jupyter/pyspark-notebook:4d9c9bd9ced0`, la quale è equipaggiata con PySpark 3.1.1. Per quanto riguarda la parte di ML, infatti, il "Frontend & ML Analyzer" sfrutta Spark come framework di computazione, eseguendolo nella modalità integrata con Kubernetes.

Esecuzione di Spark su K8s

A partire dalla versione Spark 2.3 è stato aggiunto un supporto per l'esecuzione nativa di Spark su Kubernetes. In seguito, con la release Spark 3.1 rilasciata a marzo 2021, l'esecuzione di Spark su Kubernetes è stata dichiarata production-ready.[70]
Quando viene richiesta l'esecuzione di un applicazione Spark su Kubernetes il programma contatta direttamente l'API server di Kubernetes richiedendo la creazione di un Pod in cui eseguire il driver Spark.
Una volta che il driver è stato creato, anch'esso si interfaccerà con l'API server ma questa volta per richiedere la creazione degli executor. L'API server delegherà allo Scheduler la creazione

e lo scheduling di un Pod per ogni executor richiesto dal driver.[71] A questo punto il driver avrà a propria disposizione un cluster di executor a cui poter delegare l'esecuzione dei task.
Una volta che l'applicazione Spark completa la propria esecuzione, i Pod executor terminano e vengono rimossi. Il driver Pod, invece, passa in uno stato "completed" e continua ad esistere, seppur non consumando più alcuna risorsa, per dare la possibilità di consultare i suoi log.
La sua eliminazione può essere richiesta esplicitamente oppure lasciata a carico del garbage collector, che prima o poi provvederà a rimuoverlo.[72]
I passaggi appena descritti sono illustrati in Figura 3.26:

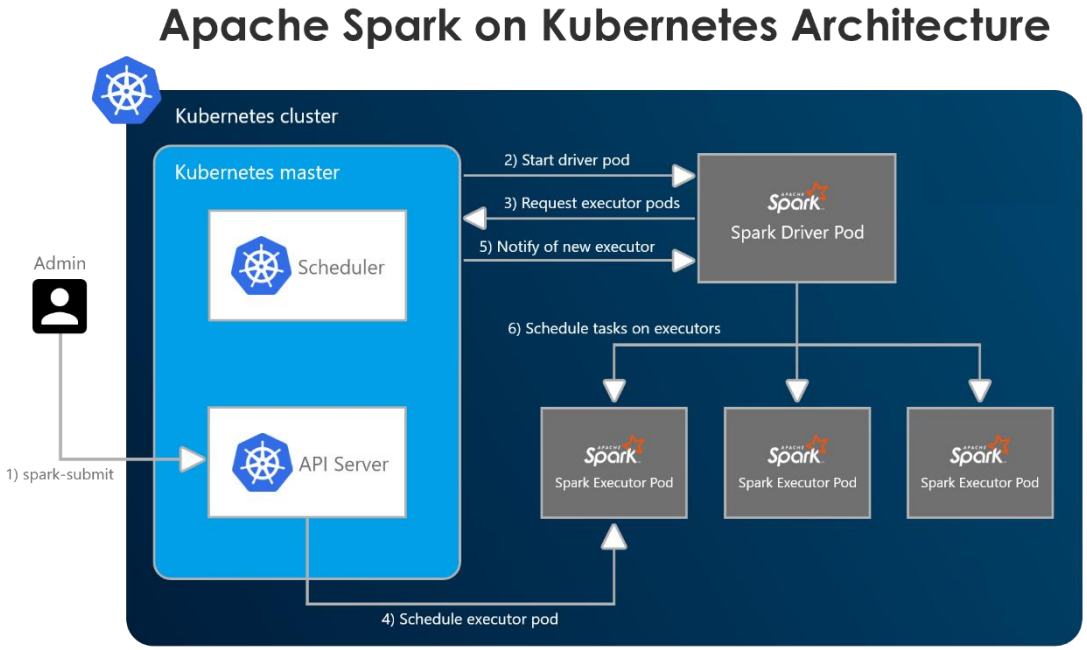


Figura 3.26 – Il flusso di esecuzione di un'applicazione Spark su Kubernetes.

Spark su Kubernetes può eseguire in due modalità che si distinguono per la scelta di dove schedulare il Pod driver:

- Nella *modalità cluster* il cliente che richiede il submit dell'applicazione si trova all'esterno del cluster Spark. Il Pod driver verrà invece schedulato come Pod all'interno del cluster, in uno qualsiasi dei nodi. In questa modalità il cliente può quindi agire in modalità *fire-and-forget*, presentando richiesta di esecuzione dell'applicazione e continuando il proprio flusso esecutivo.
- Nella *modalità client*, invece, il driver viene eseguito come processo co-locato nello stesso luogo in cui risiede il cliente, sia esso un Pod o un nodo fisico.[72]

La modalità client viene prediletta quando il cliente è in esecuzione all'interno del cluster e ha la necessità di mantenere sotto controllo l'esecuzione delle applicazioni Spark che richiede, come nel caso del mio componente "Frontend & ML Analyzer".
Ho deciso quindi di eseguire Spark su Kubernetes in modalità client. Di conseguenza, il mio driver Spark viene eseguito all'interno del Pod my-jupyter-0.

Per poter rendere possibile l'esecuzione in modalità client sono state necessarie due accortezze operative derivanti dalle seguenti osservazioni:

1. *Gli executor Spark devono essere in grado di comunicare con il driver attraverso un hostname stabile e una porta accessibile.*
   A questo fine ho creato un headless-service con cui esporre my-jupyter-0.
   Dopo aver scritto il file YAML descrittore del servizio, ne ho comandato la creazione con il comando:

   ```
   kubectl apply -f jupyter-headless-svc.yaml
   ```

2. *Il Pod my-jupyter-0 all'interno del quale esegue il driver deve avere le autorizzazioni necessarie per poter richiedere all'API server la creazione e lo scheduling dei Pod executor.*
   In Kubernetes esistono delle astrazioni chiamate ServiceAccount con cui è possibile taggare i Pod. Ad un ServiceAccount, poi, è possibile associare dei Role o ClusterRole contenenti una lista di operazioni che è autorizzato a compiere chi possiede quel ruolo. L'associazione Service Account-Role viene eseguita mediante una terza entità, chiamata ClusterRoleBinding.
   Dal momento che il mio Pod my-jupyter-0 risulta taggato con il ServiceAccount `default:default`, ho associato a questo ServiceAccount un `clusterrole=edit` che consenta di modificare gli altri Pod del cluster, e quindi anche di richiederne la creazione di nuovi.
   Il comando che ho eseguito a questo scopo è

   ```
   kubectl create clusterrolebinding spark-role --clusterrole=edit \
             --serviceaccount=default:default --namespace=default
   ```

Compatibilità delle versioni Python tra driver ed executor

Per garantire un funzionamento corretto di Spark è necessario che gli executor presentino la stessa versione di Spark e di Python utilizzate nel driver, che nel mio caso sono Spark.3.1.1 e Python 3.8. A questo scopo ho creato una custom image per gli executor ed il procedimento che ho seguito è stato il seguente:

1. Ho scaricato i file sorgente di Apache Spark 3.1.1 dalla repository ufficiale[38]. Nei sorgente è presente uno script chiamato *docker-image-tool.sh* che permette di fare la `docker build` del Dockerfile utilizzato dagli executor, anch'esso già fornito.

   L'immagine che viene creata a default è predisposta per l'esecuzione di Spark su JVM ma lo script permette di definire dei *binding* a linguaggi aggiuntivi. Nel mio caso ad esempio, utilizzando PySpark, avevo la necessità di installare Python sull'immagine degli executor. Tuttavia, il Dockerfile fornito dai sorgenti Spark installa a default la versione Python 3.7 mentre il driver lanciato dal mio *my-jupyter-0* utilizza Python 3.8.

   Questo causava un errore di incompatibilità di versioni al momento dell'esecuzione di uno Spark Job, impedendone l'esecuzione.

2. Ho dovuto quindi creare un'immagine custom per gli executor andando a modificare il Dockerfile nel seguente modo perché installasse Python 3.8:

```
ARG base_img

FROM $base_img
WORKDIR /

# Reset to root to run installation tasks
USER 0

RUN mkdir ${SPARK_HOME}/python
RUN apt-get update && \
    apt install -y wget build-essential zlib1g-dev libncurses5-dev libgdbm-dev libnss3-dev libssl-dev libsqlite3-dev libreadline-dev
RUN wget https://www.python.org/ftp/python/3.8.2/Python-3.8.2.tgz
RUN tar xzf Python-3.8.2.tgz
WORKDIR /Python-3.8.2
RUN ./configure --enable-optimizations
RUN make install
```

Figura 3.27 – Il Dockerfile da cui creare l'immagine per gli executor Spark. Si noti come venga eseguita l'installazione di Python 3.8.

3. Ho eseguito la `docker build` per creare l'immagine a partire dal Dockerfile personalizzato.

4. Ho caricato l'immagine così creata sulla mia repository nel Docker Hub, in modo da renderla scaricabile dai Pod executor che verranno creati dal "Frontend & ML Analyzer".

[38] https://www.apache.org/dyn/closer.lua/spark/spark-3.1.1/spark-3.1.1-bin-hadoop3.2.tgz (URL consultato il 06 maggio '21).

Feature offerte dal componente

Completate le doverose premesse precedenti, utili a comprendere il funzionamento di questo componente, è ora possibile descrivere le feature che offre.

Collegandosi alla porta `8888` esposta dal Pod my-jupyter-0, il Frontend si presenta nel seguente modo:

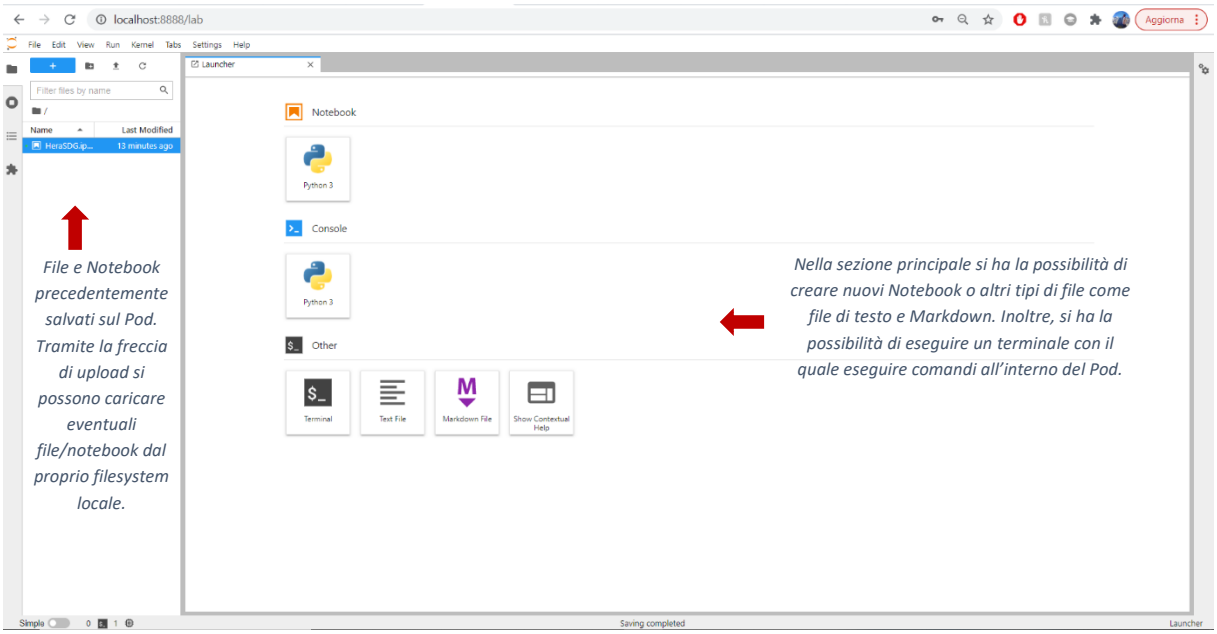


Figura 3.28 – La schermata di avvio del Frontend Jupyter Lab.

Aprendo il Notebook "HeraSDG" che ho realizzato è possibile fruire di sei funzionalità principali. A ciascuna è dedicata una sezione del Notebook in cui è presente sia codice di programmazione che plugin grafici con cui l'utente può interagire. Le feature offerte sono:

1. Creazione dello Spark Context

L'utente può richiedere la creazione di un contesto Spark specificando il numero di executor iniziali e quante risorse (RAM e CPU) dedicare a ciascuno. Il quantitativo di RAM è espresso in Gi mentre quello di CPU in millicores.[39]

[39] Il millicore è un'unità di misura con cui viene indicato $\frac{1}{1000}$ di core di CPU.

Il numero di executor, poi, viene aggiustato dinamicamente sulla base del carico di lavoro. Questo meccanismo è chiamato Dynamic Resource Allocation e verrà descritto in maniera approfondita nel sotto-paragrafo 3.4.4.

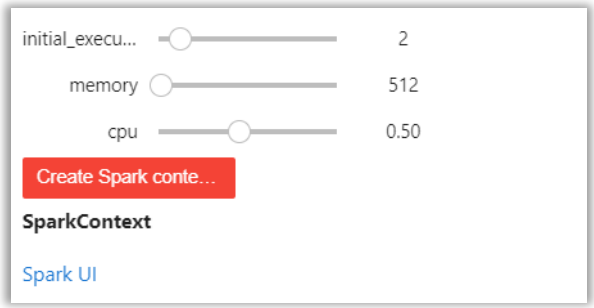


Figura 3.29 – Creazione dello Spark Context.

2. Lettura dei dati

L'utente può richiedere la lettura di tutti i dati utenti, comportamenti e premi presenti sul componente di Data Storage (HDFS) a partire da una certa data. Grazie alla funzione `read.json` offerta dallo Spark Context, i dati vengono letti dall'URL specificato e vengono caricati in tre DataFrame Spark (`utenti_df`, `comportamenti_df` e `premi_df`).

Si noti la scelta di salvare i suddetti DataFrame in cache, in modo da velocizzare le operazioni di processamento future. I benefici del caching vengono mostrati nel sotto-paragrafo 3.4.4.

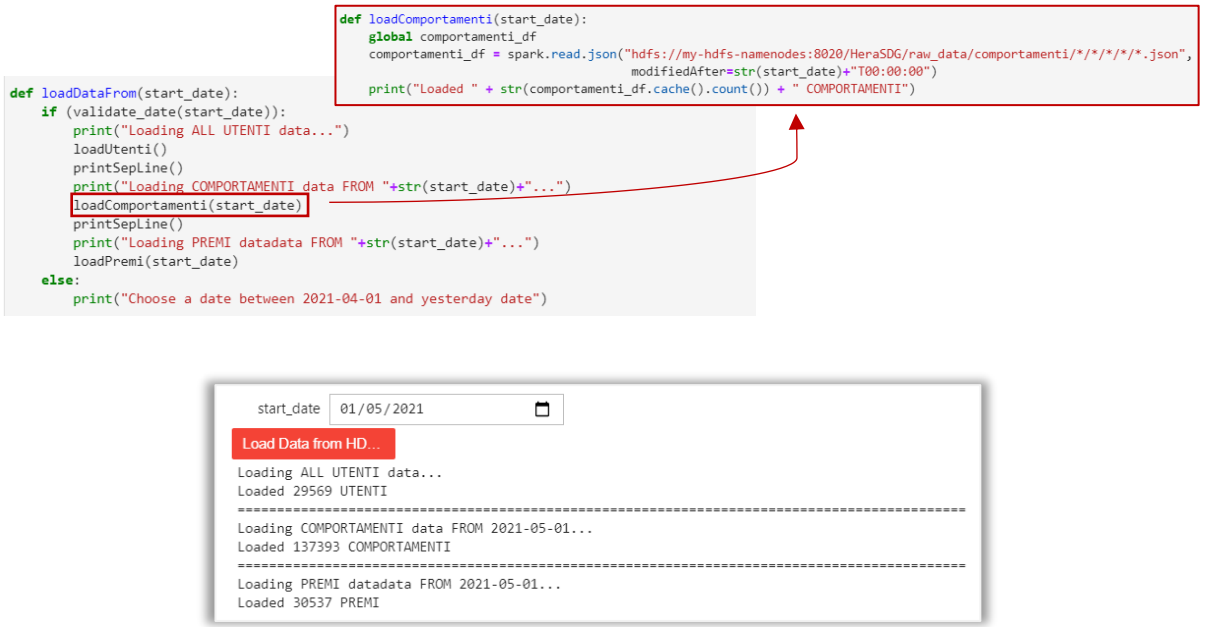


Figura 3.30 – Lettura dei dati da HDFS.

3. Processamento dei dati

I DataFrame Spark `comportamenti_df` e `premi_df` vengono aggregati per `id_utente` compiendo delle `groupBy`. Grazie alla funzione di aggregazione *somma* si ottengono i token tot. guadagnati e i token tot. spesi per ogni utente.

Si tiene traccia anche del numero tot. di comportamenti effettuati e di premi acquistati, suddivisi per partner e utente.

Infine, grazie a dei `join` relazionali si ottiene un DataFrame finale con il seguente formato:

id_utente	Data di Nascita	Eta	Provincia	Sesso	TOT_tk_guadagnati	TOT_tk_spesi	HERA_FIDELITY(%)	CONAD_FIDELITY(%)	CAMST_FIDELITY(%)
CA_11666	1935-03-14	86	RE	M	6.15	0	0.463415	0.146341	0.390244
CA_11978	1983-12-02	37	BO	F	7.15	0	0.643357	0.356643	0.000000
CA_1477	1982-06-03	38	MO	F	12.05	0	0.514523	0.286307	0.199170

Figura 3.31 – Il DataFrame finale ottenuto dal processamento dei dati.

Una volta completato il processamento, è anche possibile richiedere il salvataggio del `final_DataFrame` sul PV del Pod my-jupyter-0.

Figura 3.32 – I button per richiedere il processamento dei dati e il salvataggio del DataFrame finale ottenuto dal processamento.

4. Visualizzazione del DataFrame

Le librerie Spark non offrono supporto alla visualizzazione dei DataFrame.

Tuttavia, è possibile convertire il DataFrame Spark in un DataFrame Pandas e sfruttare l'ampia scelta di librerie di visualizzazione che Python mette a disposizione.

La conversione in un DataFrame Pandas è un'operazione che convoglia nella memoria del driver tutti i dati che sono distribuiti tra i vari executor e, se il DataFrame Spark dovesse essere parecchio esteso, questo potrebbe essere un problema.

Per questo motivo ho aggiunto la possibilità di estrarre un campione di dati dal `final_DataFrame` Spark e di convertire solo quello in un DataFrame Pandas.

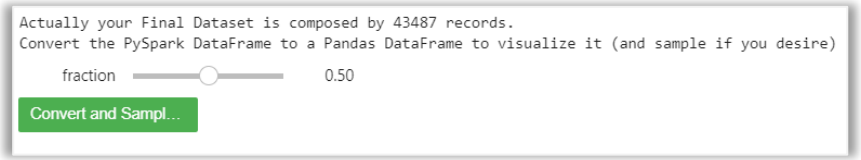


Figura 3.33 – Campionamento del final_DataFrame Spark e conversione del campione in un DataFrame Pandas.

Il DataFrame così ottenuto viene esplorato e visualizzato sfruttando la libreria pandas_profiling e la libreria Bokeh. Un esempio di output prodotto dalla prima libreria è visualizzabile in Figura 3.34. In Figura 3.35 invece è illustrato uno scatter plot ottenuto con la libreria Bokeh.

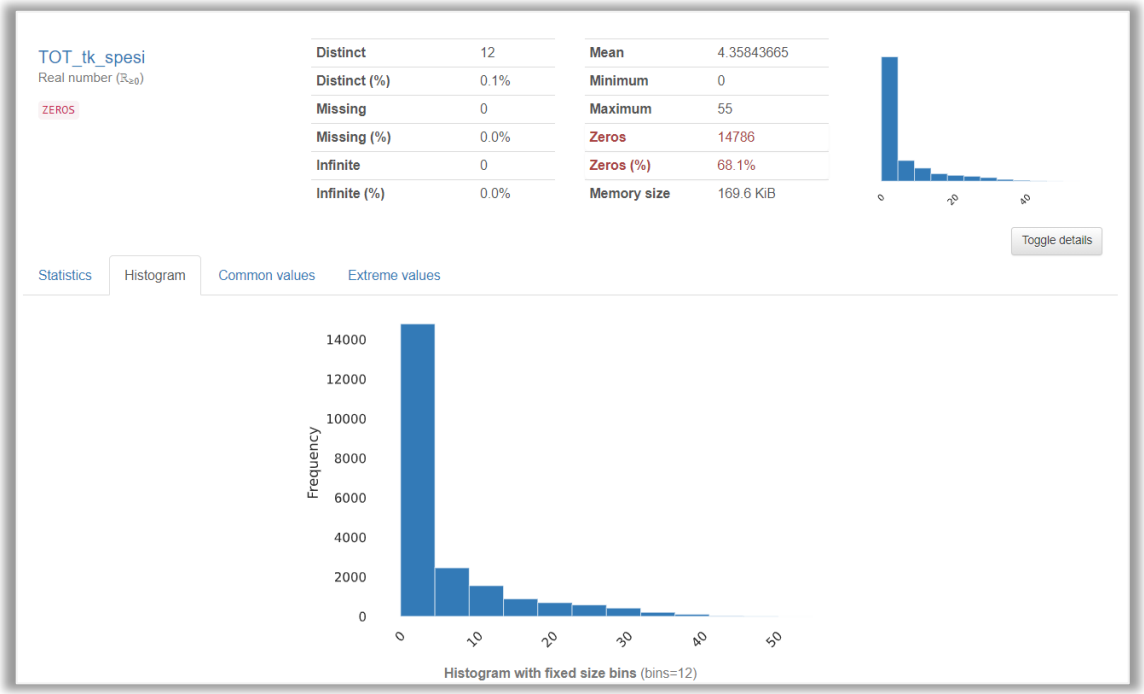


Figura 3.34 – Un esempio di Insight prodotto dalla libreria Pandas_profiling.

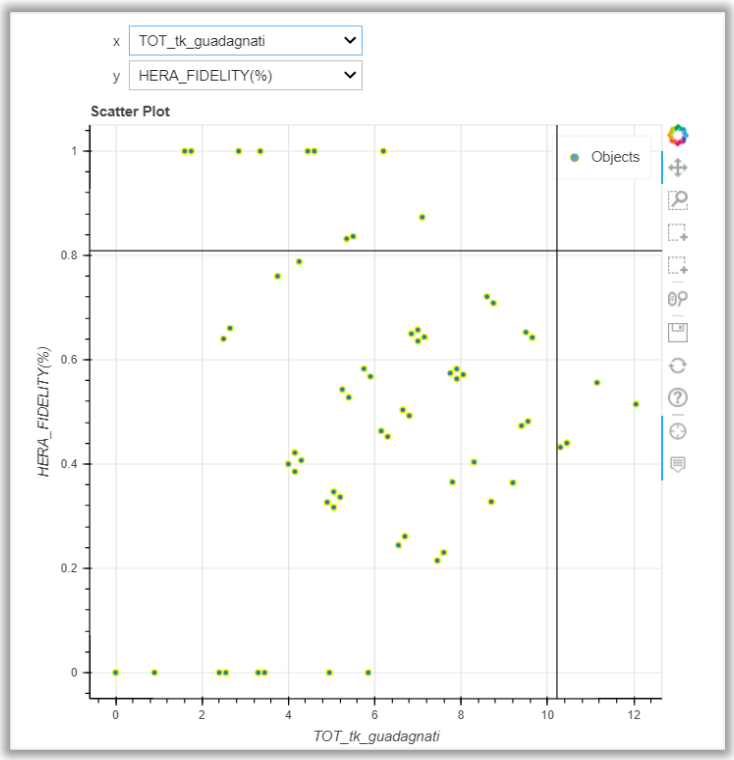


Figura 3.35 – Uno scatter plot prodotto con la libreria Bokeh.

5. ML Clustering

Con lo scopo di condurre una customer segmentation, l'utente può richiedere l'esecuzione di un ML clustering sul `final_DataFrame` sfruttando l'algoritmo K-Means | | della libreria Spark MLlib.

Per esigenze dettate dall'algoritmo, il Final DataFrame viene trasformato con la funzione `vecAssembler.transform(final_data)` accorpando tutte le sue features in un unico vettore. Dopodiché le feature vengono standardizzate usando uno `StandardScaler`. A questo punto, l'utente può richiedere l'esecuzione del clustering.

Per applicare il metodo Elbow descritto in 2.4, il clustering viene ripetuto più volte, incrementando in ciascuna iterazione il numero di centroidi iniziali. Viene quindi graficata la curva delle silhouette calcolate, dalla quale l'utente potrà individuare il numero di centroidi *k* ottimale.

Con il k scelto, l'utente potrà richiedere un clustering definitivo e salvare gli output prodotti (le label di appartenenza record-cluster) in un file sul PV del Pod my-jupyter-0.

Potrà infine visualizzare gli output attraverso scatter plot i cui punti assumono un colore diverso a seconda del cluster di appartenenza.

In Figura 3.36 viene riportato il codice della funzione che realizza il Clustering con l'Elbow Method, mentre in Figura 3.37 un grafico rappresentante l'output finale ottenuto dall'analisi di customer segmentation condotta.

```
def do_Clustering_Elbow():
    global silhouette_score
    silhouette_score = []
    evaluator = ClusteringEvaluator(predictionCol='prediction', featuresCol='standardized', \
                                    metricName='silhouette', distanceMeasure='squaredEuclidean')
    for i in range(2, 10):
        # K-Mean Clustering with "i" centers
        output  = KMeans(featuresCol = 'standardized', k = i).fit(df_kmeans).transform(df_kmeans)
        # Computing the silhouette score
        score   = evaluator.evaluate(output)
        silhouette_score.append(score)
        print("Silhouette Score: ", score)

#     Visualizing the silhouette scores in a plot
    fig, ax = plt.subplots(1, 1, figsize =(8,6))
    ax.plot(range(2, 10), silhouette_score)
    ax.set_xlabel('k')
    ax.set_ylabel('silhouette')

cluster_button = interact_manual(do_Clustering_Elbow)
cluster_button.widget.children[0].description = "Elbow Method"
cluster_button.widget.children[0].button_style = "success"
```

Elbow Method

Figura 3.36 – La funzione che effettua il ML clustering con il metodo Elbow. A questo scopo vengono utilizzati gli oggetti `ClusteringEvaluator` *e* `KMeans` *della libreria Spark MLlib.*

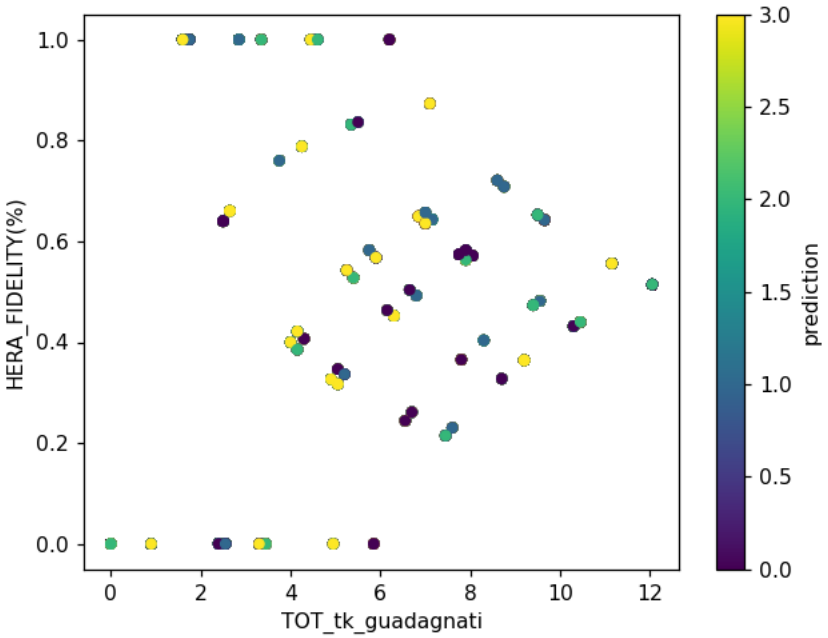


Figura 3.37 – Scatter plot raffigurante l'esito della customer segmentation. I diversi colori dei punti rappresentano l'appartenenza degli stessi a diversi cluster.

6. Clean-up delle risorse

Una volta che l'utente ha terminato la sessione di analisi può stoppare il contesto Spark causando l'eliminazione del driver e di tutti gli executor creati.

```
def stopSpark():
    global spark
    global sc
    spark.stop()
    sc.stop()

stopper = interact_manual(stopSpark)
stopper.widget.children[0].description = "Stop Spark"
stopper.widget.children[0].button_style = "danger"
```

Stop Spark

Figura 3.38 – Clean-up delle risorse.

3.4 Ottimizzazioni e Performance

3.4.1 Pod Affinity e Anti-Affinity

Lo Scheduler Kubernetes, componente del Control Plane descritto in 3.2.1, sceglie il nodo target su cui schedulare un nuovo Pod limitandosi a considerare alcuni fattori quali lo stato di carico attuale del nodo e la quantità di risorse di cui necessità il Pod, se specificate. Questo comportamento è sicuramente ragionevole e può essere sufficiente in molti contesti di sviluppo. In altre situazioni, però, lo sviluppatore potrebbe avere esigenze di deploy di livello applicativo. Ad esempio, potrebbe voler schedulare un Pod su una macchina con certe specifiche hardware, oppure co-locare due Pod con la necessità di comunicare intensamente sullo stesso nodo o su nodi dello stesso rack.
Per venire incontro a queste esigenze, Kubernetes mette a disposizione due metodi che permettono di gestire in maniera avanzata lo scheduling dei Pod:

- **Node selector:** è un metodo di scheduling statico. Nella sezione `spec.nodeSelector` del Pod da schedulare viene specificata una label con la quale designare il nodo target.

pods/pod-nginx.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

Figura 3.39 – Nel campo `spec.nodeSelector` *viene specificata la label* `disktype: ssd`. *Il Pod* `nginx` *verrà quindi eseguito su un nodo che presenti la suddetta label.*

- **Affinity e Anti-Affinity:** questo metodo offre una maggior capacità espressiva del precedente, permettendo di conseguenza un maggior controllo sul deploy.
  Consiste nello specificare delle regole di affinità (o di anti-affinità) sulla base delle quali scegliere (o escludere) il nodo su cui schedulare il Pod.

Le regole sono della forma «questo Pod dovrebbe (o, in caso di anti-affinità, non dovrebbe) essere eseguito in un nodo X se quel nodo X presenta determinate caratteristiche che soddisfino la regola Y oppure se in quel nodo X sta già eseguendo uno o più Pod che soddisfano la regola Z».

Nel caso si utilizzino regole del tipo Y si parla di *Node* Affinity/AntiAffinity mentre per le regole Z di *Pod* Affinity/AntiAffinity. Entrambe vengono scritte sotto forma di espressioni riguardanti delle label.[74]

Nel mio progetto ho utilizzato delle regole di Pod Affinity e Anti-Affinity per eseguire un deploy dei miei Pod che garantisse tolleranza al guasto del singolo nodo, parallelismo ed efficienza nelle letture e scritture.

In particolare, per ottenere i suddetti benefici:

1. I broker Kafka non devono essere co-locati nello stesso nodo.
2. I DataNode HDFS non devono essere co-locati nello stesso nodo.
3. I worker del componente Connector non devono essere co-locati nello stesso nodo. È preferibile, inoltre, che vengano collocati in un nodo in cui esegue almeno uno tra un broker Kafka e un DataNode HDFS.

Per quanto riguarda lo scheduling dei Pod executor Spark, invece, attualmente è possibile controllarlo solo mediante `nodeSelector`, in quanto, come affermato sulla documentazione ufficiale, «sarà possibile usare tecniche di scheduling più avanzate quali Node e Pod Affinity in una futura release».[72]

In Figura 3.40 sono riportate le regole di Pod Anti-Affinity che ho applicato ai Pod su cui eseguono i miei broker Kafka. Similmente ho agito anche per gli altri due componenti, adattando opportunamente le label al contesto.

```
130   # ensure the scheduler does not co-locate replicas (brokers) on a single node.
131   affinity:
132     podAntiAffinity:
133       preferredDuringSchedulingIgnoredDuringExecution:
134       - weight: 1
135         podAffinityTerm:
136           labelSelector:
137             matchExpressions:
138             - key: app.kubernetes.io/component
139               operator: In
140               values:
141               - kafka
142           topologyKey: "kubernetes.io/hostname"
```

Figura 3.40 – Le regole di Pod Anti-Affinity dei Pod kafka broker.

La regola scritta in Figura 3.40 chiede di non co-locare due Pod aventi la stessa label `app.kubernetes.io/component: kafka` all'interno del dominio definito da `topologyKey`.

In questo caso il `topologyKey` è settato a `kubernetes.io/hostname`, ossia la chiave di una label presente in ogni nodo e il cui valore è il nome (univoco) del nodo[40]. Il dominio di allocazione che viene definito, quindi, è pari a un singolo nodo.

Si noti come usando `preferredDuringSchedulingIgnoredDuringExecution` la regola assume una forma di preferenza che viene espressa allo Scheduler Kubernetes. In questo modo lo scheduling dei Pod avviene anche se non è possibile soddisfare quanto richiesto. Con il termine `required`, invece, se non si verificano le condizioni desiderate il Pod non schedulato su nessun nodo, comportamento che nel mio progetto non sarebbe sensato.

In Figura 3.41 viene mostrato come il deploy dei componenti sia stato effettivamente condizionato dalle suddette regole, mentre in Figura 3.42 viene illustrata la situazione di deploy finale.

```
$ kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP             NODE
my-hdfs-datanode-0                  1/1     Running   32         33h   10.244.1.15    piazza-2
my-hdfs-datanode-1                  1/1     Running   23         33h   10.244.0.15    piazza-1
my-hdfs-namenode-0                  1/1     Running   50         33h   10.244.1.14    piazza-2
my-hdfs-shell-84768f96cc-zlvkh      1/1     Running   0          33h   10.244.1.13    piazza-2
my-kafka-0                          1/1     Running   0          13m   10.244.0.19    piazza-1
my-kafka-1                          1/1     Running   0          13m   10.244.1.22    piazza-2
my-kafka-connect-69488b47bd-tn2jl   1/1     Running   0          12m   10.244.1.23    piazza-2
```

Figura 3.41 – I DataNode HDFS e i Broker Kafka sono stati locati su due nodi diversi del cluster.
Il Pod my-kafka-connect è stato collocato su un nodo in cui è presente sia un broker Kafka che un DataNode HDFS.

[40] I miei due nodi presentano rispettivamente le label:
- kubernetes.io/hostname: piazza-1
- kubernetes.io/hostname: piazza-2

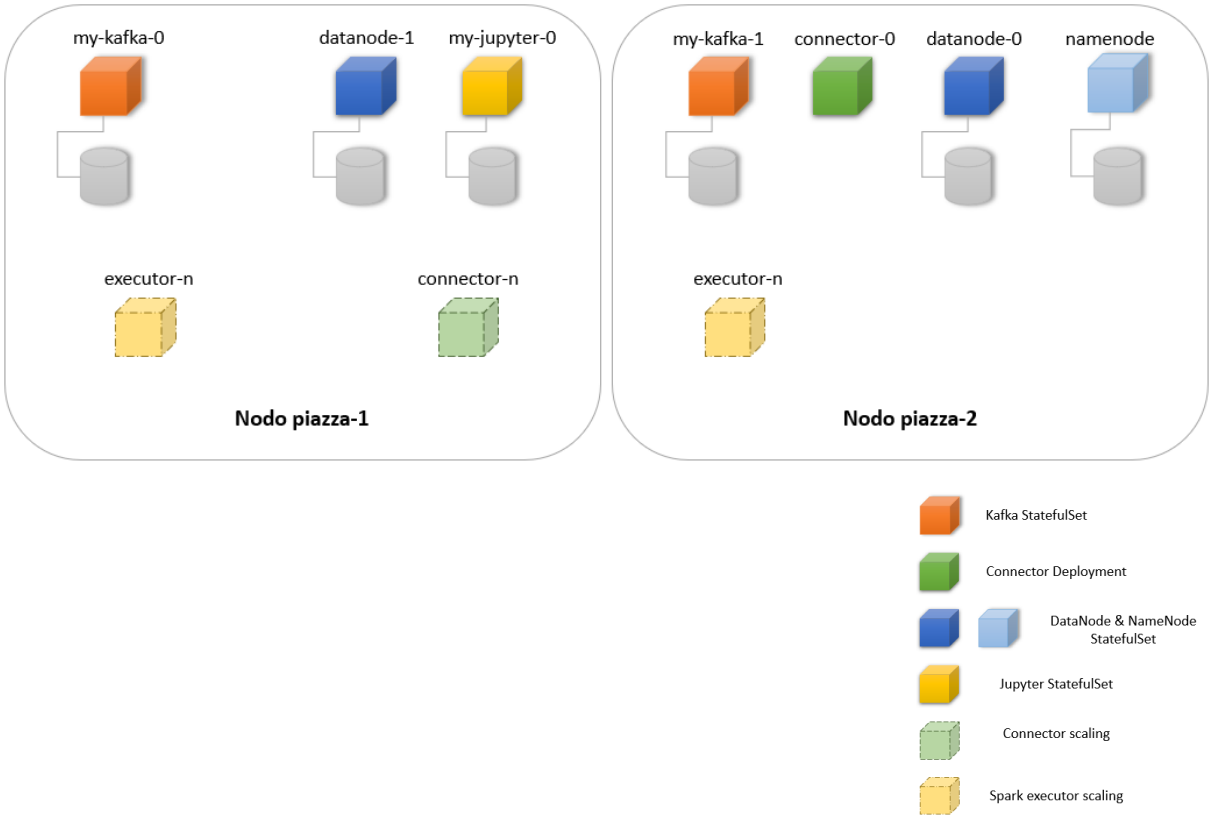


Figura 3.42 – La situazione finale ottenuta dal deploy dei componenti della pipeline di Big Data Analytics

3.4.2 Scaling del Connector & Pre-processor

Per garantire uno scaling automatico del componente "Connector & Pre-processor" ho utilizzato un Horizontal Pod Autoscaler (HPA).

L'Horizontal Pod Autoscaler è una risorsa Kubernetes in grado attuare in maniera automatica uno scaling (up & down) del numero di Pod appartenenti ad un Pod Controller basandosi sul loro utilizzo di risorse.

Per poter funzionare correttamente, l'HPA richiede che sul cluster Kubernetes sia installato un Metrics Server, un componente che monitora i vari Pod e tiene traccia del loro utilizzo di CPU e RAM[41]. L'HPA, infatti, contatta ad intervalli regolari il Metrics Server con lo scopo di recuperare le metriche relative ai Pod di proprio interesse. A default, lo scambio di informazioni avviene ogni 15 secondi ma l'intervallo può essere configurato con il flag `--horizontal-pod-autoscaler-sync-period`.[75]

Una volta ricevute le metriche dei singoli Pod, l'HPA ne calcola il valor medio, lo confronta con un valore soglia e prende eventuali decisioni di scale-up o scale-down.

L'HPA può essere creato tramite file YAML, come tutte le risorse Kubernetes, oppure da CLI con il comando `kubectl autoscale`.

Nel mio progetto ho creato un HPA sfruttando la seconda opzione:

```
kubectl autoscale deployment my-kafka-connect --min=1 --max=3 --cpu-percent=80
```

dove:

- `deployment my-kafka-connect` designa il Deployment che si desidera gestire. Nel mio caso il Deployment del componente "Connector & Pre-processor".
- `--min=1 --max=3` sono rispettivamente il limite minimo e massimo di Pod tra i quali lo scaling può oscillare.
- `--cpu-percent=80` indica che la risorsa da monitorare è la CPU e che la soglia di utilizzo è l'80%. Il valore percentuale si riferisce alla quantità `spec.resources.request.cpu` specificata nel file YAML del Deployment.

Con riferimento al file descrittore del Deployment Connector, il cui contenuto è riportato in 3.3.4, avevo infatti specificato il seguente campo:

[41] CPU e RAM sono le metriche disponibili a default. È possibile configurare anche delle altre metriche sulla base delle proprie necessità.

```
        resources:
          requests:
            memory: "256Mi"
            cpu: “200m”
```

Grazie ad esso avevo indicato allo Scheduler che i Pod del Deployment Connector necessitavano un utilizzo di almeno 200m di CPU.

Il mio HPA, quindi, comanderà uno scale-up ogni qualvolta l’utilizzo medio di CPU da parte dei Pod del componente Connector, calcolato in percentuale rispetto al `resources.request.cpu` specificato, superi il valore soglia dell’80%. Se dovesse poi tornare al di sotto del valore soglia verrà comandato uno scale-down.
Lo scaling avverrà secondo la formula: [75]

$$desiredReplicas = ceil\,[currentReplicas * (currentMetricValue\,/\,desiredMetric\,Value)]$$

Per esempio, se il *currentMetricValue* ottenuto dal MetricServer dovesse essere il 160% di `resources.request.cpu`, e il *desiredMetricValue* è 80%, il numero di repliche verrà raddoppiato, dal momento che $\frac{160}{80} = 2$.

A questo punto della trattazione è possibile riportare un esempio utile a fissare i concetti fin qui enunciati e a mostrare il funzionamento dell’HPA che ho creato.

In Figura 3.43 si vede come il Pod my-kakfa-connect stia consumando 21m di CPU, ossia il 10% del `resources.request.cpu=200m` che avevo specificato. Dal momento che questo valore è inferiore alla soglia target 80%, l’HPA non comanda alcuna azione.

```
Lorenzo@DESKTOP-QAC8QT9 MINGW64 ~/Documents/LORI/UNIVERSITA'/Magistrale/Tesi/HeraSDG-BigDataPipeline
$ kubectl top pods my-kafka-connect-cdc56bf8-wtvzw
NAME                                 CPU(cores)   MEMORY(bytes)
my-kafka-connect-cdc56bf8-wtvzw      21m          1307Mi

Lorenzo@DESKTOP-QAC8QT9 MINGW64 ~/Documents/LORI/UNIVERSITA'/Magistrale/Tesi/HeraSDG-BigDataPipeline
$ kubectl get hpa
NAME              REFERENCE                         TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
my-kafka-connect  Deployment/my-kafka-connect       10%/80%   1         3         1          72s
```

Figura 3.43 – Il Pod gestito dall’HPA presenta un consumo di CPU sottosoglia.

Per incrementare l’utilizzo di CPU da parte del Pod Connector, quindi, ho avviato il componente Datasource, in modo che iniziasse a pubblicare record su Kafka (Figura 3.44).

```
$ python DataSource/data_source.py 137.204.57.224:30001 10000

...Connecting to bootstrap_server on 137.204.57.224:30001

Initial connection established
==============================

Generate and send a first set of 10000 users and 100000 behaviours...
```

Figura 3.44 – Avvio del Datasource per aumentare il carico sul Pod my-kafka-connect.

Passati alcuni secondi, è possibile notare (Figura 3.45) come l'utilizzo di CPU da parte del Pod my-kafka-connect sia incrementato al 149%, ampiamente sopra il valore soglia.
Secondo la formula di scaling sopra presentata, l'HPA richiederà il raddoppio del numero di repliche attuali. Vediamo quindi che viene creato un secondo Pod my-kafka-connect.

```
Lorenzo@DESKTOP-QAC8QT9 MINGW64 ~/Documents/LORI/UNIVERSITA'/Magistrale/Tesi/HeraSDG-BigDataPipeline
$ kubectl get hpa
NAME               REFERENCE                         TARGETS     MINPODS   MAXPODS   REPLICAS   AGE
my-kafka-connect   Deployment/my-kafka-connect       149%/80%    1         3         1          8m11s

Lorenzo@DESKTOP-QAC8QT9 MINGW64 ~/Documents/LORI/UNIVERSITA'/Magistrale/Tesi/HeraSDG-BigDataPipeline
$ kubectl get pods
NAME                                READY   STATUS      RESTARTS   AGE
kafkacat-debugger                   1/1     Running     0          6d2h
my-hdfs-datanode-0                  1/1     Running     32         7d12h
my-hdfs-datanode-1                  1/1     Running     31         7d12h
my-hdfs-namenode-0                  1/1     Running     52         7d12h
my-hdfs-shell-84768f96cc-zlvkh      1/1     Running     0          7d12h
my-jupyter-jupyter-0                1/1     Running     0          3d10h
my-kafka-0                          1/1     Running     0          63m
my-kafka-1                          1/1     Running     1          63m
my-kafka-connect-cdc56bf8-lrvdz     0/1     Init:0/1    0          18s
my-kafka-connect-cdc56bf8-wtvzw     1/1     Running     0          19m
```

Figura 3.45 – L'utilizzo di CPU al 149% scatena la creazione di un secondo my-kafka-connect

A questo punto il componente Connector & Pre-processor sarà composto da due worker in esecuzione in modalità distribuita.
Secondo quanto descritto in 2.2.4, i worker sono in grado di coordinarsi per ribilanciare i task ogniqualvolta un nuovo worker si aggiunge o abbandona il cluster.
Per verificarlo ho sfruttato l'interfaccia REST che ogni worker espone. Mi sono collegato all'interno di uno dei due Pod my-kafka-connect e ho eseguito il comando:

```
curl localhost:8083/connectors/hdfs3-sink/status
```

ottenendo il seguente output:


```
"tasks": [
    {
        "id": 0,
        "state": "RUNNING",
        "worker_id": "10.244.0.36:8083"
    },
    {
        "id": 1,
        "state": "RUNNING",
        "worker_id": "10.244.0.36:8083"
    },
    {
        "id": 2,
        "state": "RUNNING",
        "worker_id": "10.244.0.36:8083"
    },
    {
        "id": 3,
        "state": "RUNNING",
        "worker_id": "10.244.1.47:8083"
    },
    {
        "id": 4,
        "state": "RUNNING",
        "worker_id": "10.244.1.47:8083"
    },
    {
        "id": 5,
        "state": "RUNNING",
        "worker_id": "10.244.1.47:8083"
    }
```


Figura 3.46 – Stato dei task e loro distribuzione tra i diversi connect worker presenti.

Dalla Figura 3.46 si può apprezzare come i task siano stati effettivamente equi partiti tra i due connect worker presenti. I due indirizzi IP indicati, infatti, sono gli indirizzi appartenenti ai due Pod my-kafka-connect, come è possibile verificare in Figura 3.47.

```
Lorenzo@DESKTOP-QAC8QT9 MINGW64 ~/Documents/LORI/UNIVERSITA'/Magistrale/Tesi/HeraSDG-BigDataPip
$ kubectl get pods -o wide -l app.kubernetes.io/component=connector
NAME                                READY   STATUS    RESTARTS   AGE     IP            NODE
my-kafka-connect-cdc56bf8-lrvdz     1/1     Running   0          5m57s   10.244.0.36   piazza-1
my-kafka-connect-cdc56bf8-wtvzw     1/1     Running   0          24m     10.244.1.47   piazza-2
```

Figura 3.47 – Indirizzi IP dei due Pod my-kafka-connect.

3.4.3 Misurazioni e performance del Frontend & ML Analyzer

Dynamic Resource Allocation

L'esecuzione di Spark su Kubernetes offre un meccanismo di allocazione dinamica delle risorse. Questo significa che un'applicazione Spark, se eseguita su Kubernetes, ha la possibilità di richiedere a runtime la creazione di ulteriori executor quando il carico di lavoro incrementa, e di richiederne l'eliminazione qualora il carico dovesse diminuire.

In particolare, l'allocazione di nuovi executor utilizza come indicatore di carico il numero di task Spark in stato di *pending*, cioè non ancora assegnati a nessun executor.

Le richieste di allocazione avvengono in modo incrementale e sono distribuite su più round:[73] qualora vi sia almeno un task *pending* per un periodo di tempo maggiore o uguale a `spark.dynamicAllocation.schedulerBacklogTimeout` secondi, Spark assumerà che il numero di executor attuale non è sufficiente a gestire il carico di lavoro richiesto e avvierà la prima richiesta di risorse chiedendo l'istanziazione di 1 executor.

Se ancora la coda di task *pending* dovesse persistere, dopo altri `spark.dynamicAllocation.sustainedSchedulerBacklogTimeout` secondi Spark richiederà la creazione di 2 ulteriori executor, di 4 al terzo round, di 8 al quarto e così via.[73]

Seguendo un andamento esponenziale, la politica appena descritta si traduce in una richiesta di risorse cauta e prudente in una fase iniziale, adattandosi bene a quelle situazioni in cui sono sufficienti solo pochi executor in più. Allo stesso tempo però, è in grado di raggiungere rapidamente l'allocazione di elevate quantità di risorse qualora fossero necessari molti executor.

Per quanto riguarda lo scale down degli executor, invece, la policy adottata è più semplice ed è la seguente: «l'applicazione Spark rimuove un executor quando rimane inattivo per più di `spark.dynamicAllocation.executorIdleTimeout` secondi. Qualora l'executor avesse dei contenuti salvati in cache, invece, si conteggiano `spark.dynamicAllocation.cachedExecutorIdleTimeout` secondi».[73]

La Dynamic Resource Allocation si è rivelata particolarmente utile per il mio progetto in quanto il componente "Frontend & ML Analyzer" è pensato per eseguire le operazioni di processamento e analisi dei dati *on-demand*, solo quando richiesto dal Business Manager.

Per utilizzare in maniera efficiente le risorse del cluster e ottimizzare i costi energetici, ho abilitato questa feature nel mio componente e l'ho configurata come mostrato in Figura 3.48:

```
################ DYNAMIC ALLOCATION ################
conf.set("spark.dynamicAllocation.enabled", "true")
conf.set("spark.dynamicAllocation.shuffleTracking.enabled", "true")
# remove policy
conf.set("spark.dynamicAllocation.executorIdleTimeout", "1800s")
conf.set("spark.dynamicAllocation.cachedExecutorIdleTimeout", "3600s")
# request policy:
conf.set("spark.dynamicAllocation.initialExecutors", initial_executor)
conf.set("spark.dynamicAllocation.minExecutors", 1)
conf.set("spark.dynamicAllocation.maxExecutors", 4)
conf.set("spark.dynamicAllocation.schedulerBacklogTimeout", "1s")
conf.set("spark.dynamicAllocation.sustainedSchedulerBacklogTimeout", "1s")

spark = SparkSession \
    .builder \
    .appName("Processor") \
    .config(conf = conf) \
    .getOrCreate()

# Create a context from the specified session
sc = spark.sparkContext.getOrCreate()
display(sc)
```

Figura 3.48 – Codice di configurazione della Dynamic Resource Allocation.

Con `spark.dynamicAllocation.enabled=true` e `spark.dynamicAllocation.shuffleTracking.enabled=true` si abilita la feature.

Ho settato il parametro `spark.dynamicAllocation.executorIdleTimeout=1800s` in modo che gli executor vengano rimossi dopo 30 min in cui si trovano in stato *idle* e non hanno alcun contenuto in cache. Qualora presentassero dei dati salvati in cache, `spark.dynamicAllocation.cachedExecutorIdleTimeout=3600s` specifica che verranno eliminati dopo 60 min di inattività.

Ho settato `spark.dynamicAllocation.minExecutors=1` con l'intento di avere almeno un executor sempre pronto a servire le richieste di computazioni che possono arrivare. `spark.dynamicAllocation.maxExecutors=4`, invece, l'ho impostato considerando la quantità di risorse disponibili nel cluster DISI.

Tempo di avvio degli executor

I Pod executor vengono creati in tempi abbastanza brevi.

In Figura 3.49, ottenuta dall'output del comando `kubectl get pods`, si può infatti notare come 5 Pod executor, la cui istanziazione è stata richiesta in simultanea alla creazione dello Spark Context, impieghino un tempo $t \leq 4$s per passare in stato di Running ed essere pronti a ricevere istruzioni dal driver.

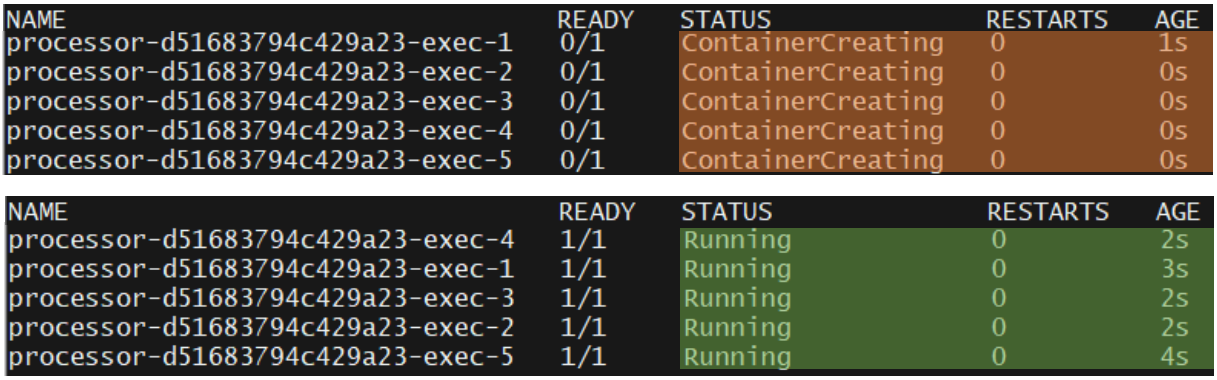


Figura 3.49 –Tempo di avvio degli executor Spark.

È doveroso sottolineare, però, che al momento dell'esecuzione di questo test entrambi i nodi piazza-1 e piazza-2 disponevano in cache dell'immagine Docker utilizzata dagli executor. Questa, infatti, viene scaricata solo la prima volta che un executor viene schedulato su un certo nodo. Dopodiché, viene salvata in cache e resa disponibile agli altri executor che in futuro dovranno eseguire su quel nodo.

Spark caching

Ho eseguito un semplice test per dimostrare quanto l'utilizzo della cache velocizzi le operazioni Spark.

In particolare, ho eseguito per due volte consecutive la funzione `loadDataFrom` che legge e conteggia i dati presenti sul Data Storage.

La suddetta funzione, il cui codice è riportato in Figura 3.50, viene tradotta in sei Job Spark, tre dei quali sono dedicati a leggere i record utenti, comportamenti e premi da HDFS e i tre restanti a contare il numero dei record letti.

Figura 3.50 – Codice della funzione `loadDataFrom`

Come si può notare dal codice in Figura, i DataFrame ottenuti con la funzione `spark.read.json` vengono memorizzati in cache.

Più precisamente, ciascun executor salverà nella propria cache le partizioni del DataFrame che ha letto. In questo modo i task che in futuro dovranno utilizzare quei dati non saranno obbligati a ricalcolarseli (a rileggerli da HDFS).

Rieseguendo quindi la stessa funzione `loadDataFrom` per una seconda volta, è possibile usufruire dei benefici del caching e ottenere un'esecuzione molto più rapida.

Con l'ausilio della Spark UI, accessibile alla porta `4040` del driver (Pod my-jupyter-0), ho tenuto traccia dei tempi impiegati dalle due esecuzioni.

Specifiche del test:

- Lettura da HDFS e conteggio di:
  - ~ 30'000 record utenti.
  - ~ 140'000 record comportamenti.
  - ~ 31'000 premi.
- RAM per ciascun executor: 512Gi.
- CPU per ciascun executor: 100m.

Esecuzione della funzione `loadDataFrom` per la prima volta - **NO caching** ✖

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total
5	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/05/08 09:12:03	1 s	2/2
4	json at NativeMethodAccessorImpl.java:0 json at NativeMethodAccessorImpl.java:0	2021/05/08 09:12:01	1 s	1/1
3	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/05/08 09:11:58	3 s	2/2
2	json at NativeMethodAccessorImpl.java:0 json at NativeMethodAccessorImpl.java:0	2021/05/08 09:11:55	2 s	1/1
1	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/05/08 09:11:49	6 s	2/2
0	json at NativeMethodAccessorImpl.java:0 json at NativeMethodAccessorImpl.java:0	2021/05/08 09:11:35	12 s	1/1

Tempo Tot = 25s

Esecuzione della funzione `loadDataFrom` per la seconda volta – **Caching abilitato** ✓

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total
11	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/05/08 09:16:25	0.4 s	2/2
10	json at NativeMethodAccessorImpl.java:0 json at NativeMethodAccessorImpl.java:0	2021/05/08 09:16:24	0.4 s	1/1
9	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/05/08 09:16:23	0.3 s	2/2
8	json at NativeMethodAccessorImpl.java:0 json at NativeMethodAccessorImpl.java:0	2021/05/08 09:16:22	1 s	1/1
7	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/05/08 09:16:21	0.5 s	2/2
6	json at NativeMethodAccessorImpl.java:0 json at NativeMethodAccessorImpl.java:0	2021/05/08 09:16:20	0.6 s	1/1

Tempo Tot = 3.2s

Figura 3.51 Risultati del test osservati con la Spark UI.

Confrontando i risultati ottenuti si può notare come l'utilizzo della cache comporti una riduzione consistente del tempo totale di esecuzione della funzione. Questo, infatti, passa dai 25s totali del primo running ai 3.2s del secondo. Inoltre, si osserva una riduzione fino a 20x del tempo di esecuzione del singolo Job.

ML Clustering - 1 executor vs 5 executor

In questo test ho messo a confronto le performance del ML clustering (funzione `doClusteringElbow` riportata in Figura 3.52) eseguendolo rispettivamente:

- Con un singolo executor Spark e Dynamic Resource Allocation disabilitata.
- Con un singolo executor Spark iniziale e Dynamic Resource Allocation che consentisse lo scaling fino a 5 executor.

```
def do_Clustering_Elbow():
    global silhouette_score
    silhouette_score = []
    evaluator = ClusteringEvaluator(predictionCol='prediction', featuresCol='standardized', \
                                    metricName='silhouette', distanceMeasure='squaredEuclidean')
    for i in range(2, 10):
        # K-Mean Clustering with "i" centers
        output  = KMeans(featuresCol = 'standardized', k = i).fit(df_kmeans).transform(df_kmeans)
        # Computing the silhouette score
        score   = evaluator.evaluate(output)
        silhouette_score.append(score)
        print("Silhouette Score: ", score)

#     Visualizing the silhouette scores in a plot
    fig, ax = plt.subplots(1, 1, figsize =(8,6))
    ax.plot(range(2, 10), silhouette_score)
    ax.set_xlabel('k')
    ax.set_ylabel('silhouette')

cluster_button = interact_manual(do_Clustering_Elbow)
cluster_button.widget.children[0].description = "Elbow Method"
cluster_button.widget.children[0].button_style = "success"
```

Figura 3.52 Codice della funzione `doClusteringElbow`.

Specifiche del test:

- Il clustering è stato condotto con l'algoritmo K-Means | | della libreria Spark MLlib. L'algoritmo viene eseguito per nove volte (Elbow Method con k che varia da 2 a 9).
- Il DataFrame utilizzato risultava composto da ~ 30'000 record e 3 feature.
- RAM per ciascun executor: 512Gi.
- CPU per ciascun executor: 400m.

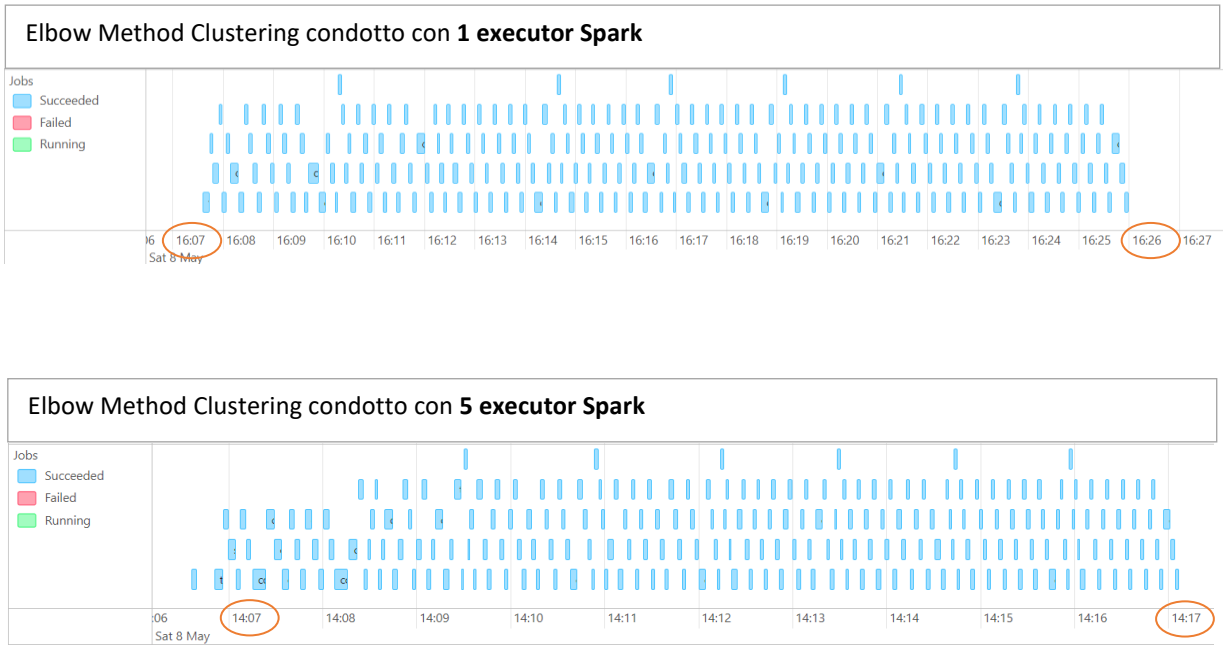


Figura 3.53 Risultati del test osservati con la Spark UI.
I riquadri azzurri rappresentano tutti i Job che sono stati eseguiti per completare la funzione doClusteringElbow.

Confrontando i risultati ottenuti si può notare come l'utilizzo dei cinque executor permetta di ridurre il tempo di esecuzione del clustering. Si passa infatti da un tempo di circa 19 min ottenuto con il singolo executor a un tempo totale di circa 10 min ottenuto con la Dynamic Resource Allocation abilitata.

Conclusioni

L'obiettivo di questa tesi era quello di realizzare e valutare una Pipeline di Big Data Analytics che potesse supportare il progetto Hera SDG nel processo di raccolta, memorizzazione e analisi dei dati prodotti dalla Community.

Il lavoro svolto ha portato alla creazione di un prototipo di pipeline che soddisfa tutta una serie di requisiti centrali per garantire una gestione del dato efficace ed efficiente. Primi fra tutti la *fault tolerance* e la *scalabilità*, che sono stati raggiunti grazie alle tecnologie utilizzate e alle scelte attuate. La pipeline realizzata, infatti, è in grado di tollerare eventuali guasti che possono avvenire ai componenti o ai calcolatori su cui questi eseguono e di adattarsi dinamicamente al carico. I servizi offerti dalla pipeline, quindi, risultano essere altamente disponibili e i dati raccolti, potranno essere memorizzati in modo affidabile senza il rischio di perderli.

Un'altra caratteristica peculiare della pipeline, tenuta in considerazione sin dalle primissime fasi di design, è rappresentata dall'*adattabilità*. Ad oggi, infatti, il progetto Hera SDG è ancora in fase di sviluppo e di conseguenza non sono ancora stati definiti in modo stabile tutta una serie di dettagli molto importanti per la progettazione della pipeline. Per questo motivo è stato fatto uno sforzo atto a realizzare uno strumento il più generalista possibile, pronto cioè, ad adeguarsi a cambiamenti ed eventuali estensioni future.

La pipeline realizzata necessita di un deploy condotto su un cluster di macchine. Grazie all'utilizzo di Kubernetes, si potrà scegliere sia di utilizzare un cluster cloud che un datacenter on-premise, a seconda delle esigenze.

Le performance del prototipo sono state testate in alcuni suoi componenti chiave mostrando risultati discreti. Tuttavia, è doveroso evidenziare come i test siano stati condotti solo in un ambiente con una disponibilità di risorse limitata e, anche a causa delle scadenze incalzanti, i test sono stati effettuati solo su un volume di dati ridotto.

Prima che la pipeline realizzata possa essere definita production-ready è indispensabile condurre ulteriori test e apportare alcune migliorie. Tra queste, risulterà indispensabile integrare la pipeline con uno scaling a livello delle risorse di cluster. Il prototipo realizzato, infatti, è stato predisposto per uno scaling orizzontale dei propri componenti che, perché sia efficace, deve essere accompagnato da uno scaling orizzontale dei nodi e delle risorse del cluster.

Il cluster su cui è stata testata la pipeline presentava un numero fisso di nodi e una quantità fissa di storage ma ad oggi tutti i maggiori cloud provider offrono uno scaling automatico dei nodi e un provisioning dinamico di memoria da cui non si può prescindere.

Ulteriori migliorie future potrebbero prevedere un'estensione del componente di "Frontend & ML Analyzer", integrando ulteriori modelli di Machine Learning ed eventualmente automatizzando alcuni processi decisionali con dei componenti di analisi prescrittiva.

In conclusione, il progetto che ho realizzato in questa tesi rappresenta un ottimo punto di partenza dal quale poter trarre uno strumento production-ready che sia in grado di supportare il progetto Hera SDG e di creare un valore condiviso su tre fronti: economico, sociale ed ambientale.

Bibliografia

[1] United Nations, “Sustainable Development Goals”,
URL: https://www.un.org/sustainabledevelopment/sustainable-development-goals/
URL consultato il 25 novembre ’20.

[2] United Nations, “Goal 17: Revitalize the global partnership for sustainable development”,
URL: https://www.un.org/sustainabledevelopment/globalpartnerships/

[3] Gruppo Hera,
URL: https://www.gruppohera.it/
URL consultato il 25 novembre ’20.

[4] wikipedia.org, “Distributed Ledger”,
URL: https://en.wikipedia.org/wiki/Distributed_ledger
URL consultato il 07 dicembre ’20.

[5] wikipedia.org, “Blockchain”,
URL: https://it.wikipedia.org/wiki/Blockchain
URL consultato il 07 dicembre ’20.

[6] ethereum.org, “What is Ethereum?”,
URL: https://ethereum.org/it/what-is-ethereum/
URL consultato il 07 dicembre ’20.

[7] aws.amazon.com, “Cosa sono i microservizi?”,
URL: https://aws.amazon.com/it/microservices/
URL consultato il 14 dicembre ’20.

[8] M. Fowler, “Microservices”,
URL: https://martinfowler.com/articles/microservices.html
URL consultato il 14 dicembre ’20.

[9] D. Frongillo, “Microservizi: introduzione e caratteristiche”,
URL: https://italiancoders.it/microservizi-introduzioni-e-caratteristiche/
URL consultato il 16 dicembre ’20.

[10] docs.microsoft.com, “The API gateway pattern versus the Direct client-to-microservice communication”,
URL: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern
URL consultato il 15 dicembre ’20.

[11] docker.com, “What is a container”,
URL: https://www.docker.com/resources/what-container
URL consultato il 10 dicembre ’20.

[12] docker.com, “Docker Engine”,
URL: https://www.docker.com/products/container-runtime
URL consultato il 10 dicembre ’20.

[13] redhat.com, “Come funzionano i container Docker?”,
URL: https://www.redhat.com/it/topics/containers/what-is-docker
URL consultato il 14 dicembre ’20.

[14] kubernetes.com, “Documentazione di Kubernetes”,
URL: https://kubernetes.io/it/docs/home/
URL consultato il 20 dicembre ’20.

[15] The Apache Software Foundations, “Apache Hadoop”,
URL: https://hadoop.apache.org/
URL consultato il 11 aprile ’21.

[16] The Apache Software Foundations, “HDFS Architecture”,
URL: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html
URL consultato il 07 marzo ’21.

[17] D.S. Kumara, What is Hadoop Distributed File System (HDFS)?
URL: https://medium.com/@dhammikasamankumara/what-is-hadoop-distributed-file-system-hdfs-36a3503f9c60
URL consultato il 14 marzo ’21.

[18] Data Flair Training, “HDFS Tutorial – a complete Hadoop HDFS Overview”,
URL: https://data-flair.training/blogs/hadoop-hdfs-tutorial/
URL consultato il 18 marzo ’21.

[19] D. Fabbri, “Apache Kafka: tutto quello che c’è da sapere”,
URL: https://www.seacom.it/apache-kafka-tutto-quello-che-ce-da-sapere/
URL consultato il 09 aprile ’21.

[20] kafka.apache.org, “Kafka: Introduction”,
URL: https://kafka.apache.org/intro
URL consultato il 09 aprile ’21.

[21] kafka.apache.org, “Documentation – Balancing leadership”,
URL: https://kafka.apache.org/documentation/#basic_ops_leader_balancing
URL consultato il 10 aprile ’21.

[22] kafka.apache.org, “Documentation – Replication,
URL: https://kafka.apache.org/documentation/#replication
URL consultato il 11 aprile ’21.

[23] L. Johanson, “Apache Kafka for beginners - What is Apache Kafka?”,
URL: https://www.cloudkarafka.com/blog/part1-kafka-for-beginners-what-is-apache-kafka.html
URL consultato il 11 aprile ’21.

[24] K. Rawal, “Kafka Broker, Kafka Topic, Consumer and Record Flow in Kafka”
URL: https://kajalrawal.medium.com/kafka-broker-kafka-topic-consumer-and-record-flow-in-kafka-ec55104977b8
URL consultato il 10 aprile ’21.

[25] J. Karee, “How to parallelise Kafka consumers”,
URL: https://medium.com/@jhansireddy007/how-to-parallelise-kafka-consumers-59c8b0bbc37a
URL consultato il 10 aprile ’21.

[26] kafka.apache.org, “Documentation – The consumer”,
URL: https://kafka.apache.org/documentation/#theconsumer
URL consultato il 11 aprile ’21.

[27] kafka.apache.org, “Documentation – Message delivery semantics”,
URL: https://kafka.apache.org/documentation/#semantics
URL consultato il 11 aprile ’21.

[28] Confluent, “Apache Kafka 101: Kafka Connect”,
URL: https://www.youtube.com/watch?v=J6adhl3wEj4&ab_channel=Confluent
URL consultato il 12 aprile ’21.

[29] Confluent, “From Zero to Hero with Kafka Connect by Robin Moffatt”,
URL: https://www.youtube.com/watch?v=Jkcp28ki82k&ab_channel=Devoxx
URL consultato il 12 aprile ’21.

[30] docs.confluent.io, “Kafka Connect Concepts”,
URL: https://docs.confluent.io/platform/current/connect/concepts.html
URL consultato il 12 aprile ’21.

[31] docs.confluent.io, “Kafka Connect Architecture”,
URL: https://docs.confluent.io/platform/current/connect/design.html
URL consultato il 12 aprile ’21.

[32] kafka.apache.org, “Documentation – Kafka Connect”,
URL: http://kafka.apache.org/documentation/#connect
URL consultato il 12 aprile ’21.

[33] aws.amazon.com, “Introduction to Apache Spark”,
URL: https://aws.amazon.com/it/big-data/what-is-spark/
URL consultato il 14 aprile ’21.

[34] K. O’Malley, “Introduction to Apache Spark”,
URL: https://databricks.com/discover/introduction-to-data-analysis-workshop-series/intro-apache-spark
URL consultato il 15 aprile ’21.

[35] spark.apache.org, “Spark Overview”,
URL: http://spark.apache.org/docs/latest/index.html
URL consultato il 15 aprile ’21.

[36] spark.apache.org, “Cluster Mode Overview”,
URL: http://spark.apache.org/docs/latest/cluster-overview.html
URL consultato il 15 aprile ’21.

[37] spark.apache.org, “RDD Programming Guide – Resilient Distributed Datasets”,
URL: http://spark.apache.org/docs/latest/rdd-programming-guide.html#resilient-distributed-datasets-rdds
URL consultato il 16 aprile ’21.

[38] spark.apache.org, “Spark SQL, DataFrames and Datasets Guide”,
URL: http://spark.apache.org/docs/latest/sql-programming-guide.html#spark-sql-dataframes-and-datasets-guide
URL consultato il 16 aprile ’21.

[39] wikipedia.org, “Supervised Learning”,
URL: https://en.wikipedia.org/wiki/Supervised_learning
URL consultato il 19 aprile ’21.

[40] wikipedia.org, “Unsupervised Learning”,
URL: https://en.wikipedia.org/wiki/Unsupervised_learning
URL consultato il 19 aprile ’21.

[41] wikipedia.org, “Curse of Dimensionality”,
URL: https://en.wikipedia.org/wiki/Curse_of_dimensionality
URL consultato il 19 aprile ’21.

[42] A. McCarley, “(Machine) Learning by example: clustering”,
URL: https://medium.com/opex-analytics/machine-learning-by-example-clustering-a4ff01d81d3a
URL consultato il 19 aprile ’21.

[43] di.unito, “Cluster Analysis”,
URL: http://www.di.unito.it/~cena/materiale/GestioneDB/Cluster.pdf
URL consultato il 19 aprile ’21.

[44] A. Sagar, “Customer Segmentation using K-Means clustering”,
URL: https://towardsdatascience.com/customer-segmentation-using-k-means-clustering-d33964f238c3
URL consultato il 20 aprile ’21.

[45] iaml.it, “Introduzione alla Cluster Analysis”,
URL: https://iaml.it/blog/introduzione-cluster-analysis
URL consultato il 20 aprile ’21.

[46] geeksforgeeks.org, “ML | K-Means++ Algorithm”,
URL: https://www.geeksforgeeks.org/ml-k-means-algorithm/
URL consultato il 20 aprile ’21.

[47] wikipedia.org, “Elbow Method (clustering)”,
URL: https://en.wikipedia.org/wiki/Elbow_method_(clustering)
URL consultato il 20 aprile ’21.

[48] B. Wang, J. Yin, Q. Hua, Z. Wu, J. Cao “Parallelizing K-Means based Clustering on Spark”,
URL: https://www.researchgate.net/publication/312487404_Parallelizing_K-Means-Based_Clustering_on_Spark
URL consultato il 20 aprile ’21.

[49] A. Das, “K-Means clustering using PySpark on Big Data”,
URL: https://towardsdatascience.com/k-means-clustering-using-pyspark-on-big-data-6214beacdc8b
URL consultato il 20 aprile ’21.

[50] S. Orlando, “Clustering”,
URL: https://www.dsi.unive.it/~dm/Slides/2_Cluster.pdf
URL consultato il 21 aprile ’21.

[51] M. Manfredi, “Analisi descrittiva, predittiva e prescrittiva: le differenze e quali adottare per i Big Data Analytics”,
URL: https://www.bnova.it/blog/bnext/analisi-descrittiva-predittiva-prescrittiva/
URL consultato il 25 aprile ’21.

[52] C. Tozzi, “Big Data 101: Dummy’s Guide to Batch vs. Streaming Data”,
URL: https://www.precisely.com/blog/big-data/big-data-101-batch-stream-processing
URL consultato il 25 aprile ’21.

[53] R. Moffatt, “From Zero to Hero with Kafka Connect by Robin Moffatt”,
URL: https://www.youtube.com/watch?v=Jkcp28ki82k&ab_channel=Devoxx
URL consultato il 26 aprile ’21.

[54] kubernetes.io, “Pods”,
URL: https://kubernetes.io/docs/concepts/workloads/pods/
URL consultato il 27 aprile ’21.

[55] kubernetes.io, “Service”,
URL: https://kubernetes.io/docs/concepts/services-networking/service/
URL consultato il 27 aprile ’21.

[56] kubernetes.io, “Workloads”,
URL: https://kubernetes.io/docs/concepts/workloads/
URL consultato il 28 aprile ’21.

[57] kubernetes.io, “StatefulSets”,
URL: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/
URL consultato il 28 aprile ’21.

[58] kubernetes.io, “Volumes”,
URL: https://kubernetes.io/docs/concepts/storage/volumes/
URL consultato il 28 aprile ’21.

[59] kubernetes.io, “Persistent Volumes”,
URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes/
URL consultato il 28 aprile ’21.

[60] cloud.google.com, “Deployments vs. StatefulSets”,
URL: https://cloud.google.com/kubernetes-engine/docs/concepts/persistent-volumes?hl=it#deployments_vs_statefulsets
URL consultato il 28 aprile ’21.

[61] kubernetes.io, “Kubernetes Component”,
URL: https://kubernetes.io/docs/concepts/overview/components/
URL consultato il 28 aprile ’21.

[62] youtube.com, “Kubernetes Tutorial for Beginners [Full Course in 4 Hours]”,
URL:
https://www.youtube.com/watch?v=X48VuDVv0do&t=9487s&ab_channel=TechWorldwithNana
URL consultato il 27 aprile ’21.

[63] kubernetes.io, “Understanding Kubernetes Objects”,
URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/
URL consultato il 28 aprile ’21.

[64] J. Rowe, “Kubernetes Services simply visually explained”,
URL: https://morioh.com/p/73d523abcce4
URL consultato il 29 aprile ’21.

[65] V. Chemitiganti, “Kubernetes Concepts and Architecture”,
URL: https://platform9.com/blog/kubernetes-enterprise-chapter-2-kubernetes-architecture-concepts/
URL consultato il 27 aprile ’21.

[66] helm.sh, “Helm”,
URL: https://helm.sh/
URL consultato il 29 aprile ’21.

[67] helm.sh, “Helm Architecture”,
URL: https://helm.sh/docs/topics/architecture/
URL consultato il 29 aprile ’21.

[68] helm.sh, “The Chart Template Developer's Guide”,
URL: https://helm.sh/docs/chart_template_guide
URL consultato il 29 aprile ’21.

[69] kafka-python.readthedocs.io, “Kafka Producer”,
URL: https://kafka-python.readthedocs.io/en/master/apidoc/KafkaProducer.html
URL consultato il 04 maggio ’21.

[70] datamechanics.co, “Apache Spark 3.1 Release: Spark on Kubernetes is now Generally Available”,
URL: https://www.datamechanics.co/blog-post/apache-spark-3-1-release-spark-on-kubernetes-is-now-ga
URL consultato il 06 maggio ’21.

[71] J. Yves, “How to guide: Set up, Manage & Monitor Spark on Kubernetes”,
URL: https://towardsdatascience.com/how-to-guide-set-up-manage-monitor-spark-on-kubernetes-with-code-examples-c5364ad3aba2
URL consultato il 06 maggio ’21.

[72] spark.apache.org, “Running Spark on Kubernetes”,
URL: https://spark.apache.org/docs/latest/running-on-kubernetes.html
URL consultato il 06 maggio ’21.

[73] spark.apache.org, “Dynamic Resource Allocation”,
URL: https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation
URL consultato il 06 maggio ’21.

[74] spark.apache.org, “Dynamic Resource Allocation”,
URL: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#node-affinity
URL consultato il 06 maggio ’21.

[75] kubernetes.io, “Horizontal Pod Autoscaler”,
URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
URL consultato il 08 maggio ’21.

[76] Repository GitHub del progetto. Lorenzo Piazza – HeraSDG-BigDataAnalyticsPipeline
URL: https://github.com/LorenzoPiazza/HeraSDG-BigDataAnalyticsPipeline