Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in
Informatica

Elaborato finale

Automazione della pipeline per il calcolo delle Community Health Metrics in Wikipedia

Supervisore

Montresor Alberto

Consonni Cristian

Laureando

Denina Andrea

Anno accademico 2024/2025

Ringraziamenti

Desidero esprimere la mia più sincera gratitudine a tutte le persone che mi hanno supportato e guidato durante il percorso universitario e, in particolare, durante il tirocinio curriculare e la stesura di questa tesi. Un ringraziamento speciale va al mio supervisore, Prof. Alberto Montresor, per la preziosa guida, i consigli e il supporto costante, e al Dott. Cristian Consonni per la disponibilità, la pazienza e l'aiuto offerto lungo tutto il tirocinio. Un pensiero riconoscente va alla mia famiglia, che mi ha permesso di vivere questa esperienza unica, sostenendomi sempre in ogni modo possibile. Desidero ringraziare in modo particolare mia sorella Maddy, che non ha mai fatto mancare la sua presenza nei momenti di difficoltà e di bisogno, e a cui auguro di vivere un percorso universitario altrettanto felice, capace di regalarle gioia e soddisfazioni. Non riuscirò mai a citare tutti gli amici che mi hanno accompagnato in questi anni, ma voglio ricordare quanto sia stato fondamentale il loro sostegno. La vita non è fatta soltanto di studio e lavoro, ma anche (e soprattutto) di momenti di socialità, senza i quali probabilmente sarei impazzito. Non potendoli nominare uno per uno, desidero allora ricordare alcuni luoghi che custodiscono i segni più belli della nostra amicizia: davanti a un impianto di pannelli fotovoltaici, nello studentato, in BUC, al Caimano, in Scaletta, al Melo Mangio, a Povo, a casa di Sarzo, a casa di Lyza, al Monte Calvario, sulle Dolomiti, al Refuel, in piscina da Luca e in giro per l'Europa. Più dei luoghi, però, ciò che davvero conta sono le persone con cui ho vissuto questi momenti, a cui va la mia più sincera gratitudine. Infine, desidero ringraziare i Camosci che, prenotando con anticipo un Airbnb a Trento, mi hanno dato la motivazione necessaria per concludere questa tesi. Un pensiero speciale va anche ai miei coinquilini di Via Tommaso Gar 4 (sia a chi è ancora presente sia a chi è già andato via) con i quali ho condiviso una parte importante della mia vita trentina, sempre ricca di conversazioni stimolanti. È grazie a loro se in casa si respira un'aria serena e allegra. Mi scuso in anticipo se questa sera (16 settembre) farò dei danni alla casa (ovviamente scherzo).

Indice

Sommario **2**

1 Introduzione **3**
1.1 Contesto . . . 3
1.2 Dataset . . . 4
1.3 Problema affrontato . . . 5
1.3.1 Obiettivi del tirocinio . . . 5

2 Tecnologie utilizzate **6**
2.1 Docker . . . 6
2.1.1 Principali componenti . . . 7
2.1.2 Immagini e Dockerfile . . . 7
2.1.3 Container . . . 8
2.1.4 Docker Compose . . . 8
2.1.5 Volumi . . . 8
2.2 Apache Airflow . . . 8
2.2.1 Concetti principali . . . 9
2.2.2 Architettura . . . 11
2.2.3 Perché utilizzare Airflow? . . . 12
2.3 Monitoraggio e metriche . . . 12
2.3.1 StatsD e StatsD Exporter . . . 13
2.3.2 Prometheus . . . 13
2.3.3 Grafana . . . 14

3 Implementazione **15**
3.1 DAG e script . . . 15
3.1.1 Struttura delle task . . . 17
3.1.2 Task create_dbs . . . 18
3.1.3 Task <langcode>_process_dump . . . 19
3.1.4 Task <langcode>_calc_flags . . . 20
3.1.5 Task <langcode>_calc_streaks . . . 20
3.1.6 Task calc_primary_language . . . 21
3.1.7 Task <langcode>_calc_vs . . . 21
3.2 Containerizzazione con Docker . . . 22
3.2.1 Dockerfile . . . 22
3.2.2 docker-compose.yml . . . 23
3.3 Metriche e monitoraggio . . . 25
3.4 Deployment . . . 27

4 Risultati e Conclusioni **28**
4.1 Risultati Ottenuti . . . 28
4.2 Competenze Acquisite e Contributo Personale . . . 30
4.3 Sviluppi Futuri e Considerazioni Conclusive . . . 30

Bibliografia **30**

Sommario

L'elaborato descrive l'attività svolta durante il tirocinio curriculare, finalizzato alla progettazione e implementazione di una pipeline dati automatizzata per il calcolo delle Community Health Metrics di Wikipedia. Queste metriche costituiscono indicatori fondamentali per valutare lo stato di salute e la vitalità delle comunità online, fornendo uno strumento utile sia per i ricercatori che per i responsabili delle piattaforme collaborative. Il lavoro prende avvio da uno script Python monolitico, privo di modularità, di strumenti di schedulazione e di funzionalità di monitoraggio. Tale soluzione risultava poco scalabile e difficilmente manutenibile, soprattutto in un contesto in cui i dati da elaborare, i MediaWiki History Dumps, sono caratterizzati da dimensioni elevate e aggiornamenti periodici.

La principale motivazione del progetto è stata quindi quella di adottare un approccio più moderno e strutturato, capace di garantire affidabilità, osservabilità e riproducibilità delle elaborazioni. Per raggiungere questo obiettivo è stato scelto di utilizzare Apache Airflow come strumento di orchestrazione dei workflow, che ha consentito di suddividere il processo in task modulari e indipendenti e di programmarne l'esecuzione periodica. Tutta l'infrastruttura è stata containerizzata con Docker e orchestrata tramite Docker Compose, in modo da garantire portabilità, isolamento e semplicità di deploy. Per quanto riguarda l'osservabilità, è stato predisposto uno stack di monitoraggio basato su StatsD, Prometheus e Grafana, che permette di raccogliere e visualizzare metriche relative alle performance e allo stato della pipeline, rendendo possibile l'individuazione di anomalie in tempo reale.

Il risultato finale è una pipeline completamente automatizzata, in grado di elaborare i dump storici di Wikipedia in maniera affidabile e scalabile. Oltre al miglioramento dell'efficienza, il sistema offre ora un controllo puntuale delle esecuzioni, con dashboard dedicate che consentono di analizzare sia l'andamento delle task sia l'utilizzo delle risorse. Il contributo personale del lavoro si è concentrato sul refactoring del codice esistente, sulla definizione e implementazione del DAG di Airflow, sulla configurazione dell'infrastruttura containerizzata e sull'integrazione del sistema di monitoraggio. Il mio contributo si è concluso con il deploy dell'applicazione su un server di Wikimedia, dove è ora attiva in produzione. In questo modo il progetto Community Health Metrics può contare su un'infrastruttura moderna e affidabile, capace di calcolare automaticamente i dati necessari alle visualizzazioni.

1 Introduzione

Questa tesi è il risultato del tirocinio curriculare che ho svolto negli scorsi mesi, nell'ambito del progetto Community Health Metrics di Wikipedia. Tale iniziativa ha l'obiettivo di misurare lo stato di salute delle comunità di Wikipedia attraverso sei metriche, fornendo strumenti utili per analizzare la partecipazione degli editor e supportarne lo sviluppo nel tempo.

Il punto di partenza di questo lavoro era costituito da uno script monolitico che, pur permettendo di calcolare le metriche desiderate, presentava diversi limiti in termini di modularità, scalabilità e assenza di strumenti di monitoraggio.

L'attività di tirocinio si è quindi focalizzata sul refactoring di questo sistema, con l'obiettivo di realizzare una pipeline dati moderna e automatizzata, basata su Apache Airflow, e integrata con un sistema di monitoraggio fondato su Prometheus e Grafana. Il lavoro si è infine concluso con il deploy in produzione della pipeline su un server Wikimedia, dove è attualmente utilizzata per il calcolo mensile delle metriche.

1.1 Contesto

Il progetto Community Health Metrics propone sei insiemi di indicatori, detti Vital Signs, per valutare crescita e rinnovamento delle comunità di Wikipedia. Tre riguardano l'intera popolazione di active editor (chi crea contenuti): Retention, Stability e Balance; tre riguardano funzioni comunitarie specifiche: Special functions, Administrators e Global participation.

Di seguito presento in dettaglio come vengono misurati questi indicatori e quali sono le metriche associate:

- **Retention**: Misura la capacità di trattenere i nuovi editor (coloro che contribuiscono alle pagine di Wikipedia) nel tempo. L'indicatore base è la retention rate: quota di nuovi editor che effettuano almeno un'altra modifica 60 giorni dopo la prima. È un segnale precoce della qualità dell'onboarding e dell'inclusione dei newcomer.

- **Stability**: Cattura la persistenza degli editor attivi. Si conta il numero di mesi consecutivi di attività per ogni editor e si analizza la distribuzione in sei fasce: 1 mese, 2 mesi di fila, 3-6, 7-12, 13-24, più di 24 mesi. Una comunità "stabile" combina sia nuove presenze sia contributori di lungo periodo.

- **Balance**: Valuta l'equilibrio tra "generazioni" di editor molto attivi. L'indicatore considera numero e percentuale di editor molto attivi per anno e per generazione (definita come un periodo di 5 anni in cui l'editor ha effettuato il primo edit). Un editor "very active" è un utente registrato (non bot) con più di 100 edit al mese nel mainspace. L'obiettivo è che più generazioni contribuiscano alla "spina dorsale" produttiva, senza dipendere solo dai veterani.

- **Special functions**: Questo indicatore prende in considerazione gli editor che svolgono compiti speciali all'interno di Wikipedia, ossia attività che vanno oltre la semplice scrittura di contenuti. In particolare, vengono distinte due categorie: da un lato coloro che si occupano della manutenzione tecnica, contribuendo nei namespace relativi a template e infrastruttura del software; dall'altro chi partecipa alle attività di coordinamento, interagendo negli spazi dedicati all'organizzazione e alla governance della comunità.

- **Administrators**: Gli amministratori svolgono un ruolo centrale nel mantenimento dell'ordine e della qualità all'interno delle comunità di Wikipedia, poiché hanno la responsabilità di supervisionare i contenuti, applicare le regole e garantire il corretto funzionamento del progetto. Per valutare la capacità di questo gruppo di supportare la comunità, l'indicatore osserva sia l'evoluzione storica degli utenti che hanno ricevuto i permessi amministrativi, sia il numero di amministratori effettivamente attivi nel tempo. Un altro aspetto rilevante riguarda il rapporto tra amministratori ed editor

attivi: questo valore fornisce un'indicazione della sostenibilità del gruppo di gestione rispetto alla dimensione complessiva della comunità. In questo modo è possibile comprendere se il numero di amministratori è sufficiente a garantire trasparenza ed efficienza, evitando situazioni di sovraccarico o, al contrario, un eccesso di concentrazione di potere decisionale.

- **Global participation**: Questa metrica misura il grado di apertura e di interconnessione di una comunità linguistica di Wikipedia con il resto del movimento Wikimedia. Da un lato considera il livello di partecipazione degli utenti al progetto Metawiki, osservando in particolare quanti editor attivi hanno come lingua primaria quella della comunità analizzata. Dall'altro analizza la composizione degli editor in base alla loro lingua primaria, distinguendo tra chi contribuisce principalmente nella Wikipedia in questione e chi invece proviene da altre edizioni linguistiche. In questo modo è possibile valutare la capacità della comunità di attrarre contributori multilingue e di favorire interazioni tra diverse wiki, elementi importanti per garantire apertura, collaborazione e scambio con il resto dell'ecosistema Wikimedia.

Le metriche descritte vengono calcolate a partire dai MediaWiki History Dumps, un insieme di file che documentano in maniera completa la cronologia delle modifiche di Wikipedia e che vengono aggiornati mensilmente. Questi dataset rappresentano la base informativa su cui si fonda il progetto Community Health Metrics, in quanto permettono di ricostruire l'evoluzione delle comunità e di derivarne gli indicatori presentati. Nel capitolo successivo verranno presentati nel dettaglio la loro struttura e i dataset utilizzati nella fase di sviluppo della pipeline.

1.2 Dataset

Il dataset dei MediaWiki History Dumps contiene la cronologia completa degli eventi relativi a revisioni, utenti e pagine delle wiki Wikimedia a partire dal 2001. I dati sono organizzati in uno schema denormalizzato, in cui tutte le informazioni sono raccolte in un unico formato, includendo campi precomputati utili alle analisi, come il numero di edit per utente e per pagina o le operazioni di revert. Questo rende i dump una fonte strutturata e coerente per lo studio dell'evoluzione delle comunità.

Il dataset viene aggiornato con cadenza mensile, generalmente entro la fine della prima settimana del mese. Ogni rilascio contiene l'intera cronologia a partire dal 2001 fino al mese corrente: questo approccio è necessario poiché eventi come cambi di nome di utenti, revert o spostamenti di pagine possono modificare retroattivamente lo stato delle tabelle, rendendo poco affidabili eventuali aggiornamenti incrementali.

I dump sono organizzati secondo un sistema di partizionamento che tiene conto della dimensione delle diverse edizioni linguistiche. Per le wiki più grandi, come enwiki, wikidatawiki e commonswiki, i file sono suddivisi su base mensile; per quelle di dimensioni intermedie la suddivisione è annuale; mentre per le comunità più piccole l'intero storico è raccolto in un singolo file. In questo modo si evita la generazione di file eccessivamente voluminosi, mantenendo ogni dump entro la dimensione di circa 2 GB. I dump sono distribuiti in formato TSV, compresso con Bzip2 per ridurre le dimensioni. La struttura delle directory segue una convenzione che comprende la versione del dump (nel formato YYYY-MM, con la disponibilità limitata agli ultimi due mesi), il nome della wiki e l'intervallo temporale della partizione, ad esempio: `/2019-12/ptwiki/2019-12.ptwiki.2018.tsv.bz2`.

Nel contesto del progetto ci interessano esclusivamente i dump relativi alle comunità linguistiche di Wikipedia, poiché costituiscono la base per il calcolo delle metriche presentate. Nella fase di sviluppo ho scelto di lavorare su un sottoinsieme di questi dati, selezionando alcune edizioni linguistiche dei dialetti italiani: piemontese, lombardo, ligure, veneto, sardo, siciliano e napoletano. La scelta è stata guidata dal fatto che tali dataset, oltre ad avere dimensioni ridotte, sono distribuiti in un unico file all-time, caratteristica che li rende particolarmente adatti per le fasi di test e sviluppo.

Per chiarezza, nella tabella seguente riporto le edizioni considerate, con il nome della lingua, il codice corrispondente e la dimensione del file dump in megabyte:

Lingua	Codice	Dimensione (MB)
Piemontese	pmswiki	64.6
Lombardo	lmowiki	91.6
Ligure	lijwiki	18.9
Veneto	vecwiki	78.9
Napoletano	napwiki	45.3
Siciliano	scnwiki	57.4
Sardo	scwiki	14.6

Tabella 1.1: Dump utilizzati nella fase di sviluppo (edizione maggio 2025)

1.3 Problema affrontato

La versione iniziale dell'architettura, disponibile su GitHub, era costituita da un unico script Python, denominato `vital_signs.py`. Questo programma aveva il compito di estrarre i dati dai MediaWiki History Dumps e di popolare due database SQLite. Le operazioni venivano eseguite in maniera sequenziale: in una prima fase lo script generava il database `vital_signs_editors.db`, contenente le informazioni di base sugli editor. Successivamente, a partire da questi dati, calcolava gli indicatori definiti come Vital Signs e li inseriva nel database `vital_signs_web.db`.

Quest'ultimo database rappresentava l'output finale del processo ed era utilizzato da un'applicazione sviluppata con il framework Dash, che permetteva di visualizzare graficamente gli indicatori attraverso una serie di dashboard. Questa applicazione è stata sviluppata da un altro tirocinante ed era stata deployata su un server wikimedia dedicato, dove era accessibile pubblicamente.

L'architettura presentava diversi limiti:

- **Monoliticità**: Tutte le operazioni erano contenute in un unico script, rendendo difficile la manutenzione e l'estensione del codice. Non c'era modularità, né possibilità di riutilizzare le singole parti del processo.
- **Assenza di schedulazione**: Lo script doveva essere eseguito manualmente, senza alcun meccanismo di schedulazione. Questo rendeva impossibile automatizzare l'aggiornamento dei dati, che doveva essere fatto manualmente ogni mese.
- **Mancanza di monitoraggio**: Non c'erano strumenti per monitorare lo stato di esecuzione dello script, né per raccogliere metriche sulle performance. In caso di errori, non era possibile risalire facilmente alla causa.
- **Prestazioni limitate**: L'approccio sequenziale e la mancanza di parallelizzazione rendevano il processo lento, soprattutto con dataset di grandi dimensioni. Non c'era modo di scalare l'elaborazione per gestire volumi crescenti di dati.
- **Difficoltà di deploy**: trattandosi di uno script, il deploy risulta più complicato rispetto ad un approccio basato su container. Non c'era isolamento tra le dipendenze e l'ambiente di esecuzione, rendendo difficile la gestione delle versioni e delle librerie.

1.3.1 Obiettivi del tirocinio

L'obiettivo principale del tirocinio è stato il refactoring dello script esistente e la progettazione di una data pipeline completamente automatizzata utilizzando Apache Airflow. In particolare, il lavoro ha previsto:

- la strutturazione del workflow in task modulari e indipendenti, per garantire chiarezza e riusabilità;
- l'introduzione di meccanismi di schedulazione, così da consentire esecuzioni periodiche e affidabili senza intervento manuale;
- l'integrazione di strumenti di monitoraggio per tracciare lo stato della pipeline e raccogliere metriche di performance;

- lo sviluppo dell'intera infrastruttura in un ambiente Docker, con tutti i componenti (Airflow, database e monitoring stack) containerizzati e orchestrati tramite docker-compose, al fine di semplificare il deploy e garantire portabilità;
- la sostituzione del database SQLite con PostgreSQL, scelta resa necessaria dal fatto che le task in Airflow vengono eseguite in parallelo e quindi richiedono un sistema in grado di gestire operazioni concorrenti in scrittura;
- il deployment della nuova architettura su un server Wikimedia, dove la pipeline è ora attiva in produzione e calcola mensilmente le metriche richieste.

Nei capitoli successivi vengono presentate in dettaglio le tecnologie utilizzate (Capitolo 2), l'implementazione della nuova architettura (Capitolo 3), e infine i risultati ottenuti e le conclusioni del lavoro (Capitolo 4).

2 Tecnologie utilizzate

In questa sezione vengono presentate, con un adeguato livello di dettaglio, le tecnologie utilizzate per la realizzazione della pipeline dati sviluppata durante il tirocinio. Lo stack tecnologico adottato è stato progettato per garantire modularità, scalabilità e osservabilità del sistema, caratteristiche fondamentali in un contesto di automazione e monitoraggio di processi dati periodici. Come si può notare, tutte le tecnologie utilizzate seguono la filosofia open source. Con il termine open source si intende software il cui codice sorgente è pubblico e liberamente accessibile: questo permette a chiunque di utilizzarlo, modificarlo e distribuirlo. Grazie a questa apertura, la collaborazione tra sviluppatori è facilitata e diventa più semplice correggere errori o aggiungere nuove funzionalità.

Le tecnologie possono essere suddivise in tre principali categorie funzionali:

- **Containerizzazione**: Docker e Docker Compose sono stati utilizzati per creare un ambiente di esecuzione isolato, facilmente replicabile e portabile. Tutti i componenti dell'infrastruttura vengono eseguiti come container indipendenti ma coordinati, permettendo una gestione semplificata dell'intero sistema.
- **Orchestrazione**: Apache Airflow è stato scelto come strumento per l'automazione e la schedulazione dei flussi di lavoro. La sua interfaccia intuitiva e il paradigma configuration as code lo rendono adatto a definire e monitorare pipeline complesse.
- **Monitoraggio e raccolta delle metriche**: il sistema di osservabilità è stato implementato utilizzando StatsD Exporter, Prometheus e Grafana. Le metriche prodotte da Airflow vengono raccolte, elaborate e visualizzate in tempo reale tramite dashboard interattive, al fine di garantire il corretto funzionamento del sistema e facilitare l'identificazione di anomalie o colli di bottiglia.

Nei paragrafi seguenti, ogni tecnologia viene descritta singolarmente.

2.1 Docker

Docker è una piattaforma open-source che consente di creare, distribuire ed eseguire applicazioni all'interno di pacchetti detti container. È importante notare che quando si parla di Docker, di solito ci si riferisce a Docker Engine, il software che ospita i container. Docker utilizza la virtualizzazione a livello di sistema operativo (containerizzazione), in particolare utilizza alcune feature del Kernel Linux per garantire l'isolamento e il controllo dei pacchetti.

Tecnicismi Alcuni esempi di strumenti del Kernel che vengono utilizzati sono i cgroup e i namespace. I cgroups (control groups) sono una funzionalità del kernel linux che permette di limitare, monitorare e assegnare le risorse del sistema ai diversi processi in esecuzione. In questo modo, ogni container può essere configurato per utilizzare solo una certa quantità di risorse, evitando che un singolo container possa

consumare tutte le risorse dell'host a discapito degli altri. I namespace, invece, sono un meccanismo che isola la visione che ha un'applicazione delle risorse di sistema. Grazie ai namespace, ogni container vede solo i processi, le interfacce di rete, i punti di mount e le variabili d'ambiente che gli sono stati assegnati, come se fosse l'unico sistema attivo sulla macchina. Questo garantisce un elevato livello di isolamento rispetto agli altri container e al sistema operativo.

Vantaggi della containerizzazione Questo approccio, basato sulla containerizzazione (Figura 2.1), rende i container molto più leggeri (nell'ordine di megabyte, contro i gigabyte di molte VM) [16] e più rapidi da avviare rispetto alle macchine virtuali tradizionali. L'efficienza ottenuta consente di eseguire molti più container sullo stesso hardware rispetto alle VM e di rendere lo sviluppo e il deployment delle applicazioni più agili e flessibili.

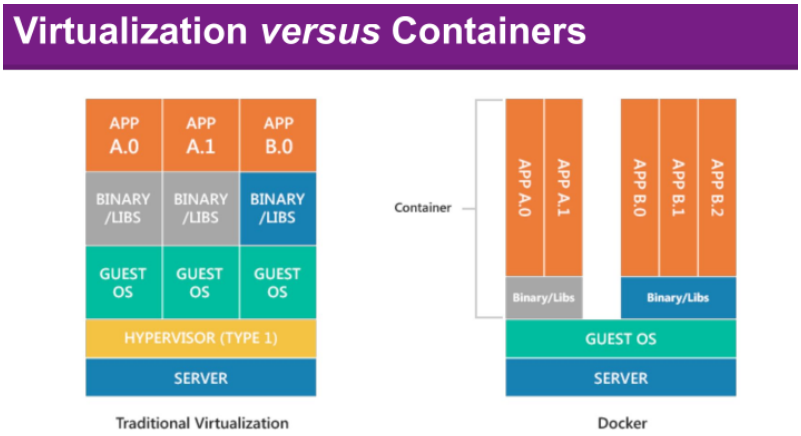

Figura 2.1: Confronto tra le macchine virtuali tradizionali e i container di Docker

2.1.1 Principali componenti

L'ecosistema Docker è composto da tre elementi principali:

- **Software**: Il cuore del sistema è il Docker daemon (dockerd), un processo che gestisce i container e le altre risorse Docker. Gli utenti interagiscono con il daemon tramite il Docker Client, ovvero un'interfaccia a riga di comando (docker) che comunica attraverso le API di Docker Engine.
- **Oggetti**: Gli oggetti Docker rappresentano le entità fondamentali per il funzionamento delle applicazioni containerizzate. I principali sono:
  - **Immagini**: template di sola lettura da cui vengono creati i container.
  - **Container**: ambienti isolati in cui vengono eseguite le applicazioni.
  - **Servizi**: permettono ai container di essere scalati su più Docker daemon creando così uno swarm (un insieme di di daemon cooperante tramite l'API di Docker).

  Gli oggetti di Docker verranno trattati con maggiore dettaglio nelle seguenti sottosezioni.
- **Registry**: I registry sono archivi centralizzati dove vengono conservate le immagini Docker. Possono essere pubblici o privati; il più noto è Docker Hub, il registry predefinito. I registry permettono di caricare e scaricare immagini.

2.1.2 Immagini e Dockerfile

Le applicazioni containerizzate in Docker vengono distribuite sotto forma di immagini Docker, ovvero template immutabili da cui si generano i container. Ogni immagine racchiude il file system e tutte le istruzioni necessarie per avviare l'applicazione all'interno di un container. La creazione di un'immagine è automatizzata tramite un file di configurazione testuale chiamato Dockerfile, in cui vengono specificati in modo dichiarativo tutti i passaggi necessari: dalla scelta dell'immagine di base (ad esempio una distribuzione Linux minimale), alla copia dei file dell'applicazione, fino all'installazione delle dipendenze e

alla configurazione dell'ambiente (cartella di lavoro, utenti, variabili d'ambiente, ecc.). Questo approccio garantisce la riproducibilità e la coerenza dell'ambiente software, eliminando i tipici problemi di compatibilità tra sistemi diversi. Le immagini Docker possono inoltre essere versionate e facilmente condivise tramite i registry, favorendo la collaborazione e la distribuzione delle applicazioni.

2.1.3 Container

Un container Docker è un'istanza attiva di un'immagine: rappresenta un ambiente isolato, leggero e portatile che esegue un'applicazione con tutte le sue dipendenze, come definito nell'immagine da cui è stato creato. Tuttavia, nella pratica, molte applicazioni moderne richiedono l'interazione tra più servizi distinti, ciascuno eseguito in un proprio container. Per gestire e orchestrare in modo semplice insiemi di container che compongono un'applicazione complessa si utilizza Docker Compose, uno strumento che permette di definire e avviare configurazioni multi-container tramite un unico file YAML. Inoltre, poiché i container sono per loro natura effimeri e i dati scritti al loro interno vengono persi alla loro rimozione, Docker mette a disposizione i volumi: uno strumento pensato per garantire la persistenza dei dati e la condivisione delle informazioni tra container diversi o tra host e container.

2.1.4 Docker Compose

Docker Compose è uno strumento che permette di definire, configurare e gestire applicazioni composte da più container. Tramite un file in formato YAML (docker-compose.yml), è possibile descrivere tutti i servizi che compongono l'applicazione, specificando per ciascuno l'immagine da usare, le variabili d'ambiente, le reti, le dipendenze e i volumi associati. Questo approccio consente di avviare, arrestare o aggiornare l'intero stack applicativo con un singolo comando, semplificando la gestione di architetture complesse e migliorando la riproducibilità degli ambienti sia in fase di sviluppo che di produzione. Compose è particolarmente utile quando si desidera far lavorare insieme più servizi (ad esempio un database, un'applicazione backend e un frontend), ciascuno eseguito nel proprio container ma coordinati tra loro.

2.1.5 Volumi

I volumi in Docker rappresentano una soluzione per la persistenza dei dati nei container. Di default, i dati creati all'interno di un container sono temporanei e vengono eliminati quando il container viene rimosso. I volumi permettono di salvare dati in uno spazio dedicato e indipendente dal ciclo di vita dei singoli container, garantendo che le informazioni rimangano disponibili anche dopo la terminazione o la ricreazione dei container. Inoltre, i volumi possono essere utilizzati per condividere dati tra più container o per facilitare l'integrazione con il sistema host. Questa caratteristica è fondamentale per applicazioni che richiedono la conservazione dei dati, come database, sistemi di caching o storage di file.

Esistono due tipi di volumi:

- **Bind Mount**: Consente di collegare una cartella o un file presente sul filesystem dell'host direttamente all'interno di uno o più container. Questo tipo di volume risulta particolarmente utile quando è necessario accedere a file specifici già esistenti o quando si dispone di spazio limitato sul disco gestito da Docker, poiché i dati vengono memorizzati direttamente nel percorso scelto sull'host.

- **Named Volume**: Si tratta di volumi gestiti direttamente da Docker, che vengono creati e organizzati in modo automatico dal Docker Engine all'interno di uno spazio dedicato dell'host. Sono ideali per la persistenza dei dati tra diverse esecuzioni dei container, e permettono una gestione più semplice e indipendente dal percorso fisico sul filesystem.

In generale, i bind mount sono consigliati quando si vuole controllare esattamente dove vengono salvati i dati o quando si ha poco spazio disponibile nell'area gestita da Docker, mentre i named volume offrono maggiore astrazione e portabilità nell'uso dei dati tra container diversi.

2.2 Apache Airflow

Airflow è una piattaforma open-source progettata per la gestione automatizzata dei flussi di lavoro. Consente di definire, pianificare e monitorare in modo programmatico i workflow, tramite una pratica interfaccia utente integrata. Creato originariamente da Airbnb, Airflow è sviluppato in Python ed è progettato secondo il paradigma del Configuration as Code.

Questo approccio prevede la gestione delle configurazioni tramite codice, invece che attraverso modifiche manuali o strumenti proprietari. In pratica, le configurazioni vengono trattate come vero e proprio codice applicativo. Questo metodo offre diversi vantaggi: garantisce coerenza e ripetibilità nelle configurazioni, riduce il rischio di errori umani e semplifica il ripristino in caso di problemi. Inoltre, essendo gestite come codice, le configurazioni possono essere versionate tramite sistemi di controllo versione (come Git), permettendo di tracciare facilmente le modifiche e tornare a versioni precedenti se necessario.

Airflow sfrutta Python per definire i flussi, a differenza di altre piattaforme simili, che tipicamente utilizzano linguaggi markup come XML. Questo approccio permette agli sviluppatori di importare librerie o script esistenti nel loro workflow.

Ogni workflow viene rappresentato come un DAG (Directed Acyclic Graph), ovvero un grafo orientato e aciclico, dove i nodi rappresentano i task (le singole operazioni da eseguire) e le frecce rappresentano le dipendenze e l'ordine di esecuzione tra i task.

2.2.1 Concetti principali

Per comprendere il funzionamento di Apache Airflow, è fondamentale introdurre alcuni concetti chiave che costituiscono la base della piattaforma e ne determinano la logica operativa.

Tra questi i principali sono:

- **DAG**: è la struttura fondamentale che descrive un workflow, ossia una sequenza di operazioni da eseguire secondo una precisa logica di dipendenze. Un DAG definisce quali task devono essere eseguiti e in quale ordine, rappresentando graficamente e logicamente il flusso di lavoro in cui ogni nodo è un task e ogni freccia una dipendenza. Nel codice, un DAG è definito come un oggetto Python in cui si specificano parametri come:
  - ID e descrizione del DAG;
  - `schedule_interval` (frequenza di esecuzione, ad esempio giornaliera o mensile);
  - `start_date` e altri parametri opzionali come ad esempio `retries` (quante volte può essere rieseguita una task);
  - i task che lo compongono e le relative dipendenze (ad esempio, usando `task1 >> task2`).

  La definizione di un DAG in Airflow consente di:
  - Modellare pipeline di qualsiasi complessità,
  - Visualizzare la struttura e lo stato delle esecuzioni tramite l'interfaccia grafica,
  - Gestire in modo chiaro retry, fallback, e dipendenze tra task.

  In seguito verrà mostrato un esempio di DAG:

Listing 2.1: Esempio di DAG in Airflow

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

with DAG(
    dag_id="esempio_dag",
    start_date=datetime(2025, 1, 1),
    schedule_interval="@daily",
    catchup=False
) as dag:
    task1 = PythonOperator(
        task_id="primo_task",
        python_callable=lambda: print("Hello␣Airflow!")
    )
    task2 = PythonOperator(
        task_id="secondo_task",
        python_callable=lambda: print("Secondo␣task!")
    )
    task1 >> task2
```

- **Task**: rappresenta l'unità base di esecuzione: ogni task corrisponde a una singola operazione o processo all'interno di un workflow. Le task vengono organizzate all'interno dei DAG e collegate tra loro tramite dipendenze che ne determinano l'ordine di esecuzione. Esistono due principali tipi di task, nella mia implementazione ho utilizzato solo gli Operator ma li citerò per fornire un maggiore contesto:
  - **Operator**: sono template di task predefiniti, pensati per svolgere operazioni comuni (ad esempio eseguire codice Python, eseguire comandi Bash, inviare email, ecc.). Sono il tipo di task più utilizzato. I principali sono `PythonOperator`, `BashOperator`, `EmailOperator`.
  - **Sensor**: sono una sottoclasse di operator. Sono progettati per attendere il verificarsi di un evento esterno.

  In Airflow, i concetti di Task e Operator sono in parte intercambiabili, ma è comunque utile pensarli come elementi distinti: in pratica, gli Operator e i Sensor rappresentano dei modelli (template), e ogni volta che ne istanzi uno all'interno di un DAG, stai effettivamente creando una Task. Ogni esecuzione di una task (detta Task Instance) attraversa diversi stati durante il proprio ciclo di vita:
  - `none`: la task non è ancora stata messa in coda perché le dipendenze non sono soddisfatte
  - `scheduled`: la task è stata programmata per l'esecuzione, avendo tutte le dipendenze soddisfatte
  - `queued`: la task è in attesa di essere eseguita da un worker
  - `running`: la task è in esecuzione
  - `success`: la task è terminata con successo
  - `failed`: la task è fallita a causa di un errore
  - `skipped`: la task è stata saltata (ad esempio per una branch condizionale)
  - `upstream_failed`: una task a monte è fallita, impedendo l'esecuzione
  - `up_for_retry`: la task è fallita, ma sono previsti tentativi di retry
  - (altri stati specifici come restarting, `up_for_reschedule`, `deferred`, `removed`, etc.)

Come si può vedere in Figura 2.2, l'obiettivo ideale è che una task attraversi lo stato da none a success, passando per `scheduled`, `queued` e `running`. Per impostazione predefinita, una task viene eseguita solo se tutte le sue upstream (le task che la precedono nel flusso logico) sono completate con successo, ma Airflow offre anche strumenti avanzati di controllo del flusso (branching, trigger rules, ecc.).

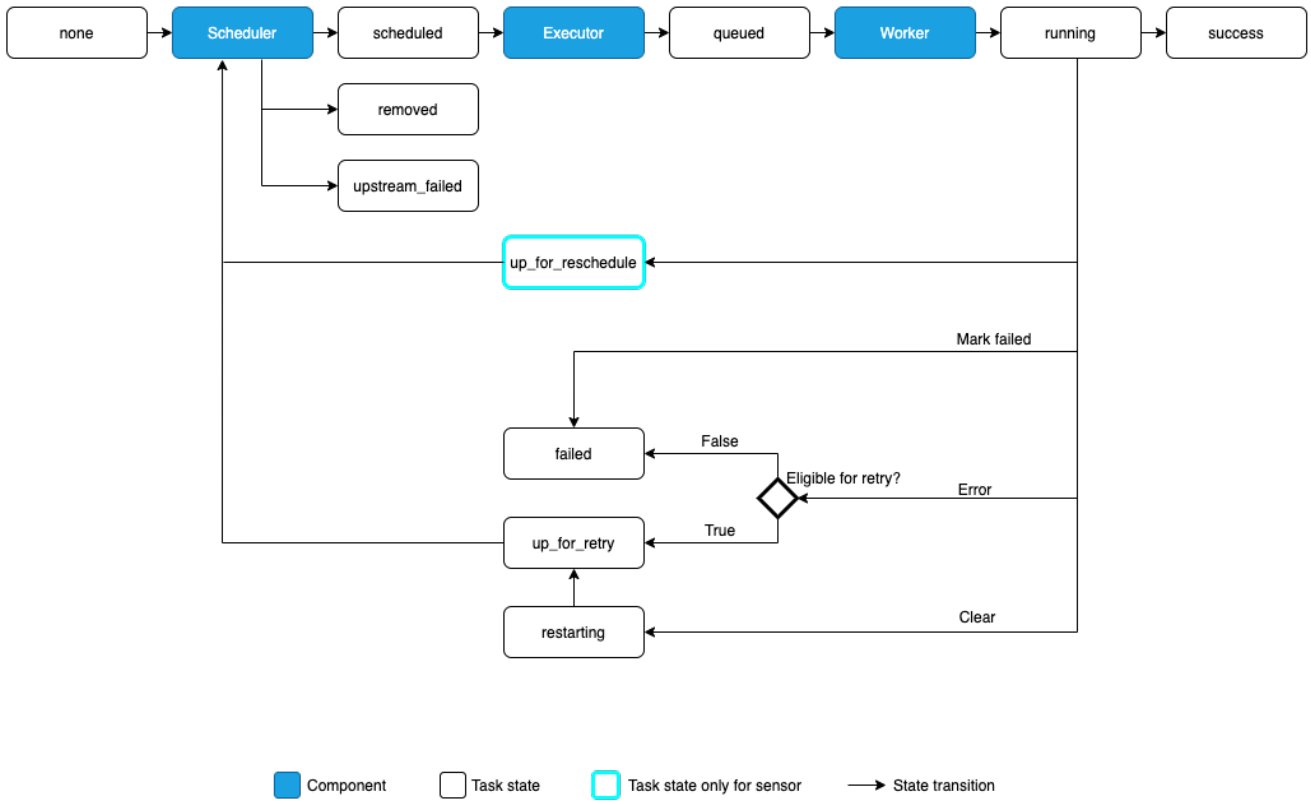

Figura 2.2: Diagramma di tutti i possibili stati di una task

2.2.2 Architettura

L'architettura di Airflow è formata da diversi componenti. Un'installazione minima di Apache Airflow è formata in questo modo:

- **Scheduler**: Il cuore del sistema, si occupa di pianificare l'esecuzione dei workflow (DAG) secondo la programmazione definita, individuando i task che devono essere eseguiti e inviandoli all'esecutore (executor). L'executor, che è una proprietà configurabile dello scheduler, gestisce l'esecuzione dei task e può lavorare in modalità locale o distribuita, a seconda delle esigenze e delle risorse disponibili.

- **Webserver**: Fornisce un'interfaccia web intuitiva che permette di visualizzare, monitorare e gestire i DAG e i task, analizzare i log, forzare l'esecuzione manuale dei workflow e diagnosticare eventuali problemi.

- **Cartella `/dags`**: Una directory locale (o su storage condiviso) in cui risiedono i file Python che definiscono i DAG. Lo scheduler esegue periodicamente la scansione di questa cartella per individuare nuovi workflow o aggiornamenti ai workflow esistenti.

- **Meta Database**: Un database relazionale (tipicamente PostgreSQL o MySQL) utilizzato da tutti i componenti di Airflow per tracciare lo stato delle esecuzioni dei DAG e dei task, memorizzare configurazioni, cronologia delle esecuzioni, log degli errori e altri metadati fondamentali.

In Figura 2.3 viene mostrata la configurazione più semplice di Airflow, in cui tutti i componenti principali (scheduler, webserver e database) vengono eseguiti sulla stessa macchina. In questo scenario si utilizza solitamente il `LocalExecutor`: scheduler e worker condividono lo stesso processo Python e i file dei DAG vengono letti direttamente dal filesystem locale. Anche l'interfaccia web (webserver) viene eseguita sulla stessa macchina. È importante far notare la differenza tra executor e worker: L'executor in Airflow determina la modalità e la logica con cui i task vengono messi in esecuzione, gestendo l'assegnazione dei task ai worker. Il worker è invece il processo che riceve i task dall'executor e li esegue.

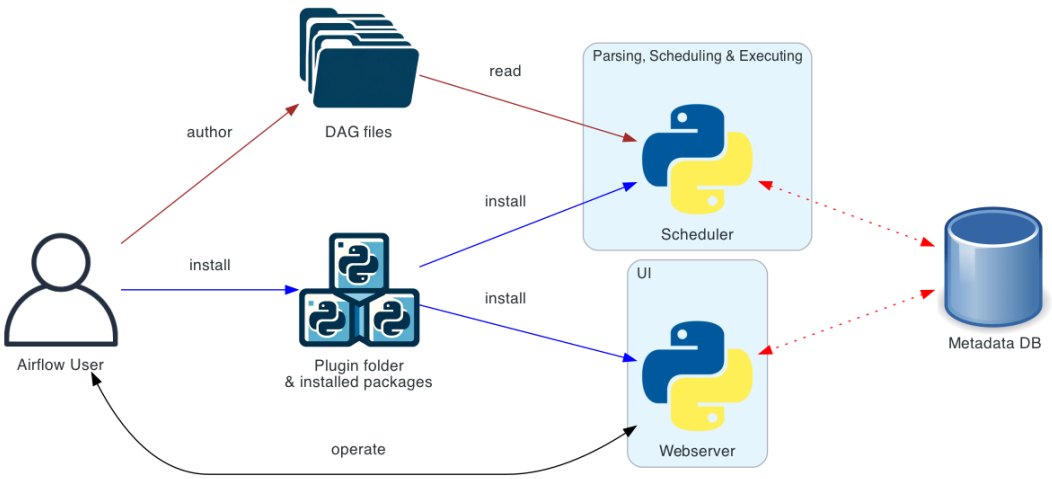

Figura 2.3: Diagramma del deployment minimo di Airflow

2.2.3 Perché utilizzare Airflow?

Apache Airflow offre numerosi vantaggi nella gestione dei workflow. In primo luogo, la flessibilità della piattaforma consente di definire pipeline complesse come codice Python, integrandosi con praticamente qualsiasi tecnologia. Inoltre, l'automazione tramite la schedulazione integrata permette di eseguire i workflow a intervalli prestabiliti, e la gestione delle dipendenze garantisce che i task vengano eseguiti nell'ordine corretto secondo le relazioni definite. Airflow è altamente scalabile, potendo operare da un singolo processo locale fino a cluster distribuiti capaci di gestire carichi di lavoro ingenti. Offre anche un'ottima osservabilità: tramite un'interfaccia web si possono visualizzare chiaramente lo stato dei DAG, i log dei task e altri indicatori, facilitando il monitoraggio e il troubleshooting dei processi. Infine, sono previsti meccanismi di retry automatico e robusta gestione degli errori, così che eventuali fallimenti vengano intercettati e i task possono essere ritentati automaticamente per garantire la continuità delle pipeline.

2.3 Monitoraggio e metriche

Monitorare significa raccogliere, analizzare e visualizzare metriche e log per osservare il comportamento di un sistema nel tempo. Nell'ambito dell'informatica e dell'ingegneria del software, il monitoring si concentra principalmente sul tracciamento di metriche predefinite e indicatori chiave di performance (KPI) con l'obiettivo di rilevare deviazioni dal comportamento atteso. Tipicamente vengono utilizzati strumenti di monitoring per raccogliere dati come utilizzo di CPU, memoria, tempo di risposta e tassi di errore. Questi dati vengono analizzati in tempo reale e, al superamento di determinate soglie, il sistema può inviare degli alert per consentire una risposta tempestiva a possibili problemi.

Nel mio progetto, il monitoring della pipeline dati è stato realizzato tramite l'integrazione di strumenti open source come StatsD, Prometheus e Grafana. Airflow esporta automaticamente metriche relative allo stato delle task, alle risorse utilizzate (CPU/RAM) e alla durata delle esecuzioni, che vengono poi raccolte e visualizzate in dashboard interattive. Questo consente di identificare rapidamente task fallite o anomalie di performance e intervenire in modo reattivo, garantendo affidabilità ed efficienza nella gestione dei workflow automatizzati.

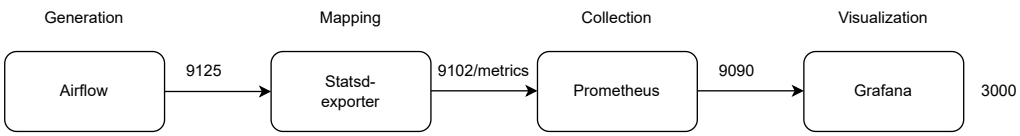

Figura 2.4: Schema logico dei servizi di estrazione metriche e monitoraggio

2.3.1 StatsD e StatsD Exporter

StatsD è un network daemon sviluppato da Etsy, progettato per raccogliere metriche inviate dalle applicazioni tramite protocolli leggeri come UDP o TCP. Ogni metrica è rappresentata dalla sintassi:

```
<nome_metrica>:<valore>|<tipo>
```

dove il nome, espresso in notazione a punti (es. `dag.task.duration`), identifica la metrica, il valore è numerico e il tipo può indicare un contatore (c), un gauge (g), un timer (ms), ecc. Airflow utilizza questo formato per esportare automaticamente informazioni come l'avvio e il completamento delle task, la durata delle esecuzioni e l'uso delle risorse, inviandole in modo asincrono e a basso overhead.

Poiché il formato StatsD non è compatibile con Prometheus, entra in gioco lo StatsD Exporter, un servizio che traduce le metriche ricevute in serie temporali etichettate secondo lo standard Prometheus. La conversione è definita tramite un file di mapping in YAML, che permette di trasformare un nome di metrica come:

```
task.cpu_usage.vital_signs.process_dump:30|g
```

nella forma Prometheus:

```
cpu_usage{dag="vital_signs", task="process_dump"} 30
```

In questo modo le metriche diventano interrogabili con PromQL e possono essere visualizzate in Grafana. Dal punto di vista infrastrutturale, StatsD Exporter è stato configurato come servizio dedicato all'interno del file `docker-compose.yml`, in ascolto sulla porta 9125 per ricevere i pacchetti StatsD da Airflow e sulla porta 9102 per esporre le metriche convertite a Prometheus.

2.3.2 Prometheus

Prometheus è una piattaforma open source progettata per il monitoraggio e l'alerting. È nata in SoundCloud e oggi fa parte della Cloud Native Computing Foundation. Il suo funzionamento si basa su un approccio pull: invece di ricevere metriche dagli applicativi, Prometheus interroga periodicamente gli endpoint degli strumenti monitorati (ad esempio ogni 5 o 10 secondi) e raccoglie le metriche esposte in un formato testuale standard. Queste metriche vengono archiviate come serie temporali, cioè insiemi di valori numerici associati a un timestamp e a un insieme di etichette (labels) che ne descrivono il contesto, ad esempio il nome del task o l'identificativo del DAG in Airflow.

Un aspetto centrale di Prometheus è la distinzione tra i tipi di metrica che le applicazioni possono esportare:

- **Counter**: rappresenta un contatore che cresce monotonamente e può solo aumentare (o essere azzerato al riavvio). È usato ad esempio per il numero di richieste servite, task completate o errori riscontrati.
- **Gauge**: è un valore numerico che può sia aumentare che diminuire. Viene utilizzato per misurazioni istantanee come l'utilizzo della memoria, la temperatura, o il numero di processi attivi in un certo momento.
- **Histogram**: registra osservazioni (come la durata delle richieste o la dimensione delle risposte) suddividendole in bucket configurabili, e produce tre serie temporali: il conteggio cumulativo per bucket, la somma totale e il numero complessivo di osservazioni. È utile per calcolare quantili.
- **Summary**: simile all'histogram, campiona osservazioni e fornisce conteggio e somma, ma calcola direttamente quantili configurabili su una finestra temporale mobile. È adatto per ottenere metriche come la latenza mediana o il 95° percentile delle durate.

Prometheus offre diversi vantaggi:

- conserva le metriche in un database ottimizzato per le serie temporali, permettendo query efficienti;

- utilizza un linguaggio dedicato, PromQL, che consente di analizzare e trasformare i dati (calcolare medie, percentili, confrontare valori tra task o host diversi);
- può generare regole di alerting, ad esempio inviando notifiche quando una metrica supera una certa soglia o quando un servizio non risponde più;
- si integra facilmente con strumenti di visualizzazione come Grafana, che permette di trasformare i dati raccolti in dashboard interattive.

2.3.3 Grafana

Grafana è un'applicazione web open-source per analisi e visualizzazione di metriche, log e tracce da molteplici sorgenti dati. Consente di costruire dashboard interattive e condivisibili tramite un sistema di datasource (es. Prometheus, InfluxDB, Elasticsearch, SQL). Serve a monitorare sistemi e applicazioni trasformando i dati (tipicamente serie temporali) in grafici, tabelle, gauge e heatmap; permette inoltre alerting, esplorazione ad-hoc e condivisione di dashboard.

Il funzionamento di Grafana si articola in alcuni passaggi fondamentali:

- configurazione di una sorgente dati (ad esempio Prometheus);
- creazione di una dashboard, suddivisa in pannelli;
- ogni pannello contiene una o più query verso la sorgente dati e una relativa visualizzazione (grafico, tabella, gauge, ecc.);
- possibilità di utilizzare variabili di dashboard per filtrare i dati dinamicamente (ad esempio per DAG, task o host specifici);
- personalizzazione dei pannelli con legende, soglie e formattazioni per facilitare l'interpretazione.

Grafana ha un'integrazione nativa con Prometheus: una volta aggiunto come data source, Grafana può eseguire query direttamente tramite PromQL e trasformare i risultati in visualizzazioni interattive. Questo legame è uno dei motivi principali per cui Prometheus e Grafana vengono spesso adottati insieme: Prometheus si occupa della raccolta e conservazione delle metriche, mentre Grafana ne gestisce la presentazione e l'esplorazione. Il linguaggio di interrogazione di Prometheus è PromQL (Prometheus Query Language), appositamente concepito per l'analisi delle serie temporali. Permette di estrarre valori istantanei (instant vector) e intervalli temporali (range vector), ed include funzioni come `rate()`, nonché operatori aritmetici e aggregati, utili per calcolare medie, percentili e confronti tra metriche. In Grafana, le query scritte in PromQL alimentano i pannelli, consentendo di trasformare dati grezzi in visualizzazioni significative.

Un esempio di query PromQL potrebbe essere:

```
rate(node_disk_written_bytes_total{job="integrations/macos-node", device!=""}[5m])
```

Questa query calcola la velocità di scrittura su disco in byte al secondo negli ultimi 5 minuti, filtrando per il job specifico e ignorando i dispositivi vuoti. Il risultato può essere visualizzato in un grafico a linee o in un gauge, a seconda delle preferenze dell'utente. La funzione `rate()` in PromQL calcola la velocità media di incremento al secondo di una metrica di tipo counter, valutata su un intervallo temporale specificato (es. ultimi 5 minuti).

Figura 2.5: Esempio di visualizzazione di una query PromQL in Grafana che mostra la velocità di scrittura su disco

3 Implementazione

In questo capitolo viene mostrata l'implementazione della pipeline dati sviluppata durante il tirocinio. Dopo aver descritto il contesto e le tecnologie adottate nei capitoli precedenti, qui l'attenzione si sposta sugli aspetti pratici della realizzazione. L'obiettivo principale è stato quello di trasformare un unico script monolitico in un sistema modulare e automatizzato, basato su Apache Airflow e containerizzato tramite Docker. La nuova architettura non solo consente di programmare e monitorare l'esecuzione dei workflow, ma integra anche strumenti di osservabilità, rendendo possibile analizzare in tempo reale lo stato delle esecuzioni e le risorse utilizzate.

Nel corso del capitolo verranno quindi analizzati:

- la definizione del DAG e delle task che lo compongono, con l'illustrazione delle modifiche apportate agli script originari per adattarli al nuovo contesto;
- la containerizzazione del sistema, descrivendo nel dettaglio il `Dockerfile` e il `docker-compose.yml`;
- l'infrastruttura di monitoraggio, con le configurazioni di StatsD, Prometheus e Grafana;
- il deployment della pipeline su un server di produzione.

In questo modo si vuole mostrare il percorso seguito per passare da una soluzione monolitica e difficile da mantenere a un'infrastruttura completa, affidabile, automatizzata e osservabile.

3.1 DAG e script

Come introdotto nella Sezione 2.2, il Directed Acyclic Graph (DAG) è lo strumento con cui Apache Airflow permette di modellare un workflow come un insieme di task con relazioni di dipendenza.

Nel progetto ho definito il DAG `vital_signs`, contenuto nel file `vital_signs_dag.py` all'interno della cartella `dags/`. Questo DAG ha il compito di processare i dati provenienti dai MediaWiki History Dumps, eseguendo diversi calcoli su di essi e popolando progressivamente due database distinti: prima quello degli editor, che raccoglie le metriche derivate dai dump, e successivamente quello web, in cui vengono calcolati e salvati i vital sign a partire dalle metriche estratte.

Vediamo ora come è stato definito il DAG, le sue caratteristiche principali e come sono stati adattati gli script esistenti per funzionare in questo nuovo contesto.

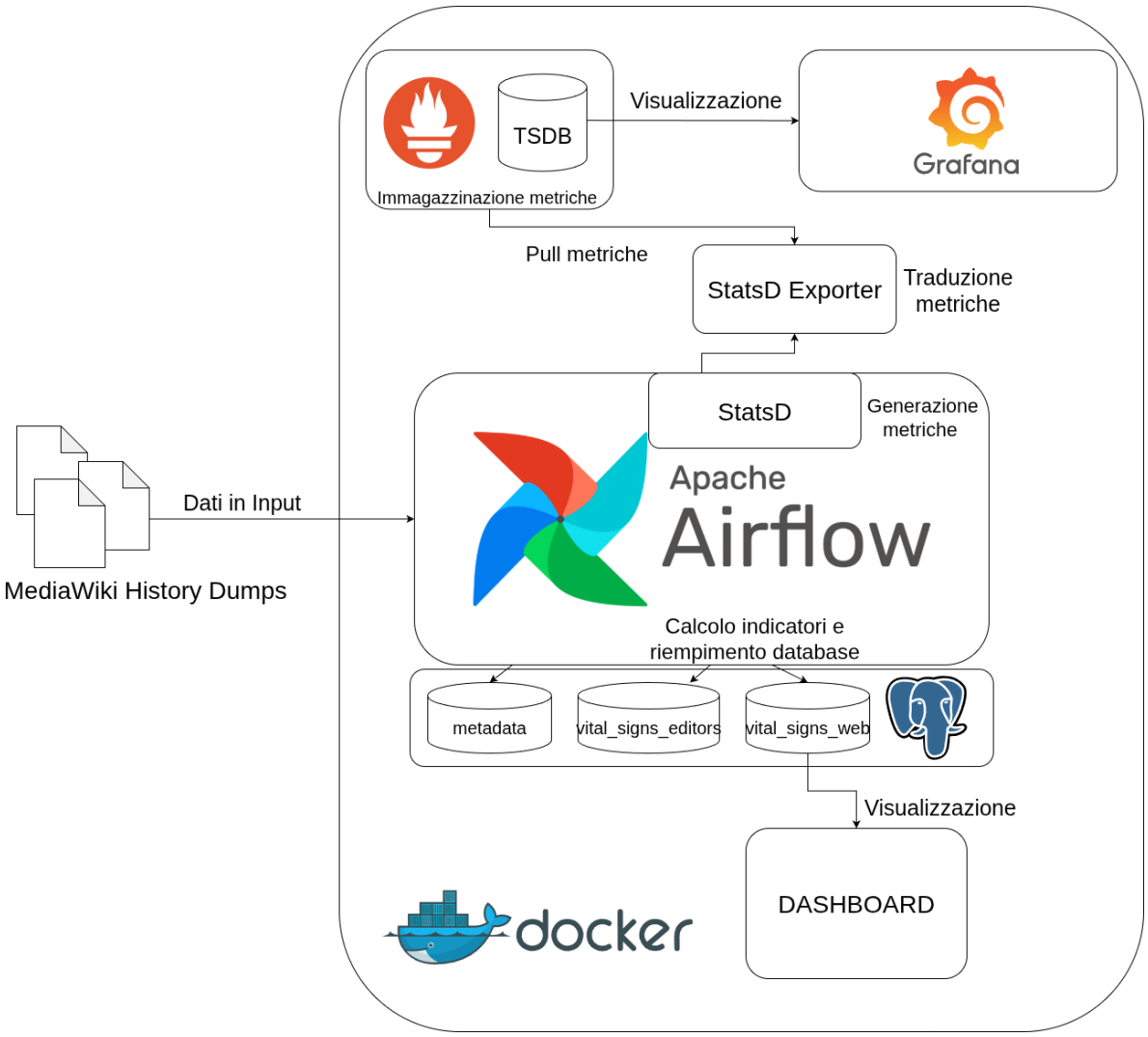

Figura 3.1: Schema logico dell'architettura del sistema

Listing 3.1: Definizione del DAG in Airflow

```
# Import delle librerie necessarie e funzioni di supporto
with DAG(
    dag_id='vital_signs',
    default_args={
        'owner': 'andrea_denina',
        'depends_on_past': False,
        'start_date': datetime(2025, 4, 15),
        'retries': 1,
        'retry_delay': timedelta(minutes=5),
    },
    description='Compute Community Health Metrics (CHM)
    from MediaWiki History dumps for multiple languages',
    schedule_interval='0 0 10 * *',
    catchup=False,
    max_active_runs=1
) as dag:
    # Definizione dei task
```

Il codice in lst. 3.1 mostra la definizione del DAG `vital_signs` e i suoi parametri principali. In particolare:

- `dag_id`: assegna un identificativo univoco al DAG, in questo caso `vital_signs`.
- `default_args`: raccoglie i parametri comuni a tutte le task, come l'owner, la data di inizio, il numero di retry e il ritardo tra un tentativo e l'altro.
- `start_date`: definisce la data a partire dalla quale Airflow può schedulare le esecuzioni del DAG.

- `schedule_interval`: stabilisce la frequenza di esecuzione. È stata scelta un'espressione cron che fa partire il DAG ogni 10 del mese, in linea con le tempistiche di pubblicazione dei dump.
- `catchup`: impostato a `False`, evita che al primo avvio il DAG tenti di recuperare retroattivamente tutte le esecuzioni mancate dal `start_date`.
- `max_active_runs`: limita il numero di esecuzioni contemporanee del DAG a uno, per prevenire conflitti nell'accesso ai dati.

3.1.1 Struttura delle task

Nel DAG `vital_signs` le task sono tutte implementate come `PythonOperator`, quindi ciascuna invoca uno script Python specifico. La pipeline è delimitata da due sentinelle logiche, `start` ed `end` (entrambi `EmptyOperator`), fra le quali si sviluppa il flusso di elaborazione. Subito dopo l'avvio, la task `create_dbs` prepara l'ambiente applicativo creando e inizializzando le tabelle dei due database coinvolti (editor e web). A questo punto il DAG si ramifica per lingua (in base a `wikilanguagecodes`) e per ogni edizione linguistica vengono eseguiti in sequenza tre step: `<code>_process_dump` per estrarre e caricare le metriche dai MediaWiki History Dumps nel DB degli editor, `<code>_calc_flags` per calcolare i flag/profili degli editor, `<code>_calc_streaks` per derivare le streak di attività.

Terminata la fase per lingua sul database `vital_signs_editors`, una fase cross-wiki (`calc_primary_language`) calcola la lingua primaria degli utenti a partire dalle metriche calcolate nelle precedenti fasi. Infine, avviene una seconda ramificazione per lingua dove, con `code_calc_vs`, si computano i vital sign e si popolano le tabelle del database `vital_signs_web` a partire dalle metriche già consolidate. L'intera catena è vincolata da dipendenze esplicite (`>>`) che assicurano l'ordine corretto; inoltre, ogni task utilizza callback di logging di esito (`on_success_callback`, `on_failure_callback`) per tracciare in modo uniforme completamenti e fallimenti delle esecuzioni. In questo modo il DAG rimane leggibile e modulare: ogni task incapsula un passaggio ben definito e l'orchestrazione di Airflow garantisce una progressione deterministica dall'ingestione dei dump fino alla produzione dei vital sign.

Listing 3.2: Esempio di definizione di una task e dipendenze del DAG

```
start = EmptyOperator(task_id='start', dag=dag)
end = EmptyOperator(task_id='end', dag=dag)

create_dbs_task = PythonOperator(
    task_id="create_dbs",
    python_callable=create_db,
    dag=dag,
    op_args=[wikilanguagecodes],
    on_success_callback=log_task_end,
    on_failure_callback=log_task_failure,
)

...

start >> create_dbs_task >> editors_db_group \
      >> primary_language_task >> web_db_group >> end
```

Nel codice in lst. 3.2 è mostrato un esempio di definizione di una task (`create_dbs`) e la struttura delle dipendenze del DAG. Come si può notare la task chiama la funzione `create_db`, definita in un file nella cartella `scripts/` e importata come una libreria. Il parametro `op_args` permette di passare argomenti alla funzione, in questo caso la lista delle lingue da processare.

Dopo aver descritto la struttura complessiva del DAG, è utile analizzare nel dettaglio le singole task che lo compongono. Per rendere la pipeline più modulare e leggibile, lo script originario è stato suddiviso in sei parti principali, ciascuna delle quali è associata a una specifica fase del flusso e implementata come task indipendente. In questo modo ogni passaggio è incapsulato in una funzione python separata ed eseguito tramite un `PythonOperator`.

Nelle sottosezioni che seguono verranno illustrate le task una ad una, evidenziandone lo scopo e la logica implementativa. Infine, verrà presentato un esempio di refactoring relativo al calcolo della lingua primaria degli editor, dove l'utilizzo della libreria Pandas ha permesso di migliorare leggibilità ed efficienza del codice.

3.1.2 Task create_dbs

La prima task del DAG è `create_dbs`, responsabile dell'inizializzazione dei database utilizzati dalla pipeline. Si tratta di un passo preliminare che assicura la presenza e la corretta struttura delle tabelle necessarie sia per il database degli editor sia per quello dei vital sign. La task è implementata come `PythonOperator` e chiama la funzione `create_db`, definita in un file (`create_db.py`) nella cartella `scripts/`.

Lo script utilizza la libreria `SQLAlchemy` per gestire le connessioni ai due database (tramite le stringhe di connessione definite in `config.py`). Per ciascuna lingua in `wikilanguagecodes` vengono create due tabelle nel database `vital_signs_editors`:

- `<langcode>wiki_editors`, che raccoglie le informazioni anagrafiche e di profilo degli editor.
- `<langcode>wiki_editor_metrics`, che memorizza le metriche derivate dai dump a livello di utente e mese.

Infine, nel database `vital_signs_web` viene inizializzata la tabella `vital_signs_metrics`, che conterrà gli indicatori aggregati calcolati nelle fasi successive.

Il codice seguente mostra le definizioni SQL semplificate delle tre tabelle principali:

Listing 3.3: Definizione delle tabelle create dalla task create_dbs

```
-- Tabella <langcode>wiki_editors
CREATE TABLE IF NOT EXISTS <langcode>wiki_editors (
    user_id INTEGER,
    user_name TEXT PRIMARY KEY,
    bot TEXT,
    user_flags TEXT,
    highest_flag TEXT,
    highest_flag_year_month TEXT,
    gender TEXT,
    primarybinary INTEGER,
    primarylang TEXT,
    edit_count INTEGER,
    primary_ecount INTEGER,
    totallangs_ecount INTEGER,
    primary_year_month_first_edit TEXT,
    primary_lustrum_first_edit TEXT,
    numberlangs INTEGER,
    registration_date TEXT,
    year_month_registration TEXT,
    first_edit_timestamp TEXT,
    year_month_first_edit TEXT,
    year_first_edit TEXT,
    lustrum_first_edit TEXT,
    survived60d TEXT,
    last_edit_timestamp TEXT,
    year_last_edit TEXT,
    lifetime_days INTEGER,
    days_since_last_edit INTEGER
);
-- Tabella <langcode>wiki_editor_metrics
CREATE TABLE IF NOT EXISTS <langcode>wiki_editor_metrics (
    user_id INTEGER,
    user_name TEXT,
    abs_value TEXT,
    rel_value REAL,
    metric_name TEXT,
    year_month TEXT,
    timestamp TEXT,
    PRIMARY KEY (user_id, metric_name, year_month, timestamp)
);
-- Tabella vital_signs_metrics
CREATE TABLE IF NOT EXISTS vital_signs_metrics (
    langcode TEXT,
    year_year_month TEXT,
    year_month TEXT,
    topic TEXT,
    m1 TEXT,
    m1_calculation TEXT,
    m1_value TEXT,
```

```
    m2 TEXT,
    m2_calculation TEXT,
    m2_value TEXT,
    m1_count FLOAT,
    m2_count FLOAT,
    PRIMARY KEY (langcode, year_year_month, year_month, topic,
                 m1, m1_calculation, m1_value,
                 m2, m2_calculation, m2_value)
);
```

In sintesi, le prime due tabelle costituiscono il livello "di base" per ciascuna lingua, registrando rispettivamente i dati anagrafici degli editor e le metriche puntuali calcolate dai dump. La terza tabella rappresenta invece il livello "aggregato" e contiene i vital sign, calcolati a partire dalle metriche di base e organizzati in modo idempotente grazie a una chiave primaria composta che impedisce inserimenti duplicati.

Al termine dell'esecuzione di questa task l'ambiente risulta pronto per accogliere i dati estratti dai dump e per memorizzare i risultati delle analisi successive.

3.1.3 Task <langcode>_process_dump

La task `<langcode>_process_dump` è la più impegnativa del DAG in termini computazionali. Per ogni edizione linguistica viene creata una task dedicata che legge i MediaWiki History Dumps compressi (`.bz2`) e ne estrae i dati rilevanti sugli editor, trasformando i record grezzi in metriche strutturate da salvare nel database `vital_signs_editors`. Questa impostazione consente al DAG di scalare su più comunità linguistiche semplicemente modificando la lista `wikilanguagecodes`, da cui Airflow genera dinamicamente le task.

Idx	Variabile	Tipo	Descrizione
1	event_entity	string	Entità dell'evento: revision, user o page.
2	event_type	string	Tipo di evento: create, move, delete, ecc.
3	event_timestamp	string	Timestamp in cui si è verificato l'evento.
5	event_user_id	bigint	ID dell'utente che ha generato l'evento (null se anonimo o revision-deleted).
7	event_user_text	string	Username corrente dell'utente (IP se anonimo).
11	event_user_groups	array <string >	Gruppi correnti a cui appartiene l'utente.
13	event_user_is_bot_by	array <string >	Informazioni sullo stato di bot (nome o gruppo).
17	event_user_is_anonymous	boolean	Indica se l'utente è anonimo.
20	event_user_registration_timestamp	string	Timestamp di registrazione dell'utente (da tabella user).
21	event_user_creation_timestamp	string	Timestamp di creazione account (da logging).
22	event_user_first_edit_timestamp	string	Timestamp del primo edit dell'utente.
23	event_user_revision_count	bigint	Numero cumulativo di revisioni effettuate fino all'evento (include l'evento stesso).
30	page_namespace	int	Namespace corrente della pagina.
38	user_id	bigint	Identificativo dell'utente (in eventi di tipo user).
40	user_text	string	Username o indirizzo IP corrente (in eventi di tipo user).
44	user_groups	array <string >	Gruppi correnti dell'utente (in eventi di tipo user).

Tabella 3.1: Variabili dei MediaWiki History Dumps utilizzate dalla pipeline con relativi indici, tipi e descrizioni.

I principali dati estratti e rielaborati sono i seguenti:

- **Identificazione dell'utente**: vengono considerati solo eventi con `event_user_id` valido e non anonimo (`event_user_is_anonymous = False`). A ciascun utente vengono associati username (`event_user_text`), flag e stato (bot/non-bot).
- **Informazioni temporali**: da `event_timestamp` si derivano date di registrazione, primo edit, ultimo edit, anno e mese dei contributi.
- **Flag e gruppi**: dalle variabili `event_user_groups` e dagli eventi di tipo `altergroups` vengono ricostruite le modifiche di gruppo nel tempo. Questi eventi sono salvati in `wiki_editor_metrics` con metriche `granted_flag` e `removed_flag`.
- **Monthly edits**: per ogni utente e mese viene calcolato il numero di revisioni (`revision_id`) effettuate, salvato come metrica `monthly_edits`.

- **Namespace edits**: il campo `page_namespace` distingue i contributi nei namespace di coordinamento (4, 12) e tecnici (8, 10), salvati rispettivamente come `monthly_edits_coordination` e `monthly_edits_technical`.

- **Metriche di sopravvivenza**: a partire dal `event_user_first_edit_timestamp` vengono calcolate le soglie di attività entro 24 ore, 7 giorni, 30 giorni e 60 giorni dal primo edit (`edit_count_24h`, `edit_count_7d`, `edit_count_30d`, `edit_count_60d`). Tali metriche servono come base per la successiva analisi di retention.

- **Caratteristiche anagrafiche e storiche**: per ogni utente vengono salvati attributi quali data di registrazione, anno/mese del primo e ultimo edit, lustro di ingresso (`lustrum_first_edit`), numero di giorni di vita dell'account (`lifetime_days`) e giorni trascorsi dall'ultimo edit (`days_since_last_edit`). È inoltre marcato se l'utente è sopravvissuto almeno 60 giorni (`survived60d`).

I risultati vengono infine inseriti in due tabelle distinte:

- `<langcode>wiki_editor_metrics`, che raccoglie le metriche mensili (edits totali, edits per namespace) e le misure di sopravvivenza (edit_count_*).

- `<langcode>wiki_editors`, che conserva le informazioni anagrafiche e di attività degli utenti (flag, lingua primaria, date di registrazione/attività, edit count complessivo, ecc.).

Tutti gli inserimenti avvengono in modalità idempotente (`ON CONFLICT DO NOTHING` o `DO UPDATE`), in modo da evitare duplicazioni quando la pipeline viene rieseguita sugli stessi dump. In questo modo la task produce un dataset consistente e normalizzato, da cui le fasi successive del DAG (`calc_flags`, `calc_streaks`, `calc_vs`) possono derivare i vital sign.

3.1.4 Task <langcode>_calc_flags

La task `<langcode>_calc_flags` ha lo scopo di analizzare i flag assegnati agli editor e individuare per ciascun utente quello di rango più elevato, salvandolo nella tabella degli editor. Per ogni edizione linguistica viene generata una task dedicata che prende come input il codice della lingua e utilizza i dati prodotti dalla fase precedente e salvati nel database `vital_signs_editors`.

Lo script associato alla task `calc_flags` (`calculate_editors_flag`) si occupa di determinare, per ciascun utente, il flag più rilevante ricevuto durante la sua attività. In un primo passaggio vengono contati i flag presenti tra tutti gli editor, così da costruire un dizionario di supporto. Successivamente è definita una gerarchia di priorità (`flag ranks`) che assegna un valore crescente ai diversi flag: dai più comuni, come `confirmed`, fino a quelli di maggiore responsabilità, come `steward` o `founder`.

Per ogni editor viene quindi valutata la lista dei flag posseduti e viene individuato quello con rango massimo, salvato nel campo `highest_flag` della tabella `wiki_editors`. Nel caso in cui più flag abbiano lo stesso livello, viene selezionato quello più diffuso all'interno della comunità. Lo script aggiorna inoltre il campo `highest_flag_year_month`, basandosi sugli eventi di concessione flag (`granted_flag`) registrati in `wiki_editor_metrics`, così da associare il flag principale al momento in cui è stato effettivamente ottenuto. Infine, viene gestito il caso specifico degli utenti che hanno ricevuto un flag di tipo `bot`, aggiornando opportunamente il campo `bot` nella tabella degli editor.

In questo modo la task arricchisce il profilo di ciascun editor con informazioni sul ruolo più significativo ricoperto nella comunità e sul momento in cui tale ruolo è stato raggiunto.

3.1.5 Task <langcode>_calc_streaks

La task `<langcode>_calc_streaks` ha l'obiettivo di analizzare la continuità dell'attività degli editor, calcolando per ciascun utente le cosiddette streaks, ovvero il numero di mesi consecutivi in cui ha effettuato almeno un edit. Per ogni lingua inclusa in `wikilanguagecodes` viene generata una task che prende in input il codice linguistico e lavora sui dati presenti nel database `vital_signs_editors`.

Lo script associato (`calculate_editor_activity_streaks`) esegue i seguenti passaggi: parte dai record della tabella `wiki_editor_metrics` relativi alla metrica `monthly_edits`, ordinati per utente e mese. Per ciascun editor viene mantenuto un contatore che misura la lunghezza della sequenza di mesi consecutivi attivi. Se i mesi seguono senza interruzioni, il contatore viene incrementato e il valore corrente

della streak viene salvato come nuova metrica denominata `active_months_row`. Se invece vengono rilevati mesi mancanti, la streak viene interrotta e il contatore azzerato.

Al termine dell'elaborazione, i risultati vengono inseriti nuovamente nella tabella `<langcode>wiki_editor_metrics`, utilizzando un'operazione di insert con clausola `ON CONFLICT DO NOTHING` per garantire l'idempotenza. In questo modo ogni editor viene arricchito con un indicatore che riflette la sua costanza di partecipazione nel tempo, utile per stimare il livello di stabilità e di continuità della comunità.

3.1.6 Task calc_primary_language

La task `calc_primary_language` ha l'obiettivo di determinare, per ciascun editor che contribuisce a più progetti linguistici, la lingua principale in cui è attivo e di arricchire le informazioni presenti nelle rispettive tabelle degli editor. In questo modo è possibile ottenere una visione trasversale dell'attività degli utenti e valutare il peso relativo che ciascuna lingua ha nella loro storia di contributi.

Lo script associato (`cross_wiki_editor_metrics`) utilizza la libreria `pandas` per aggregare i dati provenienti dalle varie tabelle `wiki_editors`, una per ogni lingua contenuta nella lista `wikilanguagecodes`. In una prima fase estrae da ciascuna tabella le informazioni di base (username, numero di edit, data del primo edit, lustro di ingresso) e le unifica in un unico `DataFrame`. Successivamente calcola:

- il numero totale di edit effettuati da ogni editor su tutte le lingue;
- il numero di lingue in cui l'editor ha realizzato più di quattro edit, utilizzato come indicatore di attività multi-lingua;
- la lingua primaria, individuata come quella in cui l'editor ha effettuato il maggior numero di edit, insieme al corrispondente numero di edit e alle date del primo contributo.

Infine, per ciascun editor i risultati vengono salvati nuovamente in tutte le tabelle `wiki_editors`, aggiornando i campi `primarylang`, `primary_ecount`, `totallangs_ecount`, `numberlangs`, `primary_year_month_first_edit` e `primary_lustrum_first_edit`. In questo modo ogni record è arricchito con informazioni coerenti sul ruolo dell'editor all'interno del panorama cross-wiki, fornendo un indicatore utile per le analisi successive sui vital sign.

3.1.7 Task <langcode>_calc_vs

Lo scopo della task `<langcode>_calc_vs` è calcolare e salvare un insieme di indicatori, i cosiddetti vital sign, che descrivono lo stato di salute della comunità di una specifica edizione linguistica di Wikipedia. Questa fase prende i dati già preprocessati nelle tabelle `wiki_editors` e `wiki_editor_metrics` e li combina in metriche aggregate su base mensile o annuale, producendo misure riguardanti la retention, la numerosità e distribuzione degli editor attivi, la stabilità e il ricambio generazionale, i profili speciali (technical editor e coordinator), la distribuzione delle flag amministrative e la distinzione fra editor primari e globali. I risultati vengono salvati nella tabella centralizzata `vital_signs_metrics`.

La funzione `compute_wiki_vital_signs(languagecode)` realizza questi calcoli connettendosi ai database tramite SQLAlchemy: `vital_signs_editors` come sorgente e `vital_signs_web` come destinazione. Se non esiste, viene creata la tabella `vital_signs_metrics` con una chiave primaria composta da: `(langcode, year_year_month, year_month, topic, m1, m1_calculation, m1_value, m2, m2_calculation, m2_value)`. Questa struttura assicura l'idempotenza: lo stesso conteggio non può essere inserito più volte poiché gli `INSERT` avvengono con la clausola `ON CONFLICT DO NOTHING`. Ogni riga rappresenta un esperimento di conteggio, in cui `m1` identifica la metrica primaria (ad es. monthly_edits, threshold, 5) e `m2` una condizione aggiuntiva o una suddivisione (ad es. active_months_row, bin, 7-12). I valori numerici sono salvati nei campi `m1_count` (denominatore) e `m2_count` (numeratore condizionato).

Retention Per misurare la capacità della comunità di trattenere i nuovi utenti, lo script costruisce due baseline, ovvero i valori di riferimento: il numero di utenti registrati in ciascun mese (`year_month_registration`) e il numero di utenti al primo edit (`year_month_first_edit`). Su queste coorti calcola quanti editor restano attivi dopo 24 ore, 7, 30, 60, 365 e 730 giorni dal primo contributo, interrogando i campi `edit_count_*` in `wiki_editor_metrics`. Ogni misura produce due righe: una confrontata con la baseline dei registrati e una con quella dei first edit.

Active editor Gli editor vengono classificati in base al numero di edit mensili: attivi (almeno 5) e molto attivi (almeno 100). Oltre alle soglie vengono costruiti istogrammi per classi di attività (1-5, 5-10, 10-50, ... fino a >10000). Questi valori alimentano altre metriche, ad esempio stability e balance.

Stability La stabilità combina le soglie di attività con la durata della partecipazione, misurata tramite `active_months_row`, che indica i mesi consecutivi di attività di un editor. I valori sono suddivisi in bin (2, 3-6, 7-12, 13-24, >24 mesi). Per ciascun mese la somma dei bin è confrontata con la baseline, mentre la classe "1 mese attivo" è calcolata come complemento, garantendo che le percentuali coprano il 100%.

Balance Il bilanciamento misura il ricambio generazionale suddividendo gli editor per `lustrum_first_edit` (lustro del primo contributo, ad esempio 2001–2005, 2006–2010). In questo modo si osserva la quota di nuove e vecchie generazioni fra gli editor attivi.

Special function e primary editor `monthly_edits_technical` per i technical editor e `monthly_edits_coordination` per i coordinators. Inoltre, vengono calcolati i primary editor, distinguendo quanti editor attivi hanno come lingua principale quella analizzata e quanti provengono da altre wiki (`primarylang`).

Flag e amministratori Tra gli editor attivi vengono conteggiati i ruoli più alti (`highest_flag`, ad esempio sysop, autopatrolled, bureaucrat). A livello annuale si calcolano anche i dati sugli amministratori distinguendo tra flussi e stock:

- i flussi rappresentano le variazioni, cioè quanti utenti hanno ricevuto un flag (`granted_flag`) o lo hanno perso (`removed_flag`) in un certo anno;
- lo stock rappresenta lo stato complessivo, cioè quanti utenti detengono un certo flag in quell'anno.

Ad esempio, se nel 2021 cinque utenti hanno ricevuto il ruolo di sysop e due lo hanno perso, i flussi registrano +5 e -2, mentre lo stock indica quanti sysop risultano attivi complessivamente nel 2021.

In sintesi, la task `<langcode>_calc_vs` trasforma le metriche elementari sugli editor in indicatori di livello superiore, permettendo di monitorare nel tempo la vitalità e la sostenibilità delle comunità. I risultati, salvati nella tabella `vital_signs_metrics`, costituiscono la base dati per le analisi e la visualizzazione tramite la dashboard.

3.2 Containerizzazione con Docker

In questa sezione viene descritta la containerizzazione della pipeline, realizzata attraverso Docker. L'obiettivo di questa fase è stato quello di racchiudere ogni componente del sistema (Airflow, database, strumenti di monitoraggio) all'interno di container isolati ma interoperabili, così da semplificare il deploy, garantire portabilità e facilitare la manutenzione.

3.2.1 Dockerfile

Per utilizzare Apache Airflow all'interno della pipeline è stata realizzata un'immagine Docker personalizzata, costruita a partire da `apache/airflow:2.10.5-python3.12`. Il Dockerfile ha lo scopo di configurare un ambiente Airflow completo, includendo le DAG sviluppate, gli script Python di supporto, le dipendenze esterne e le directory necessarie per la gestione dei log. In questo modo si ottiene un ambiente isolato, replicabile e facilmente distribuibile nei vari contesti di esecuzione.

Di seguito è riportato il contenuto del Dockerfile utilizzato nel progetto:

Listing 3.4: Dockerfile per la creazione dell'immagine personalizzata di Airflow

```
FROM apache/airflow:2.10.5-python3.12
WORKDIR /opt/airflow

USER root
COPY requirements.txt /requirements.txt
COPY --chown=airflow:root dags/ /opt/airflow/dags/
COPY --chown=airflow:root scripts/ /opt/airflow/scripts/
RUN mkdir -p  /opt/airflow/logs
RUN chown -R airflow: /opt/airflow/logs

USER airflow
RUN pip install --no-cache-dir -r /requirements.txt
```

3.2.2 docker-compose.yml

Il file `docker-compose.yml` rappresenta il cuore del progetto: attraverso la sua definizione è possibile orchestrare in modo coordinato tutte le componenti della pipeline. Grazie a Docker Compose, con un unico comando si avviano i diversi servizi che compongono l'architettura, garantendo isolamento, riproducibilità e una gestione semplificata delle dipendenze. Nel file sono stati evitati sia l'utilizzo del tag `latest` per le immagini, sia l'impiego di named volume, in quanto sconsigliati in ambienti di produzione.Ho inoltre utilizzato un file `.env` (letto automaticamente da Docker Compose) per la gestione delle variabili d'ambiente, in modo da separare le configurazioni sensibili dal codice.

I servizi definiti nel file possono essere suddivisi in quattro gruppi funzionali principali:

- **Airflow**: comprende i servizi necessari all'esecuzione e al monitoraggio dei workflow, ovvero `airflow` (il webserver), `scheduler` e `airflow_init` (il servizio per l'inizializzazione del database di Airflow);
- **Database**: include il servizio `postgres`, utilizzato come backend sia per Airflow che per la persistenza delle metriche calcolate;
- **Monitoraggio**: raccoglie i servizi `prometheus`, `grafana` e `statsd-exporter`, responsabili dell'osservabilità della pipeline;
- **Dashboards**: comprende il servizio `dash-app`, che fornisce un'interfaccia interattiva per la visualizzazione delle metriche sulle comunità.

In seguito, ciascun servizio viene descritto in dettaglio, illustrandone il ruolo e le configurazioni principali.

Servizi Airflow

Tutti i servizi utilizzano l'immagine `custom-airflow:v1` (costruita dal `Dockerfile`) e montano i volumi `/opt/airflow/dags`, `/opt/airflow/scripts`, `/opt/airflow/mediawiki_history` e `/opt/airflow/logs`.

x-airflow-env I servizi di Airflow condividono la stessa configurazione tramite l'ancora YAML `x-airflow-env`, che imposta in modo uniforme le principali variabili d'ambiente:

- `TMPDIR=/opt/airflow/tmp`: directory temporanea usata dai processi di Airflow per file intermedi.
- `AIRFLOW__DATABASE__SQL_ALCHEMY_CONN`: URI di connessione a PostgreSQL (`postgresql+psycopg2://...`) usato come backend dei metadati.
- `AIRFLOW__CORE__EXECUTOR=LocalExecutor`: abilita l'esecuzione concorrente delle task come processi locali.
- `AIRFLOW__CORE__LOAD_EXAMPLES=False`: evita il caricamento dei DAG di esempio.

- `AIRFLOW__WEBSERVER__DEFAULT_UI_TIMEZONE=utc`: imposta il fuso orario della UI per coerenza di log e pianificazioni.
- `AIRFLOW__CORE__DAGS_FOLDER=/opt/airflow/dags`: path delle definizioni dei DAG.
- `AIRFLOW__METRICS__STATSD_ON=True`, `...HOST=statsd-exporter`, `...PORT=9125`, `...PREFIX=airflow`: esportazione delle metriche Airflow via StatsD verso lo `statsd-exporter`.
- `AIRFLOW__LOGGING__BASE_LOG_FOLDER=/opt/airflow/logs`: directory di log condivisa tra servizi.
- `AIRFLOW__LOGGING__REMOTE_LOGGING=False`: log mantenuti localmente.

airflow_init `airflow_init` prepara l'ambiente prima dell'esecuzione dei componenti runtime. La sua esecuzione dipende da `postgres`. All'avvio esegue la sequenza di inizializzazione:

```
/bin/bash -c "airflow db init && airflow users create \
    --username ${AIRFLOW_USER} \
    --password ${AIRFLOW_PASSWORD} \
    --firstname Andrea --lastname Denina \
    --role Admin --email andredeninaf@gmail.com"
```

Il primo comando crea tutte le tabelle dei metadati su PostgreSQL; il secondo registra un utente amministratore per l'accesso all'interfaccia web. Questo servizio deve completarsi con successo prima di avviare webserver e scheduler.

airflow (webserver) Il servizio `airflow` esegue il webserver (`command: webserver`). Espone le porte `8080` (interfaccia web) e `8000`. Il webserver viene avviato dopo l'inizializzazione del DB e la creazione dell'utente.

scheduler Lo `scheduler` interpreta i DAG e pianifica le task, per essere avviato viene eseguito il comando `scheduler`. Oltre ai volumi condivisi, monta anche `/opt/airflow/tmp` come volume temporaneo. Dipende da `airflow_init` e `postgres`, assicurando che il database sia pronto prima di iniziare la pianificazione.

PostgreSQL

Il servizio `postgres` fornisce il database relazionale utilizzato sia come metadatabase di Airflow sia come storage applicativo per le tabelle della pipeline. È basato sull'immagine ufficiale `postgres:15` ed è configurato tramite le variabili d'ambiente `POSTGRES_USER`, `POSTGRES_PASSWORD` e `POSTGRES_DB`, che definiscono rispettivamente utente, password e nome del database predefinito (usato da Airflow per i metadati tramite l'URI `AIRFLOW__DATABASE__SQL_ALCHEMY_CONN`).
Per garantire la persistenza, i dati sono salvati nel volume `./postgres-data:/var/lib/postgresql/data`. All'avvio, lo script `init_multidb.sql` viene montato nella directory di bootstrap `/docker-entrypoint-initdb.d/` e crea i due database applicativi:

```
CREATE DATABASE vital_signs_web; CREATE DATABASE vital_signs_editors;
```

Il servizio espone un healthcheck basato su `pg_isready` eseguito ogni 10 secondi con 5 tentativi (`retries: 5`); finché il database non risulta pronto, gli altri container che vi dipendono non proseguono l'inizializzazione. La policy `restart: always` assicura il riavvio automatico in caso di errore. Complessivamente, questa configurazione rende il database affidabile e idoneo sia all'orchestrazione (metadati Airflow) sia alla persistenza dei risultati della pipeline.

Servizi di monitoraggio

L'infrastruttura di monitoraggio è composta da tre servizi: `prometheus`, `grafana` e `statsd-exporter`. Questi container permettono di raccogliere, indicizzare e visualizzare le metriche prodotte da Airflow.

prometheus Il servizio `prometheus` si basa sull'immagine `prom/prometheus:v2.53.5`. Monta il file di configurazione locale `./monitoring/prometheus.yml` all'interno del container nel percorso `/etc/prometheus/prometheus.yml`. Espone la porta `9090`, attraverso cui è possibile accedere all'interfaccia web e alle API per l'interrogazione delle metriche.

grafana Il servizio `grafana` utilizza l'immagine `grafana/grafana:12.0.2`. Per garantire la persistenza dei dati e una configurazione automatica monta tre volumi:

- `./grafana-data:/var/lib/grafana`, per i dati persistenti (utenti, dashboard, preferenze);
- `./monitoring/grafana/dashboards:/etc/grafana/provisioning/dashboards`, per il caricamento delle dashboard personalizzate;
- `./monitoring/grafana/datasources:/etc/grafana/provisioning/datasources`, per la definizione delle sorgenti dati.

Espone la porta `3000`, accessibile tramite browser. È configurato mediante le variabili d'ambiente:

- `GF_SECURITY_ADMIN_USER` e `GF_SECURITY_ADMIN_PASSWORD`, che impostano le credenziali dell'utente amministratore;
- `GF_INSTALL_PLUGINS`, che abilita l'installazione automatica di plugin aggiuntivi (nel progetto: `grafana-piechart-panel`).

statsd-exporter Il servizio `statsd-exporter` è basato sull'immagine `prom/statsd-exporter:v0.27.2`. Monta il file di mapping `./monitoring/statsd.yaml` nella directory `/etc/statsd.yaml`. Espone due porte: `9125` per la ricezione delle metriche StatsD da Airflow, e `9102` per l'esposizione delle metriche in formato Prometheus. Il servizio viene avviato con un comando personalizzato che abilita il logging dettagliato e carica la configurazione di mapping:

```
statsd_exporter --log.level debug --statsd.mapping-config=/etc/statsd.yaml
```

Nella sezione successiva (Sezione 3.3) verrà approfondita l'integrazione e la configurazione dell'intero sistema di estrazione e analisi delle metriche, mostrando come questi componenti cooperino per fornire un'osservabilità completa della pipeline.

Dashboard

Il servizio `dash-app` è costruito a partire dal `Dockerfile` presente nella directory `./dashboards` ed espone la porta `8050`. Su questo container viene eseguita un'applicazione Dash che fornisce le dashboard per la visualizzazione dei vital sign. Il servizio è stato utilizzato unicamente nella fase di sviluppo, con lo scopo di verificare visivamente la correttezza delle metriche calcolate e valutare l'integrazione della nuova architettura con il frontend preesistente.

3.3 Metriche e monitoraggio

Il sistema di monitoraggio della pipeline è progettato per raccogliere, trasformare, archiviare e visualizzare le metriche generate da Airflow. Ogni componente è connesso al successivo in una catena di elaborazione che va dall'emissione delle metriche fino alla loro rappresentazione grafica nelle dashboard.

Metriche generate da Airflow Airflow produce automaticamente un insieme di metriche in formato `StatsD`, riguardanti lo stato e le performance dei DAG e delle task. Tra le numerose metriche disponibili, ho scelto di monitorare in particolare:

- numero di task iniziate e completate (`ti.start`, `ti.finish` per ciascun `dag_id`/`task_id`);
- stato di esecuzione delle task (successo o fallimento);

- durata delle task (`dag.<dag_id>.<task_id>.duration`);
- utilizzo medio di CPU e memoria da parte delle task (`task.cpu_usage`, `task.mem_usage`);

Queste metriche sono inviate da Airflow al servizio `statsd-exporter` sulla porta `9125`:

Nome - StatsD	Nome - Prometheus	Tipo	Descrizione
`ti.start.<dag_id>.<task_id>`	`ti_start`	Counter	Numero di task iniziate in un dato DAG.
`ti.finish.<dag_id>.<task_id>.<state>`	`ti_finish`	Counter	Numero di task completate in un dato DAG, etichettate per stato (success, failed, ecc.).
`task.cpu_usage.<dag_id>.<task_id>`	`cpu_usage`	Gauge	Percentuale di CPU utilizzata da una task.
`task.mem_usage.<dag_id>.<task_id>`	`mem_usage`	Gauge	Percentuale di memoria utilizzata da una task.
`dag.<dag_id>.<task_id>.duration`	`task_duration`	Histogram	Millisecondi necessari per eseguire una certa task in un dato DAG.

Tabella 3.2: Metriche raccolte e trasformate da StatsD a Prometheus con relativo tipo e descrizione.

Mapping con statsd-exporter Il servizio `statsd-exporter` riceve i pacchetti StatsD e li converte in metriche compatibili con Prometheus. Il file di configurazione `statsd.yaml` definisce il mapping tra i nomi in notazione puntata (dot notation) e metriche con etichette Prometheus. Ad esempio, la metrica `ti.finish.my_dag.my_task.success` viene trasformata in:

```
ti_finish{dag_id="my_dag", task_id="my_task", state="success"} 1
```

L'output delle metriche trasformate è esposto sull'endpoint HTTP `/metrics` della porta `9102`. Un esempio di output è il seguente:

```
# HELP ti_finish Numero di Task completate in un dato DAG
# TYPE ti_finish counter
ti_finish{dag_id="vital_signs", task_id="calc_vs", state="success"} 42
```

Prometheus come storage Prometheus è configurato per effettuare lo scraping dell'endpoint `statsd-exporter:9102/metrics` ogni 5 secondi. Il file `prometheus.yml` definisce lo scrape job e l'intervallo di campionamento. In questo progetto Prometheus non è stato utilizzato per l'alerting, ma esclusivamente come:

- time series database;
- datasource per Grafana.

In tal modo le metriche generate da Airflow vengono archiviate e rese interrogabili attraverso PromQL.

Grafana Per Grafana è stato adottato il meccanismo di provisioning, ovvero la definizione dichiarativa della configurazione tramite file, che consente di automatizzare la creazione delle sorgenti dati e il caricamento delle dashboard senza dover intervenire manualmente dall'interfaccia web.

In particolare, il file `datasources.yaml` definisce Prometheus come sorgente dati principale, specificandone il tipo, l'URL di accesso (`http://prometheus:9090`) e ulteriori opzioni di configurazione. Allo stesso modo, il file `dashboards.yaml` stabilisce i parametri per il caricamento automatico delle dashboard, che sono codificate in formato JSON e mantenute all'interno della cartella `./monitoring/grafana/dashboards`. Questa configurazione garantisce che l'intero sistema di visualizzazione sia riproducibile e portabile: ogni volta che Grafana viene avviato, datasource e dashboard vengono istanziati coerentemente all'ambiente, evitando interventi manuali e assicurando uniformità tra diverse esecuzioni. Queste sono le query PromQL utilizzate per costruire le dashboard:

```
avg by(dag_id) (cpu_usage{dag_id="vital_signs"})
avg by(dag_id) (mem_usage{dag_id="vital_signs"})
```

calcolano rispettivamente l'utilizzo medio di CPU e memoria per il DAG `vital_signs`, permettendo di monitorare il consumo delle risorse computazionali.

```
sum(ti_finish{dag_id="vital_signs", state="failed"})
sum(ti_finish{dag_id="vital_signs", state="success"})
sum(ti_start{dag_id="vital_signs"})
```

conteggiano rispettivamente il numero di task fallite, completate con successo e avviate. Questi valori vengono combinati in una visualizzazione che mostra la distribuzione degli esiti delle task.

```
up
```

metrica di servizio che indica se i target monitorati da Prometheus sono attivi e raggiungibili. Nella dashboard è utilizzata per fornire un'indicazione sullo stato del sistema.

```
task_duration_sum{dag_id="vital_signs", task_id="$task"} /
task_duration_count{dag_id="vital_signs", task_id="$task"}
```

calcola la durata media di esecuzione di una specifica task. La variabile globale `$task` contiene tutti i valori di `task_id` ed è impostabile tramite un menu a tendina nella dashboard, consentendo di filtrare la visualizzazione in base alla task selezionata.

```
histogram_quantile(0.95, sum by(le)(rate(task_duration_bucket[$__rate_interval])))
histogram_quantile(0.90, sum by(le)(rate(task_duration_bucket[$__rate_interval])))
histogram_quantile(0.50, sum by(le)(rate(task_duration_bucket[$__rate_interval])))
```

stimano rispettivamente i percentili 95, 90 e 50 (mediana) della distribuzione dei tempi di esecuzione delle task. Queste query permettono di analizzare in dettaglio la distribuzione e individuare anomalie o possibili colli di bottiglia.

Attraverso queste query è stato possibile costruire dashboard che evidenziano l'andamento delle esecuzioni, lo stato di salute del sistema e l'utilizzo delle risorse, fornendo una panoramica completa delle prestazioni della pipeline.

Connessione tra i componenti L'intero sistema segue un flusso lineare:

1. Airflow genera metriche in formato StatsD;
2. `statsd-exporter` le riceve sulla porta `9125`, le trasforma e le espone in formato Prometheus sulla porta `9102`;
3. Prometheus esegue lo scraping periodico dell'endpoint, salvando le serie temporali nel proprio database;
4. Grafana interroga Prometheus come data source e visualizza le metriche nelle dashboard configurate.

Questa integrazione assicura una completa osservabilità della pipeline.

3.4 Deployment

La pipeline è attualmente in esecuzione su un server Wikimedia, lo stesso che ospita i MediaWiki History Dumps. Tutti i componenti (Airflow, PostgreSQL, Prometheus, Grafana e applicazione Dash) sono containerizzati e orchestrati tramite `docker-compose`, consentendo di avviare l'intero sistema in modo coerente e riproducibile.

In questo ambiente di produzione l'esecuzione del DAG è programmata con cadenza mensile, in linea con il rilascio dei dump. Questo garantisce che le metriche vengano aggiornate regolarmente senza necessità di interventi manuali.

Al momento l'accesso al sistema è riservato e avviene esclusivamente tramite connessione `SSH` al server, senza un'interfaccia pubblica né una gestione multi-utente. Nonostante questa limitazione, la pipeline

opera a tutti gli effetti in un contesto produttivo, elaborando i dati e rendendo disponibili i risultati per le successive analisi e visualizzazioni.

4 Risultati e Conclusioni

In questo capitolo vengono presentati i principali risultati ottenuti al termine del tirocinio e le considerazioni conclusive sull'attività svolta. L'obiettivo è mostrare in che modo la nuova architettura implementata abbia migliorato l'efficienza, l'affidabilità e l'osservabilità del processo di calcolo delle Community Health Metrics, mettendo in evidenza le differenze rispetto alla soluzione originaria basata su uno script monolitico.

Dopo una breve panoramica dei risultati, corredata da esempi grafici tratti dall'interfaccia di Airflow, dalle dashboard di Grafana e dall'applicazione Dash, verranno descritte le competenze acquisite e il contributo personale portato al progetto. Infine, saranno discusse le possibili linee di sviluppo futuro e le riflessioni conclusive sull'esperienza, sottolineandone l'impatto sia sul piano tecnico sia su quello formativo.

4.1 Risultati Ottenuti

I principali risultati ottenuti al termine del tirocinio possono essere riassunti nei seguenti punti:

- **Modularizzazione del processo di calcolo:** La suddivisione del processo in task distinte ha permesso di migliorare la manutenibilità e la leggibilità del codice, facilitando l'individuazione e la risoluzione di eventuali problemi.
- **Automazione e schedulazione:** L'utilizzo di Apache Airflow ha consentito di automatizzare l'esecuzione del processo, con la possibilità di schedulare le esecuzioni e di monitorare lo stato delle task attraverso un'interfaccia web intuitiva.
- **Containerizzazione:** La containerizzazione della pipeline ha reso possibile l'isolamento delle dipendenze e la standardizzazione dell'ambiente di esecuzione, facilitando il deployment e la scalabilità del sistema.
- **Monitoraggio e osservabilità:** L'integrazione di Prometheus e Grafana ha permesso di raccogliere metriche dettagliate sul funzionamento della pipeline, offrendo una visione completa delle performance e dello stato del sistema attraverso dashboard personalizzate.
- **Migrazione a PostgreSQL:** La migrazione da SQLite a PostgreSQL ha migliorato la gestione dei dati, garantendo maggiori performance nelle operazioni di lettura e scrittura.

Questi risultati hanno portato a un sistema più robusto, efficiente e facile da gestire, in grado di rispondere meglio alle esigenze di calcolo delle Community Health Metrics. Per dare una dimostrazione concreta dei risultati, si riportano due schermate significative relative al funzionamento della nuova architettura.

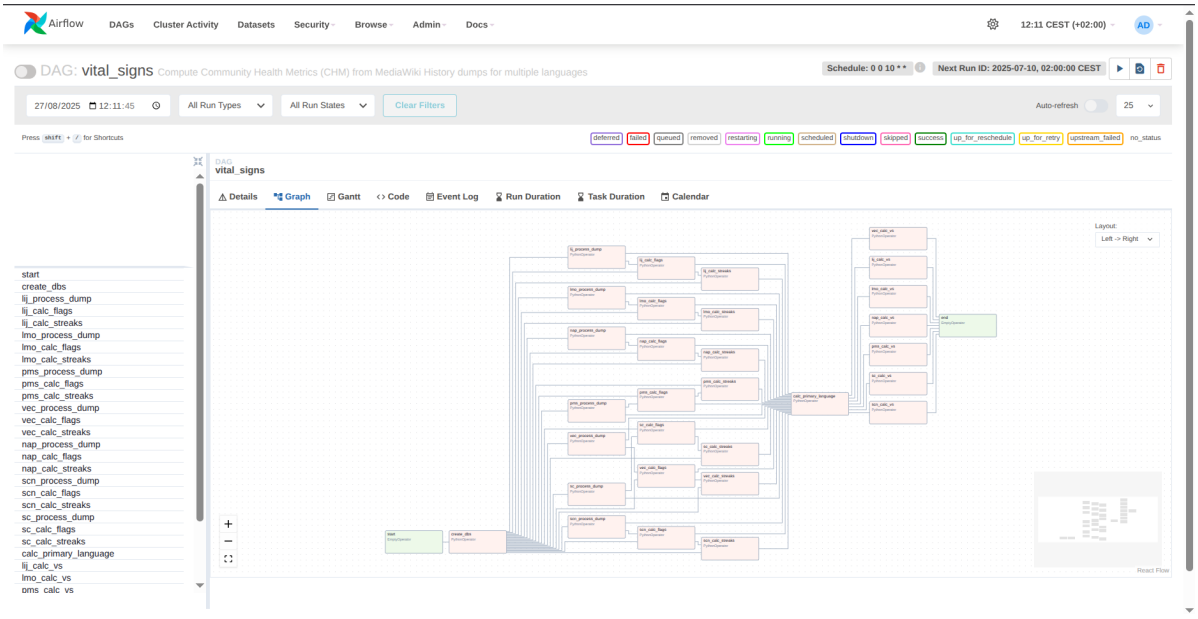

Figura 4.1: Visualizzazione del DAG nell'interfaccia grafica di Apache Airflow

Figura 4.2: Dashboard di monitoraggio realizzata con Grafana

4.2 Competenze Acquisite e Contributo Personale

Il tirocinio mi ha permesso di consolidare e ampliare in modo significativo le mie competenze nell'ambito dell'ingegneria del software. Il contributo personale si è concentrato sul refactoring del codice esistente, sulla progettazione del DAG in Airflow e sulla configurazione dell'infrastruttura containerizzata con Docker e docker-compose.

Dal punto di vista tecnico, le principali competenze acquisite sono state:

- **Python:** durante il progetto ho utilizzato per la prima volta Python in un contesto reale, acquisendo familiarità con la scrittura di script modulari, l'uso di librerie per la manipolazione dei dati e l'integrazione con database tramite SQLAlchemy.
- **SQL e PostgreSQL:** ho approfondito le mie conoscenze di SQL, imparando a scrivere query più complesse e a interagire con un database PostgreSQL, comprendendo le differenze rispetto a SQLite.
- **Docker**: ho imparato a creare immagini Docker personalizzate, a gestire contenitori e a orchestrare più servizi con docker-compose.
- **Apache Airflow:** ho avuto modo di sperimentare per la prima volta uno strumento di orchestrazione di workflow, imparando a definire DAG, gestire dipendenze tra task, configurare la schedulazione e utilizzare l'interfaccia web per il monitoraggio delle esecuzioni.
- **Tecniche di monitoraggio e osservabilità:** ho acquisito conoscenze di base sui concetti di metriche, logging e osservabilità, sperimentando l'integrazione di StatsD, Prometheus e Grafana per la raccolta e la visualizzazione dei dati relativi alle esecuzioni della pipeline.

Dal punto di vista personale, l'esperienza mi ha insegnato a lavorare su un progetto esistente apprendendo rapidamente tecnologie mai utilizzate prima, a risolvere problemi di integrazione tra componenti diversi e a strutturare una pipeline dati modulare e più affidabile.

4.3 Sviluppi Futuri e Considerazioni Conclusive

Il progetto realizzato durante il tirocinio rappresenta una solida base per ulteriori sviluppi e miglioramenti. Alcune possibili direzioni future includono:

- **Adattamento delle dashboard:** rendere compatibile l'applicazione Dash con il nuovo database PostgreSQL, consentendo la visualizzazione delle Community Health Metrics calcolate dalla pipeline.
- **Deployment delle dashboard:** deployare l'applicazione Dash su un server di produzione, integrandola con l'infrastruttura esistente e garantendo l'accesso sicuro agli utenti.
- **Ottimizzazione delle performance:** analizzare i colli di bottiglia nella pipeline e ottimizzare le performance delle task, ad esempio parallelizzando alcune operazioni o migliorando l'efficienza delle query SQL.
- **Implementazione di alerting:** configurare Prometheus per inviare notifiche in caso di anomalie o fallimenti nelle esecuzioni, migliorando la reattività del sistema.
- **Scalabilità:** esplorare soluzioni per scalare orizzontalmente la pipeline, ad esempio utilizzando Kubernetes per la gestione dei container.
- **Integrazione con altri strumenti:** valutare l'integrazione con altri strumenti di data engineering o machine learning per arricchire le funzionalità della pipeline.

In conclusione, il tirocinio ha rappresentato un'esperienza formativa estremamente positiva, permettendomi di acquisire competenze tecniche avanzate e di contribuire a un progetto reale. La nuova architettura implementata ha migliorato notevolmente l'efficienza, l'affidabilità e l'osservabilità del processo di calcolo delle Community Health Metrics, ponendo le basi per ulteriori sviluppi futuri.

Bibliografia

[1] Apache airflow. https://en.wikipedia.org/wiki/Apache_Airflow. Accesso: Agosto 2025.

[2] Apache airflow documentation. https://airflow.apache.org/docs/apache-airflow/2.10.5/index.html. Accesso: Agosto 2025.

[3] Astronomer - apache airflow. https://www.astronomer.io/airflow. Accesso: Agosto 2025.

[4] Dags - apache airflow. https://airflow.apache.org/docs/apache-airflow/2.10.5/core-concepts/dags.html. Accesso: Agosto 2025.

[5] Overview - apache airflow. https://airflow.apache.org/docs/apache-airflow/2.10.5/core-concepts/overview.html. Accesso: Agosto 2025.

[6] Tasks - apache airflow. https://airflow.apache.org/docs/apache-airflow/2.10.5/core-concepts/tasks.html. Accesso: Agosto 2025.

[7] What is configuration as code (cac) and 5 tips for success. https://configu.com/blog/what-is-configuration-as-code-cac-and-5-tips-for-success/. Accesso: Agosto 2025.

[8] Docker compose. https://docs.docker.com/compose/. Accesso: Agosto 2025.

[9] Docker overview. https://docs.docker.com/get-started/docker-overview/. Accesso: Agosto 2025.

[10] Docker storage volumes. https://docs.docker.com/engine/storage/volumes/. Accesso: Agosto 2025.

[11] Docker wikipedia en. https://en.wikipedia.org/wiki/Docker_(software). Accesso: Agosto 2025.

[12] Docker wikipedia it. https://it.wikipedia.org/wiki/Docker. Accesso: Agosto 2025.

[13] Dockerfile concepts. https://docs.docker.com/build/concepts/dockerfile/. Accesso: Agosto 2025.

[14] What is a container? https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/. Accesso: Agosto 2025.

[15] What is an image? https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-an-image/. Accesso: Agosto 2025.

[16] Docker IBM. https://www.ibm.com/think/topics/docker. Accesso: Agosto 2025.

[17] David Laniado Marc Miquel-Ribé, Cristian Consonni. Community vital signs: Measuring wikipedia communities' sustainable growth and renewal. *https://www.mdpi.com/journal/sustainability*, 2022. Accesso: Agosto 2025.

[18] Community health metrics - wikimedia. `https://meta.wikimedia.org/wiki/Community_Health_Metrics`. Accesso: Agosto 2025.

[19] Docker - wikitech. `https://wikitech.wikimedia.org/wiki/Docker`. Accesso: Agosto 2025.

[20] Mediawiki history dumps - wikimedia. `https://dumps.wikimedia.org/other/mediawiki_history/readme.html`. Accesso: Agosto 2025.

[21] Mediawiki history dumps - wikitech. https://wikitech.wikimedia.org/wiki/Data_Platform/Data_Lake/Edits/MediaWiki_history_dumps#Technical_Documentation. Accesso: Agosto 2025.

[22] Configure the prometheus data source - grafana labs. https://grafana.com/docs/grafana/latest/datasources/prometheus/configure/. Accesso: Agosto 2025.

[23] Grafana. https://en.wikipedia.org/wiki/Grafana. Accesso: Agosto 2025.

[24] Introduction to grafana & prometheus. https://medium.com/ Accesso: Agosto 2025.

[25] Observability vs monitoring. https://circleci.com/blog/observability-vs-monitoring/. Accesso: Agosto 2025.

[26] Prometheus overview. https://prometheus.io/docs/introduction/overview/. Accesso: Agosto 2025.

[27] Prometheus (software). https://en.wikipedia.org/wiki/Prometheus_(software). Accesso: Agosto 2025.

[28] Prometheus statsd exporter. https://github.com/prometheus/statsd_exporter. Accesso: Agosto 2025.

[29] Statsd. https://github.com/statsd/statsd. Accesso: Agosto 2025.

[30] What is observability with prometheus - grafana labs. https://grafana.com/docs/grafana-cloud/introduction/what-is-observability/prometheus/. Accesso: Agosto 2025.