Specifica Tecnica

v0.1

7Last

Versioni

Ver.	Data	Autore	Verificatore	Descrizione
0.1	02/06/2024	Matteo Tiozzo		Stesura struttura del documento

Indice

Indice delle tabelle

Indice delle immagini

1 Introduzione

1.1 Scopo della specifica tecnica

Questo documento è rivolto a tutti gli *stakeholder* coinvolti nel progetto ***SyncCity*** - *A smart city monitoring platform*. Esso ha lo scopo di fornire una visione dettagliata riguardo l'architettura del sistema, i *design pattern* utilizzati, le tecnologie adottate e le scelte progettuali effettuate. Inoltre, contiene diagrammi UML delle classi e delle attività.

1.2 Scopo del prodotto

Lo scopo del prodotto è realizzare un prototipo di una piattaforma di monitoraggio per una *Smart City*, la quale permetta di raccogliere e analizzare dati provenienti da sensori IoT posizionati nelle città. Questi dati, una volta elaborati, devono essere visualizzati in maniera chiara e intuitiva, tramite grafici e mappe, per permettere alle autorità locali della città di prendere decisioni tempestive e mirate per migliorare la qualità della vita dei cittadini.

1.3 Glossario

Per evitare qualsiasi ambiguità o malinteso sui termini utilizzati nel documento, verrà adottato un glossario. Questo glossario conterrà varie definizioni. Ogni termine incluso nel glossario sarà indicato applicando uno stile specifico:

- aggiungendo una "G" al pedice della parola;
- fornendo il link al glossario online.

1.4 Riferimenti

1.4.1 Normativi

-
-

1.4.2 Informativi

-
-

2 Tecnologie

Questa sezione si occupa di fornire una panoramica delle tecnologie utilizzate per implementare il sistema software. In particolare, delinea le piattaforme, gli strumenti, i linguaggi di programmazione, i framework e altre risorse tecnologiche che sono state impiegate durante lo sviluppo.

2.1 Docker

È una piattaforma di virtualizzazione leggera che semplifica lo sviluppo, il testing e il rilascio delle applicazioni fornendo un ambiente isolato e riproducibile. È utilizzato per creare ambienti di sviluppo standardizzati, facilitare la scalabilità delle applicazioni e semplificare la gestione delle risorse.

2.1.1 Ambienti

Per lo sviluppo di questo progetto sono stati ipotizzati i due seguenti scenari di esecuzione, separati grazie all'utilizzo di profili diversi di Docker Compose:

- `local`: utilizzato dagli sviluppatori per testare e sviluppare le funzionalità dell'applicazione sui propri computer. Questo ambiente permette di eseguire tutti i componenti del sistema all'interno di un container Docker, ad eccezione del simulatore Python. Esso viene eseguito direttamente sul sistema operativo dell'utente, in modo da facilitare il debugging e il testing delle funzionalità, senza dover necessariamente eseguire la *build* dell'immagine Docker ad ogni modifica del codice;
- `release`: utilizzato quando si desidera simulare un ipotetico ambiente di produzione o non è necessario modificare il codice Python. Consente di non dover manualmente installare le dipendenze o configurare l'ambiente di esecuzione. In questo caso, tutti i componenti del sistema vengono eseguiti all'interno di container Docker.

2.1.2 Immagini Docker

Nello sviluppo di questo progetto *7Last* ha utilizzato diverse immagini Docker di seguito elencate.

- **Simulator - Python**
  - **Immagine**: python:3.11.9-alpine;

  - **Riferimento**: Python Docker Image [Ultima consultazione: 2024-06-02].
  - **Ambiente**: `release`;

- **Redpanda Init**: l'immagine di `alpine` viene utilizzata per creare un container che si occupa di inizializzare il broker Redpanda.
  - **Immagine**: alpine:3.20.1;
  - **Riferimento**: Alpine [Ultima consultazione: 2024-06-25].
  - **Ambiente**: `local`, `release`.

- **Redpanda**
  - **Immagine**: docker.redpanda.com/redpandadata/redpanda:v23.3.11;
  - **Riferimento**: Redpanda Docker Image [Ultima consultazione: 2024-06-02].
  - **Ambiente**: `local`, `release`.

- **Redpanda console**
  - **Immagine**: docker.redpanda.com/redpandadata/console:v2.4.6;
  - **Riferimento**: Redpanda Console Docker Image [Ultima consultazione: 2024-06-02].
  - **Ambiente**: `local`, `release`.

- **Connectors**
  - **Immagine**: docker.redpanda.com/redpandadata/connectors:v1.0.27;
  - **Riferimento**: Redpanda Connectors Docker Image [Ultima consultazione: 2024-06-02].
  - **Ambiente**: `local`, `release`.

- **ClickHouse**
  - **Immagine**: clickhouse/clickhouse-server:24-alpine;
  - **Riferimento**: ClickHouse Docker Image [Ultima consultazione: 2024-06-02].
  - **Ambiente**: `local`, `release`.

- **Grafana**

  - **Immagine**: grafana/grafana-oss:10.3.0;
  - **Riferimento**: Grafana Docker Image [Ultima consultazione: 2024-06-02].
  - **Ambiente**: `local`, `release`.

- **Apache Flink**
  - **Immagine**: flink:1.18.1-java17;
  - **Riferimento**: Flink Docker Image [Ultima consultazione: 2024-06-02].
  - **Ambiente**: `local`, `release`.

2.2 Linguaggi e formato dati

Nome	Versione	Descrizione	Impiego
Python	3.11.9	Linguaggio di programmazione ad alto livello, interpretato e multiparadigma.	Simulatore di sensori, *testing*, *script* per automatizzare il *deployment* dei *job* di Flink.
JSON	-	Formato di dati semplice da interpretare e generare, ampiamente utilizzato per lo scambio di dati tra applicazioni.	Configurazione *dashboard* Grafana.
YAML	-	Linguaggio di serializzazione dei dati leggibile sia per gli esseri umani sia per le macchine.	Docker Compose, provisioning Grafana e configurazione *alert*, file di *workflow* per le GitHub Actions.
SQL	Ansi SQL	Linguaggio di programmazione specificamente progettato per la gestione e la manipolazione di dati all'interno di sistemi di gestione di database.	*Query* e gestione database ClickHouse.

Nome	Versione	Descrizione	Impiego
TOML	1.0.0	Linguaggio di *markup* progettato per essere più leggibile e facile da scrivere rispetto ad altri formati di configurazione come JSON e YAML.	Configurazione e gestione dei sensori simulati.
Java	17	Linguaggio di programmazione ad alto livello, orientato agli oggetti.	Creazione di job per le aggregazioni dei dati di Flink.

Tabella 1: Linguaggi e formato dati

2.3 Librerie

Python		
Nome	**Versione**	**Impiego**
`confluent_avro`	1.8.0	Serializzazione dei dati in formato Avro.
`coverage`	7.5.1	Strumento per misurare la percentuale di linee di codice e rami coperti dai test.
`isodate`	0.6.1	Libreria per la manipolazione delle date e delle ore in formato ISO8601.
`kafka-python-ng`	2.2.2	Client Kafka per Python.
`ruff`	0.3.5	Libreria per l'analisi statica del codice.
`toml`	0.10.2	Libreria per effettuare il parsing dei file di configurazione in formato TOML.
Java		
`flink-streaming-java`	1.18.0	Utilizzo di DataStream API di Flink.
`flink-connector-kafka`	3.1.0-1.18	Connessione di Flink a Kafka.
`flink-clients`	1.18.0	Creazione di *job* di Flink.
`flink-java`	1.18.0	Creazione di *job* di Flink.
`flink-avro-confluent-registry`	1.18.0	Connessione di Flink a uno *schema registry* che utilizza Avro.

Nome	Versione	Impiego
`flink-shaded-guava`	31.1-jre-17.0	Gestione delle dipendenze di Flink.
`slf4j-simple`	1.7.36	Implementazione di SLF4J.
`lombok`	1.18.32	Libreria per la generazione di codice *boilerplate*.
`maven-assembly-plugin`	3.7.1	Plugin Maven per la creazione di un *fat jar*.

Tabella 2: Librerie utilizzate

2.4 Servizi

2.4.1 Redpanda

Redpanda è una piattaforma di streaming sviluppata in C++. Il suo obiettivo è fornire una soluzione leggera, semplice e performante, pensata per essere un'alternativa ad Apache Kafka. Viene utilizzato per disaccoppiare i dati provenienti dal simulatore.

- **Versione**: v23.3.11;
- **documentazione**: `https://docs.redpanda.com/current/home/` [Ultima consultazione: 2024-06-02].

2.4.1.1 Vantaggi

I vantaggi nell'utilizzo di questo strumento consistono in:

- ***performance***: è scritto in C++ e utilizza il *framework* Seastar, offrendo un'architettura *thread-per-core* ad alte prestazioni. Ciò permette di ottenere un'elevata *throughput* e latenze costantemente basse, evitando cambi di contesto e blocchi. Inoltre, è progettato per sfruttare l'*hardware* moderno, tra cui unità NVMe, processori *multi-core* e interfacce di rete ad alta velocità;
- **semplicità di configurazione**: oltre al *message broker*, contiene anche un *proxy* HTTP e uno *schema registry*;
- **minore richiesta di risorse**: rispetto ad Apache Kafka, richiede meno risorse per l'esecuzione in locale, rendendolo più adatto per l'esecuzione su *hardware* meno potente;

- **compatibilità con le API di Kafka**: è compatibile con le API di Apache Kafka, consentendo di utilizzare le librerie e gli strumenti esistenti;

2.4.1.2 Casi d'uso

Tra i casi d'uso di Redpanda si possono citare:

- ***streaming* di eventi**, permettendo la gestione e l'elaborazione di flussi di dati in tempo reale;
- ***data integration***, agisce come un intermediario flessibile e robusto per l'integrazione dei dati, consentendo la raccolta, il trasporto e la trasformazione dei dati provenienti da diverse sorgenti verso varie destinazioni;
- **elaborazione di *big data***, permette di gestire e processare enormi volumi di dati in modo efficiente e scalabile;
- **messaggistica *real time***, supporta la messaggistica in tempo reale tra applicazioni e sistemi distribuiti.

2.4.1.3 Impiego nel progetto

Il **broker** Redpanda gestisce i dati provenienti dai simulatori e li rende disponibili per i due consumatori. Inoltre, con lo *schema registry* integrato è possibile garantire la compatibilità tra i dati prodotti dai simulatori e i consumatori. I consumatori sono:

- Il **connector sink ClickHouse**, che salva i dati nelle tabelle di ClickHouse;
- **Apache Flink**, che elabora i dati in tempo reale.

2.4.2 ClickHouse

ClickHouse è un sistema di gestione di database colonnare *open-source* progettato per l'analisi dei dati in tempo reale e l'elaborazione di grandi volumi di dati.

- **Versione**: v24-alpine;
- **documentazione**: `https://clickhouse.com/docs/en/intro` [Ultima consultazione: 2024-06-02].

2.4.2.1 Vantaggi

I vantaggi nell'utilizzo di questo strumento consistono in:

- **alte prestazioni**, è progettato per eseguire query analitiche complesse in modo estremamente rapido;
- **scalabilità orizzontale**, può essere scalato orizzontalmente su più nodi, permettendo di gestire grandi volumi di dati;
- **elaborazione in tempo reale**, è in grado di gestire l'ingestione e l'elaborazione dei dati in tempo reale, rendendolo ideale per applicazioni che richiedono l'analisi immediata dei dati appena arrivano;
- **compressione efficiente**, utilizza algoritmi di compressione avanzati per ridurre lo spazio di archiviazione e migliorare l'efficienza I/O;
- **facilità di integrazione**, si integra facilmente con molti strumenti di visualizzazione dei dati e piattaforme di business intelligence come Grafana;
- **partizionamento e indici**, supporta il partizionamento dei dati e l'uso di indici per ottimizzare le query;

2.4.2.2 Casi d'uso

ClickHouse è utilizzato in una varietà di casi d'uso, tra cui:

- **analisi dei *log* e monitoraggio**, utilizzato per l'analisi e il monitoraggio dei log in tempo reale;
- ***business intelligence***, impiegato in applicazioni di BI per eseguire analisi approfondite dei dati aziendali, supportando la presa di decisioni basata sui dati;
- ***data warehousing***, funziona come data warehouse per memorizzare e analizzare grandi volumi di dati.

2.4.2.3 Impiego nel progetto

ClickHouse viene utilizzato per memorizzare i dati grezzi provenienti dai simulatori; attraverso il *connector sink* di Redpanda, i *record* pubblicati nei *topic* vengono salvati in tabelle di ClickHouse. Inoltre, tramite l'utilizzo di Materialized Views, vengono effettuate delle

semplici aggregazioni sui dati, come ad esempio la media oraria o giornaliera, le quali sono poi memorizzate in apposite tabelle, in modo da poterne monitorare l'andamento nel tempo.
Le aggregazioni più complesse che coinvolgono dati provenienti da sensori differenti sono invece effettuate utilizzando Apache Flink, come meglio descritto nella sezione 2.4.3.
ClickHouse si integra semplicemente con Grafana, attraverso l'utilizzo del plugin `datasource-clickhouse`, fornito da Grafana Labs.

2.4.3 Apache Flink

Apache Flink è un framework open-source per l'elaborazione dei dati in tempo reale e in *batch*. Sviluppato in Java e Scala, è progettato per gestire *data stream* in modo efficiente, consentendo l'elaborazione di grandi volumi di dati in tempo reale. Flink si distingue per la sua capacità di fornire elaborazione a bassa latenza, esecuzione *fault-tolerant* e scalabilità orizzontale.

- **Versione**: v1.18.1;
- **documentazione**: `https://flink.apache.org` [Ultima consultazione: 2024-06-25].

2.4.3.1 Vantaggi

- **Elaborazione a bassa latenza**: Flink è progettato per elaborare i dati in tempo reale con latenza estremamente bassa, rendendolo ideale per applicazioni che richiedono risposte rapide ai cambiamenti dei dati.
- **Fault Tolerance**: Flink utilizza una tecnologia chiamata *Stateful Stream Processing* che garantisce che lo stato dell'applicazione venga memorizzato in modo sicuro e possa essere recuperato in caso di guasti. Questo consente un'elaborazione affidabile e continua anche in presenza di errori hardware o software.
- **Scalabilità**: Flink può scalare orizzontalmente su cluster di grandi dimensioni, distribuendo il carico di lavoro tra molteplici nodi per gestire volumi di dati crescenti senza compromettere le prestazioni.
- **Modello di programmazione flessibile**: Flink offre due tipologie di API per l'elaborazione dei dati: DataStream API, utilizzato per l'elaborazione di flussi di dati non strutturati

in tempo reale, e Table API, un'astrazione di livello superiore per manipolare dati strutturati come tabelle, facilitando l'uso di operazioni simili a SQL;

- **Supporto per analisi complesse**: Flink fornisce potenti funzionalità di analisi come aggregazioni, join e *windowing*, che consentono di realizzare analisi complesse sui flussi di dati.

2.4.3.2 Casi d'uso

Tra i principali casi d'uso di Apache Flink si trovano:

- **Applicazioni *event-driven***: applicazioni *stateful* che elaborano eventi provenienti da uno o più flussi di eventi e reagiscono agli eventi in ingresso attivando calcoli, aggiornamenti di stato o azioni esterne;
- ***data analytics***: estrazione di informazioni e *insight* a partire dall'elaborazione dei dati grezzi, sia in *real time* che in modalità *batch*;
- ***data pipeline***: ad esempio per la costruzione di *Extract-transform-load* (ETL) o integrazione di dati provenienti da sorgenti differenti.

2.4.3.3 Impiego nel progetto

Vengono utilizzate le *Data Streaming API* per elaborare ed aggregare dati provenienti da sensori di tipologie differenti. Nello specifico, per ciascuno degli indici in seguito elencati è stato sviluppato un *job*; i dettagli implementativi di ciascuno di essi sono meglio discussi nella sezione .

- **Heat Index**: una misura che combina la temperatura dell'aria e l'umidità relativa per determinare la temperatura percepita dall'uomo. Questa misura riflette meglio il livello di disagio che una persona potrebbe sperimentare rispetto alla sola temperatura dell'aria;
- **European Air Quality Index**: un indice che misura la qualità dell'aria in base ai livelli di inquinanti atmosferici come il biossido di azoto, il biossido di zolfo, l'ozono e le particelle sospese;
- **Efficienza delle colonnine elettriche**

2.4.4 Grafana

È una potente piattaforma di visualizzazione dei dati progettata per creare, esplorare e condividere *dashboard* interattive che visualizzano metriche, *log* e altri dati di monitoraggio in tempo reale.

- **Versione**: v10.3.0;
- **documentazione**: `https://grafana.com/docs/grafana/v10.4/` [Ultima consultazione: 2024-06-02].

2.4.4.1 Vantaggi

- **Facilità d'uso**: possiede un'interfaccia intuitiva che rende facile la creazione e la gestione delle dashboard;
- **flessibilità**: La capacità di integrarsi con molteplici sorgenti dati e l'ampia gamma di plugin disponibili la rendono estremamente flessibile;
- **personalizzazione**: permette una personalizzazione completa delle dashboard, soddisfando ogni possibile necessità di visualizzazione dei dati;
- **gestione degli accessi**: offre funzionalità avanzate di gestione degli accessi e delle autorizzazioni, consentendo di controllare chi può accedere alle *dashboard* e quali azioni possono eseguire.

2.4.4.2 Casi d'uso

- **Monitoraggio delle infrastrutture**: utilizzato per monitorare le prestazioni e la disponibilità delle infrastrutture IT, inclusi server, database, servizi cloud e altro;
- **analisi delle performance delle applicazioni**: utilizzato per monitorare le prestazioni delle applicazioni e identificare eventuali problemi di prestazioni;
- **analisi delle serie temporali**: utilizzato per visualizzare e analizzare dati di serie temporali, come metriche di monitoraggio, log e dati di sensori;
- **business intelligence**: utilizzato per creare *dashboard* personalizzate per l'analisi dei dati aziendali e la visualizzazione delle metriche chiave.

2.4.4.3 Impiego nel progetto

- **Visualizzazione dei dati**: creazione *dashboard* interattive che visualizzano i dati salvati su ClickHouse;
- **notifiche superamento soglie**: invio di notifiche nel caso in cui vengano superate delle soglie prestabilite, che rappresentano situazioni di eventuale pericolo, forte disagio o disservizio per i cittadini.

3 Architettura di sistema

3.1 *Data processing architectures*

Le architetture di tipo *data processing* sono progettate per gestire l'*ingestion*, *processing* e memorizzazione di grandi quantità di dati. Esse permettono di analizzare e ottenere informazioni utili (*insight*) da questi dati, consentendo di ottimizzare i processi decisionali e migliorare le prestazioni aziendali. Esistono diverse architettura, ciascuna con le proprie caratteristiche e vantaggi. Tra le più comuni troviamo l'architettura *lambda* e l'architettura *kappa*.

3.1.1 Architettura *lambda*

L'architettura *lambda* è costituita dalle seguenti quattro componenti:

- **sorgente di dati**: responsabile dell'acquisizione dei dati grezzi da diverse sorgenti;
- ***batch layer***: responsabile dell'elaborazione e persistenza di dati storici in *batch* di grandi dimensioni; il suo scopo è fornire risposte complete e accurate, anche se con una latenza più elevata rispetto allo *speed layer*. Tale componente è tipicamente rappresentata da *framework* come Apache Hadoop o Apache Spark;
- ***speed* (*real-time*) *layer***: responsabile dell'elaborazione e persistenza di dati in tempo reale. I dati vengono elaborati in modo rapido e con una latenza molto bassa, fornendo tuttavia risposte elaborate rispetto al *batch layer*. Questa componente è tipicamente rappresentata da *framework* come Apache Storm o Apache Flink;
- ***serving layer***: responsabile della fornitura dei dati elaborati in modo veloce ed affidabile, indipendentemente dal *layer* di elaborazione utilizzato.

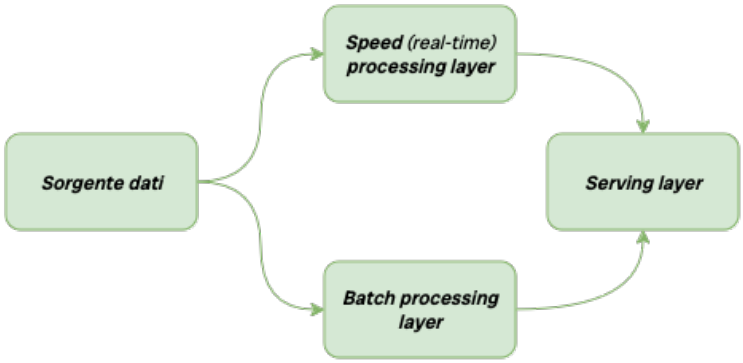

Figura 1: Architettura *lambda*

3.1.1.1 Vantaggi e svantaggi

L'architettura *lambda* offre diversi vantaggi, tra cui la **scalabilità orizzontale**, la **tolleranza ai guasti** e la **flessibilità**. Tuttavia, la presenza di due *layer* di elaborazione separati può portare a problemi di coerenza dei dati, duplicazione della logica di aggregazione e complessità aggiuntiva nella gestione del sistema. Inoltre, rispetto all'architettura *kappa*, l'architettura *lambda* può avere una latenza più elevata.

3.1.1.2 Casi d'uso

L'architettura *lambda* è particolarmente adatta per applicazioni che richiedono sia un'analisi sui dati in tempo reale che un'analisi storica.

3.1.2 Architettura *kappa*

L'architettura *kappa* è stata introdotta per semplificare l'architettura *lambda*, eliminando la necessità di gestire due *layer* di elaborazione separati per il *batch processing* e il *real-time processing*. Essa si divide in tre componenti principali:

- **sorgente di dati**: responsabile dell'acquisizione dei dati grezzi da diverse sorgenti;
- ***processing layer***: responsabile dell'elaborazione dei dati in tempo reale, senza la necessità di separare i dati in *batch* e *real-time*;
- ***serving layer***: responsabile della fornitura dei dati elaborati in modo veloce ed affidabile.

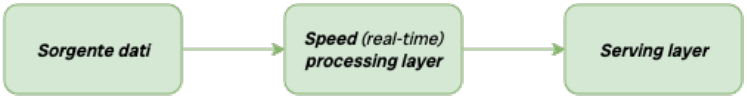

Figura 2: Architettura *kappa*

3.1.2.1 Vantaggi e svantaggi

L'architettura *kappa* offre diversi vantaggi, tra cui la **semplicità**, la **riduzione dei costi** e la **bassa latenza**. Tuttavia, può non essere adatta per applicazioni che richiedono un'analisi storica dei dati.

3.1.2.2 Casi d'uso

L'architettura *kappa* è particolarmente adatta per gli scenari in cui sono critici i dati in tempo reale e l'analisi dei dati storici è meno importante. Inoltre, semplifica notevolmente il processo di sviluppo e manutenzione dei sistemi di elaborazione dei dati.

3.2 Architettura scelta

Nello scenario del capitolato proposto da *SyncLab S.r.L.*, è critica l'analisi in tempo reale, in quanto i dati provenienti dai sensori IoT devono fornire informazioni sempre aggiornate ed eventualmente sollevare allarmi in caso di situazioni critiche. Inoltre, non è richiesta l'aggregazione storica di dati, dunque i vantaggi dell'architettura *lambda* non risultano utili per i nostri fini. Per soddisfare tali requisiti, è stata dunque scelta l'architettura *kappa*.

3.2.1 Componenti di sistema

All'interno del sistema da noi progettato sono dunque presenti le seguenti componenti:

- **sorgenti di dati**: costituite dal simulatore di sensori, il quale genera periodicamente i dati grezzi che in un contesto reale sarebbero provenienti dai sensori IoT;
- ***streaming layer***: gestisce il flusso di dati in tempo reale provenienti dai sensori. È composto da *Redpanda* e lo *Schema Registry*;
- ***processing layer***: elabora i dati in tempo reale per calcolare metriche e indici. È composto da *Apache Flink*;
- ***storage layer***: memorizza i dati elaborati per l'analisi e la visualizzazione. È composto da *ClickHouse*;
- ***data visualization layer***: fornisce un'interfaccia utente per visualizzare i dati elaborati. È composto da *Grafana*.

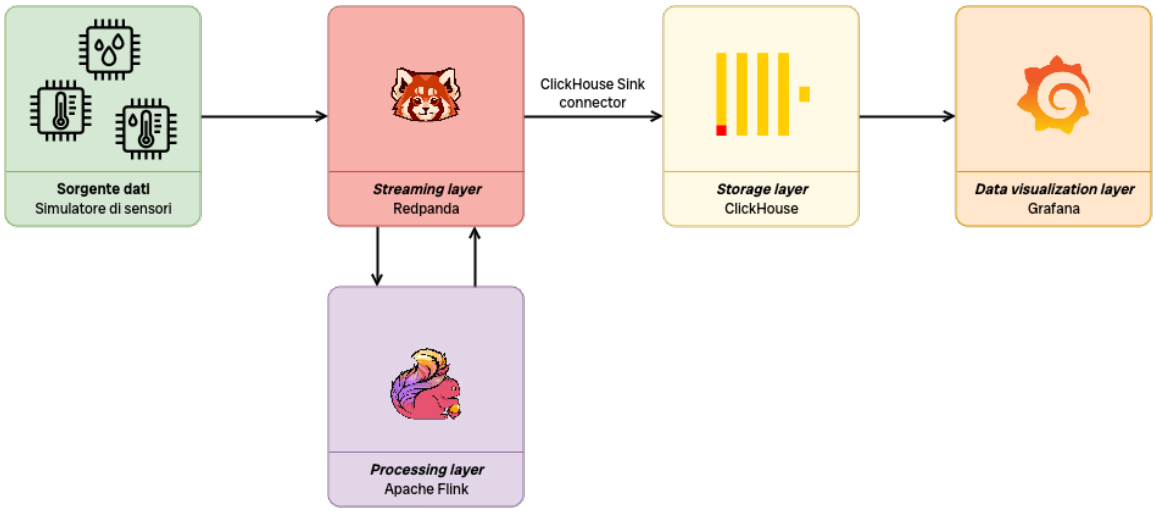

Figura 3: Componenti di sistema

3.3 Flusso di dati

3.4 Architettura dei simulatori

3.4.1 Modulo producers

3.4.2 Modulo serializers

3.4.3 Modulo simulators

- Classe astratta `Simulator`: rappresenta un simulatore generico. Contiene i seguenti metodi e attributi:
  - Metodi:
    - `start() None [public]`: avvia il simulatore;
    - `stop() None [public]`: ferma il simulatore;
    - `stream() Iterable[RawData]`: metodo astratto che verrà implementato dalle sottoclassi per generare i dati;
  - Attributi:
    - `sensor_uuid UUID [public]`: identificativo univoco del sensore;
    - `sensor_name str [public]`: nome del sensore;
    - `latitude float [public]`: latitudine del sensore;
    - `longitude float [public]`: longitudine del sensore;

    * `generation_delay timedelta [public]`: tempo effettivamente atteso tra la generazione di un dato e il successivo;
    * `points_spacing timedelta [public]`: distanza temporale dei dati generati;
    * `limit int [public]`: limite di dati da produrre;
    * `timestamp datetime [public]`: data da cui iniziare a produrre dati;
    * `running bool [private]`: flag che indica se il simulatore è in esecuzione.

- Classe concreta `AirQualitySimulator`: estende `Simulator` e rappresenta un simulatore per la generazione di dati relativi alla qualità dell'aria. Contiene i seguenti metodi e attributi:
  - Metodi:
    * `stream() Iterable[AirQualityRawData] [public]`: genera i dati relativi alla qualità dell'aria;
  - Attributi aggiuntivi rispetto a `Simulator`:
    * `o3_coefficient float [private]`: coefficiente generato con distribuzione uniforme tra -50 e 50 che viene sommato ogni volta ai valori generati per far si che varino a seconda del sensore che li ha generati;
    * `no2_coefficient float [private]`: coefficiente generato con distribuzione uniforme tra -50 e 50 che viene sommato ogni volta ai valori generati per far si che varino a seconda del sensore che li ha generati;
    * `so2_coefficient float [private]`: coefficiente generato con distribuzione uniforme tra -50 e 50 che viene sommato ogni volta ai valori generati per far si che varino a seconda del sensore che li ha generati;
    * `pm10_coefficient float [private]`: coefficiente generato con distribuzione uniforme tra -50 e 50 che viene sommato ogni volta ai valori generati per far si che varino a seconda del sensore che li ha generati;
    * `pm25_coefficient float [private]`: coefficiente generato con distribuzione uniforme tra -50 e 50 che viene sommato ogni volta ai valori generati per far si che varino a seconda del sensore che li ha generati.
  - Funzioni di supporto:
    * `daily_variation(timestamp: datetime) float [private]`: genera una variazione giornaliera in base alla data e ora passata come argomento;
    * `weekly_variation(timestamp: datetime) float [private]`: genera una variazione settimanale in base alla data e ora passata come argomento;

    * `seasonal_variation(timestamp: datetime) float [private]`: genera una variazione stagionale in base alla data e ora passata come argomento;
    * `sinusoidal_value(timestamp: datetime) float [private]`: genera un valore sinusoidale in base alla data e ora passata come argomento, richiamando le funzioni di variazione giornaliera, settimanale e stagionale e sommandole tra di loro.

- Classe concreta `RecyclingPointSimulator`: estende `Simulator` e rappresenta un simulatore per la generazione di dati relativi alle isole ecologiche. Contiene i seguenti metodi e attributi:
  - Metodi:
    * `stream() Iterable[RecyclingPointRawData] [public]`: genera i dati relativi alle isole ecologiche;
    * `calculate_fill_rate() None [private]`: calcola la velocità di riempimento dell'isola ecologica dall'ultimo dato generato;
    * `filling() float [private]`: calcola la percentuale di riempimento dell'isola ecologica;
  - Attributi aggiuntivi rispetto a `Simulator`:
    * `last_value float [private]`: percentuale di riempimento dell'isola ecologica dell'ultimo dato generato;
    * `prev_timestamp datetime [private]`: data e ora dell'ultimo dato generato;
    * `fill_rate float [private]`: velocità di riempimento dell'isola ecologica;
    * `emptying_hours List[Tuple[int, int]] [private]`: lista di giorni della settimana in cui l'isola ecologica verrà svuotata;
    * `noise_limit float [private]`: numero casuale generato con distribuzione uniforme che rappresenta il rumore;
    * `partial_emptying_chance float [private]`: probabilità che l'isola ecologica venga svuotata parzialmente;
    * `partial_emptying_max_percentage float [private]`: percentuale di rifiuti rimanente dopo uno svuotamento parziale;
  - Funzioni di supporto:
    * `generate_emptying_hours() List[Tuple[int, int]] [private]`: genera una lista di giorni della settimana in cui l'isola ecologica verrà svuotata;

- Classe concreta `TemperatureSimulator`: estende `Simulator` e rappresenta un simulatore per la generazione di dati relativi alla temperatura. Contiene i seguenti metodi e attributi:
  - Metodi:
    - `stream() Iterable[TemperatureRawData] [public]`: genera i dati relativi alla temperatura;
  - Funzioni di supporto:
    - `seasonal_coefficient(timestamp: datetime) float [private]`: genera un coefficiente stagionale in base alla data e ora passata come argomento;
    - `thermal_excursion(timestamp: datetime) float [private]`: genera un'escursione termica giornaliera in base alla data e ora passata come argomento;
    - `sinusoidal_value(timestamp: datetime) float [private]`: genera un valore sinusoidale in base alla data e ora passata come argomento, richiamando le funzioni di variazione stagionale e di escursione termica.
- Classe concreta `TrafficSimulator`: estende `Simulator` e rappresenta un simulatore per la generazione di dati relativi al traffico. Contiene i seguenti metodi e attributi:
  - Metodi:
    - `stream() Iterable[TrafficRawData] [public]`: genera i dati relativi al traffico;
  - Costanti:
    - `VEHICLES_MULTIPLICATIVE_FACTOR int [private]`: costante moltiplicativa per il numero di veicoli;
    - `SPEED_MULTIPLICATIVE_FACTOR int [private]`: costante moltiplicativa per la velocità media.
  - Funzioni di supporto:
    - `multimodal_gauss_value(x: float, modes: list[tuple[float, float]]) float [private]`: genera un valore gaussiano multimodale in base ai secondi passati dall'inizio della giornata e ai modi passati come argomento;

3.4.4 Modulo models

3.4.4.1 config

Contiene le classi e le funzioni utilizzate per leggere ed effettuare il parsing del file di configurazione *sensor.toml*, utilizzato per costruire i vari simulatori.

3.4.4.1.1 Classi e metodi

- Classe concreta `EnvConfig`: legge le variabili d'ambiente e nel caso siano assenti ma ammettano valore di default lo assegna.
  - Metodi
    - `get_or_throw(key: str) str [private]`: ritorna il valore della variabile d'ambiente con quella chiave o se non esiste lancia un'eccezione;
    - `get_or_none(key: str) str | None [private]`: ritorna il valore della variabile d'ambiente con quella chiave o se non esiste ritorna `None`;
    - `bootstrap_server() str [public]`: ritorna concatenati i valori delle variabili d'ambiente `KAFKA_HOST` e `KAFKA_PORT` separati da il simbolo ":";
  - Attributi
    - `kafka_host str [public]`: host del broker Kafka;
    - `kafka_port str [public]`: porta del broker Kafka;
    - `log_level str [public]`: livello di log;
    - `max_block_ms str [public]`: massimo numero di millisecondi che il producer kafka può rimanere bloccato durante `send()`.
- Classe concreta `SensorConfig`: legge il file di configurazione *sensor.toml* e ne effettua il parsing.
  - Attributi
    - `uuid UUID [public]`: identificativo univoco del sensore;
    - `limit int [public]`: limite di dati da produrre;
    - `begin_date datetime [public]`: data da cui iniziare a produrre dati. Opzionale e nel caso non sia presente viene presa la data attuale;
    - `latitude float [public]`: latitudine del sensore;
    - `longitude float [public]`: longitudine del sensore;
    - `generation_delay timedelta [public]`: tempo effettivamente atteso tra la generazione di un dato e il successivo. Espresso in formato ISO 8601 e nel caso non sia conforme viene sollevata un'eccezione;
    - `points_spacing timedelta [public]`: distanza temporale dei dati generati. Espresso in formato ISO 8601 e nel caso non sia conforme viene sollevata un'eccezione.

3.4.4.2 raw_data

- Classe astratta `RawData`: rappresenta un dato grezzo generato da un sensore. Contiene i seguenti metodi e attributi:
  - Metodi
    - `accept(visitor: SerializerVisitor) Dict [str,any] [public]`: metodo astratto che verrà utilizzato dalle sottoclassi per poter implementare il pattern Visitor;
    - `topic() str [public]`: ritorna il nome del topic Kafka a cui il dato deve essere inviato;
    - `subject() str [public]`: ritorna un identificativo associato allo schema del raw data. Nello specifico ritorna il nome del topic seguito da "-value";
  - Attributi
    - `sensor_uuid UUID [public]`: identificativo univoco del sensore;
    - `sensor_name str [public]`: nome del sensore;
    - `latitude float [public]`: latitudine del sensore;
    - `longitude float [public]`: longitudine del sensore;
    - `timestamp datetime [public]`: data e ora della misurazione;
- Classe concreta `AirQualityRawData`: estende `RawData` e rappresenta un dato grezzo generato da un sensore di qualità dell'aria. Contiene i seguenti metodi e attributi:
  - Metodi
    - `accept(visitor: SerializerVisitor) Dict [str,any] [public]`: implementa il metodo della classe padre, invocando il metodo per la serializzazione della classe AirQualityRawData;
    - `topic() str [public]`: ritorna "air_quality".
  - Attributi aggiuntivi rispetto a `RawData`:
    - `pm25 float [public]`: quantità di particolato atmosferico con un diametro aerodinamico inferiore o uguale a 2.5 micrometri e misurato in $\mu g/m^3$;
    - `pm10 float [public]`: quantità di particolato atmosferico con un diametro aerodinamico inferiore o uguale a 10 micrometri e misurato in $\mu g/m^3$;
    - `no2 float [public]`: concentrazione di biossido di azoto misurata in $\mu g/m^3$;
    - `o3 float [public]`: concentrazione di ozono misurata in $\mu g/m^3$;
    - `so2 float [public]`: concentrazione di biossido di zolfo misurata in $\mu g/m^3$.

- Classe concreta `RecyclingPointRawData`: estende `RawData` e rappresenta un dato grezzo generato da un sensore di isola ecologica. Contiene i seguenti metodi e attributi:
  - Metodi:
    - `accept(visitor: SerializerVisitor) Dict [str,any] [public]`: implementa il metodo della classe padre, invocando il metodo per la serializzazione della classe RecyclingPointRawData;
    - `topic() str [public]`: ritorna "recycling_point".
  - Attributi aggiuntivi rispetto a `RawData`:
    - `filling float [public]`: percentuale di riempimento dell'isola ecologica.
- Classe concreta `TemperatureRawData`: estende `RawData` e rappresenta un dato grezzo generato da un sensore di temperatura. Contiene i seguenti metodi e attributi:
  - Metodi:
    - `accept(visitor: SerializerVisitor) Dict [str,any] [public]`: implementa il metodo della classe padre, invocando il metodo per la serializzazione della classe TemperatureRawData;
    - `topic() str [public]`: ritorna "temperature".
  - Attributi aggiuntivi rispetto a `RawData`:
    - `value float [public]`: valore della temperatura rilevata espressa in °C.
- Classe concreta `TrafficRawData`: estende `RawData` e rappresenta un dato grezzo generato da un sensore di traffico. Contiene i seguenti metodi e attributi:
  - Metodi:
    - `accept(visitor: SerializerVisitor) Dict [str,any] [public]`: implementa il metodo della classe padre, invocando il metodo per la serializzazione della classe TrafficRawData;
    - `topic() str [public]`: ritorna "traffic".
  - Attributi aggiuntivi rispetto a `RawData`:
    - `vehicles int [public]`: numero di veicoli rilevati;
    - `avg_speed float [public]`: velocità media dei veicoli rilevati espressa in km/h.

3.4.5 Progettazione - Panoramica UML

3.5 Redpanda

3.5.1 Kafka topic

I Kafka topic sono categorie o canali di messaggi all'interno di *Redpanda*. Un topic in Kafka è come una cassetta postale virtuale o una categoria di messaggi in cui i dati vengono pubblicati dai produttori e letti dai consumatori.

3.5.1.1 Formato dei dati e Schema Registry

7Last ha scelto di adottare lo standard Avro per i messaggi scambiati tra i produttori e i consumatori. Avro è un sistema di serializzazione dati che offre un modo efficiente per rappresentare dati complessi in un formato binario, rendendoli adatti per il trasporto su rete o per la persistenza su disco. Uno dei principali vantaggi di Avro è la possibilità di definire uno schema per i dati, che viene incluso nei dati stessi. Ecco una panoramica del formato dei messaggi Avro:

- **schema**: definito in formato JSON e descrive la struttura dei dati. Include informazioni come il tipo di ogni campo e la loro posizione all'interno della struttura dei dati;
- **serializzazione binaria**: Avro serializza i dati in un formato binario compatto, che rende efficiente il trasporto e la memorizzazione dei dati. Utilizza un'organizzazione binaria che incorpora lo schema dei dati insieme ai dati stessi. Questo significa che non c'è bisogno di includere esplicitamente il tipo di dato per ogni campo, poiché lo schema fornisce questa informazione;
- **compatibilità**: è progettato per supportare l'evoluzione dei dati nel tempo. Puoi aggiungere nuovi campi, rimuovere campi esistenti o modificare il tipo di dati di un campo mantenendo la compatibilità con le versioni precedenti degli schemi;
- **condivisione dello schema**: supporta la condivisione dello schema tramite un registro dello schema (Schema Registry). Questo consente di registrare gli schemi Avro utilizzati nel sistema in un registro centralizzato, in modo che i produttori e i consumatori possano recuperare gli schemi necessari quando ne hanno bisogno.

3.6 Flink - Processing Layer

3.6.1 Introduzione

È un framework di elaborazione dati distribuito e open-source che si distingue per la sua capacità di gestire sia dati di flusso in tempo reale che dati batch. Una delle sue caratteristiche principali è la capacità di gestire dati in tempo reale con latenze molto basse. Ciò significa che può elaborare i dati man mano che arrivano, consentendo alle applicazioni di reagire istantaneamente ai cambiamenti nell'input. Questa caratteristica è particolarmente importante per le applicazioni che richiedono analisi in tempo reale, come il monitoraggio di sensori, il rilevamento di anomalie o la personalizzazione di contenuti.

3.6.2 Componenti Flink & Processing Layer

È costituito da diverse componenti fondamentali che lavorano insieme per consentire l'elaborazione efficiente e scalabile dei dati in tempo reale e batch. Queste componenti formano il cuore del sistema Flink e forniscono le basi per la sua potente capacità di elaborazione dei dati.

- **JobManager**: è il componente centrale di Flink responsabile della pianificazione e del coordinamento dei job di elaborazione dei dati. Gestisce il flusso di lavoro complessivo, assegnando i task ai TaskManager per l'esecuzione;
- **TaskManager**: è responsabile dell'esecuzione effettiva delle operazioni di elaborazione dei dati, eseguendo i task assegnati loro dal JobManager e gestendo il caricamento, l'elaborazione e la distribuzione dei dati all'interno del cluster;
- **Processing layer**: è responsabile dell'esecuzione delle operazioni di elaborazione dei dati all'interno del cluster distribuito. Questa layer sfrutta le risorse di calcolo e memorizzazione disponibili nei nodi del cluster per eseguire le operazioni di trasformazione, aggregazione, filtraggio e altro ancora sui dati in ingresso. Utilizzando un modello di esecuzione distribuita, la Processing Layer di Flink è in grado di scalare orizzontalmente per gestire grandi volumi di dati e carichi di lavoro ad alta intensità computazionale.

3.6.3 Processing layer data-flow

1. **Acquisizione dei dati**;

2. **partizione e distribuzione**: i dati vengono divisi in parti più piccole e distribuiti tra i nodi del cluster per massimizzare l'utilizzo delle risorse;
3. **pianificazione dei task**: il JobManager assegna compiti di elaborazione ai TaskManager basandosi sullo stato del cluster e sull'ottimizzazione delle prestazioni;
4. **esecuzione dei task**: i TaskManager eseguono i compiti assegnati in parallelo, elaborando i dati in base alla logica definita nell'applicazione Flink;
5. **scambio e movimento dei dati**: i dati possono essere scambiati e spostati tra i nodi del cluster per supportare operazioni complesse come il join o l'aggregazione;
6. **persistenza e output**: una volta elaborati, i risultati vengono eventualmente salvati o inviati ad altre destinazioni per l'analisi o l'utilizzo successivo.

3.6.4 *Job*

3.6.4.1 Heat Index

A partire dai dati rilevati dai sensori di temperatura e umidità; esso consente di stimare la percezione della temperatura da parte dell'essere umano. Nella configurazione del Simulatore, oltre a posizione e identificativo per ciascun sensore, è possibile specificare un `group_name`, una stringa che identifica il gruppo o zona di appartenenza; si suppone che sensori situati in posizioni geografiche vicine abbiano lo stesso `group_name`. Il *job* calcola prima separatamente la temperatura e l'umidità media per finestre di un'ora, aggregando i dati provenienti da sensori dello stesso gruppo. Successivamente con i valori ottenuti computa lo Heat Index, utilizzando la formula empirica ideata da Blazejczyk [Ultima consultazione 2024-06-25]. Nel risultato finale, oltre al valore dell'Heat Index, vengono restituiti anche i valori di temperatura e umidità medi, il centro di massa del gruppo di sensori (utilizzando la formula Haversine [Ultima consultazione 2024-06-25] per il calcolo della distanza) e la distanza dal centro di massa al sensore più lontano. Questi ultimi due dati sono impiegati in una mappa interattiva su Grafana per poter disegnare un cerchio, rappresentante la zona di influenza del gruppo di sensori.

3.6.4.2 European Air Quality Index

3.6.4.3 Efficienza delle colonnine elettriche

3.7 Database ClickHouse

3.7.1 Funzionalità utilizzate

3.7.1.1 Materialized View

Sono una potente funzionalità di *ClickHouse* per migliorare le prestazioni delle query e semplificare l'analisi dei dati. In sostanza, una materialized view è una vista pre-calcolata o una copia di una query, memorizzata fisicamente su disco in forma tabellare. Ciò consente di evitare il calcolo ripetuto dei risultati della query ogni volta che viene eseguita.

Documentazione

`https://clickhouse.com/docs/en/guides/developer/cascading-materialized-views`
[Ultima consultazione 2024-06-05]

Utilizzi

- **Aggregazioni pre-calcolate**: le materialized view possono essere utilizzate per memorizzare i risultati di aggregazioni complesse, come somme, medie, conteggi, ecc., in modo che non debbano essere calcolati ogni volta che viene eseguita una query;
- **rapporti pre-calcolati**: possono essere utilizzate per memorizzare i risultati di query complesse o di rapporti, in modo che i risultati siano immediatamente disponibili senza dover eseguire la query ogni volta;
- **join ottimizzati**: possono essere utilizzate per memorizzare i risultati di join complessi tra più tabelle, in modo che i risultati siano immediatamente disponibili senza dover eseguire il join ogni volta;
- **filtraggio e selezione efficiente**: possono essere utilizzate per filtrare e selezionare dati in base a criteri specifici, migliorando le prestazioni delle query che richiedono l'accesso solo a una parte dei dati.

3.7.1.2 MergeTree

MergeTree è uno dei principali motori di archiviazione di ClickHouse, progettato per gestire grandi volumi di dati e fornire elevate prestazioni di lettura e scrittura. È particolarmente adatto per applicazioni in cui i dati vengono aggiunti in modo incrementale e le query vengono eseguite su intervalli di tempo specifici. Le caratteristiche principali sono:

- **partizionamento**, in cui i dati vengono partizionati in base a una colonna di data o di tempo, in modo che i dati più recenti siano memorizzati in partizioni separate e possano essere facilmente eliminati o archiviati;
- **ordine dei dati**, dove i dati vengono ordinati in base a una colonna di ordinamento, in modo che i dati siano memorizzati in modo sequenziale e possano essere letti in modo efficiente;
- **indice primario**, tramite il quale i dati vengono indicizzati in base a una colonna di chiave primaria, in modo che le query di ricerca e di join siano veloci ed efficienti;
- **merging dei dati**, in questo modo i dati vengono uniti in modo incrementale in background, in modo che le query di aggregazione e di analisi siano veloci ed efficienti;
- **compressione**, i dati vengono compressi in modo efficiente per ridurre lo spazio di archiviazione e migliorare le prestazioni di lettura e scrittura;
- **replica e distribuzione**, i dati possono essere replicati e distribuiti su più nodi per garantire l'affidabilità e la disponibilità del sistema.

Documentazione

`https://clickhouse.com/docs/en/engines/table-engines/mergetree-family/mergetree`
[Ultima consultazione 2024-06-05]

Utilizzi

- **Analisi dei dati storici**: i dati storici vengono memorizzati in tabelle MergeTree per consentire l'analisi e l'elaborazione dei dati storici;
- **applicazioni di business intelligence**: i dati vengono memorizzati in tabelle MergeTree per consentire l'analisi e la generazione di report per le applicazioni di business intelligence;

- **log e monitoraggio**: i dati di log e di monitoraggio vengono memorizzati in tabelle MergeTree per consentire l'analisi e il monitoraggio delle attività di sistema.

3.7.2 Trasferimento dati tramite Materialized View

Le Materialized View in ClickHouse sono viste che memorizzano fisicamente i risultati di una query specifica in modo da permettere un accesso rapido e efficiente ai dati pre-elaborati. Quando vengono create, le Materialized View eseguono la query definita e archiviano i risultati in una struttura di dati ottimizzata per l'accesso veloce. Questo consente di evitare il calcolo ripetuto dei risultati della query ogni volta che viene eseguita, migliorando notevolmente le prestazioni complessive del sistema. I vantaggi derivanti da questo approccio sono molteplici, tra questi troviamo:

- **prestazioni ottimizzate**: grazie alla memorizzazione fisica dei risultati delle query, le Materialized View consentono un accesso rapido ai dati pre-elaborati, riducendo i tempi di risposta delle query complesse;
- **riduzione del carico di lavoro**: trasferendo i dati pre-elaborati in Materialized View, si riduce il carico di lavoro sul sistema sorgente, consentendo una maggiore scalabilità e riducendo il rischio di sovraccarico del sistema durante le operazioni di estrazione dei dati;
- **sempre aggiornate**: possono essere progettate per aggiornarsi automaticamente in risposta alle modifiche nei dati sottostanti, garantendo che i risultati siano sempre aggiornati e coerenti con lo stato attuale dei dati;
- **semplificazione dell'architettura**: è possibile semplificare l'architettura complessiva del sistema eliminando la necessità di eseguire query complesse e costose ogni volta che si accede ai dati.

3.7.3 Misurazioni isole ecologiche

Di seguito viene riportata la configurazione della tabella per le misurazioni delle isole ecologiche. Le misurazioni includono:

- **sensor_uuid**: identificativo univoco del sensore (formato UUID);
- **sensor_name**: nome del sensore (formato String);
- **timestamp**: data e ora della misurazione (formato DateTime64);

- **latitude**: latitudine del sensore (formato Float64);
- **longitude**: longitudine del sensore (formato Float64);
- **filling_value**: percentuale di riempimento dell'isola ecologica (formato Float32).

3.7.4 Misurazioni temperatura

Di seguito viene riportata la configurazione della tabella per le misurazioni della temperatura. Le misurazioni includono:

- **sensor_uuid**: identificativo univoco del sensore (formato UUID);
- **sensor_name**: nome del sensore (formato String);
- **timestamp**: data e ora della misurazione (formato DateTime64);
- **value**: valore della temperatura rilevata (formato Float32);
- **latitude**: latitudine del sensore (formato Float64);
- **longitude**: longitudine del sensore (formato Float64);

3.7.5 Misurazioni traffico

Di seguito viene riportata la configurazione della tabella per le misurazioni della traffico. Le misurazioni includono:

- **sensor_uuid**: identificativo univoco del sensore (formato UUID);
- **sensor_name**: nome del sensore (formato String);
- **timestamp**: data e ora della misurazione (formato DateTime64);
- **latitude**: latitudine del sensore (formato Float64);
- **longitude**: longitudine del sensore (formato Float64);
- **vehicles**: numero di veicoli rilevati (formato Int32);
- **avg_speed**: velocità media del traffico (formato Float32).

3.8 Grafana

Grafana è uno strumento di analisi e monitoraggio che permette di visualizzare dati provenienti da una varietà di fonti. È sviluppato principalmente in Go e Typescript ed è noto per la sua capacità di creare dashboard personalizzabili e intuitive.

3.8.1 Dashboard

3.8.2 ClickHouse datasource plugin

Il plugin ClickHouse per Grafana è un'implementazione che consente di utilizzare ClickHouse come fonte di dati per Grafana. Questo plugin facilita la connessione e l'interrogazione dei dati archiviati in ClickHouse direttamente da Grafana, permettendo di creare dashboard dinamiche e interattive.

Documentazione

`https://grafana.com/grafana/plugins/grafana-clickhouse-datasource/`
[Ultima consultazione 2024-06-05]

3.8.2.1 Configurazione del Datasource

La configurazione grafana/provisioning/datasources/default.yaml

3.8.3 Variabili Grafana

3.8.3.1 Documentazione

`https://grafana.com/docs/grafana/latest/dashboards/variables/`
[Ultima consultazione 2024-06-05]

Variabili nella dashboard principale

Le variabili presenti nella dashboard principale sono:

- **tipo sensore**: permette di selezionare il tipo di sensore da visualizzare (temperatura, traffico, isola ecologica...);
- **nome sensore**: permette di selezionare il nome del sensore da visualizzare (ad esempio sensore1, sensore2, ecc.);

Variabili nella dashboard dettagliata

Le variabili presenti nelle dashboard dettagliate sono:

- **nome sensore**: permette di selezionare il nome del sensore da visualizzare (es. sensore1, sensore2, ecc.);

3.8.4 Grafana Alerts

Sono una funzionalità che permettono di definire, configurare e gestire avvisi basati su condizioni specifiche rilevate nei dati monitorati. Questi avvisi consentono agli utenti di essere informati tempestivamente su eventuali problemi o cambiamenti critici nei loro sistemi, applicazioni o infrastrutture.

Documentazione

`https://grafana.com/docs/grafana/latest/alerting/` [Ultima consultazione 2024-06-05]

3.8.4.1 Configurazione delle regole di alert

Definiscono le condizioni che devono essere soddisfatte per attivare un alert. Gli eventi che generano un alert sono:

- temperatura maggiore di 40°C per più di 30 minuti;
- isola ecologica piena al 100% per più di 24 ore;
- superamento dell'indice 3 dell'EAQI (indice di qualità dell'aria);
- livello di precipitazioni superiore a 10 mm in 1 ora.

Gli alert possono possedere tre diversi tipi di stati:

- **normal**, indica che l'alert non è attivo perché le condizioni definite per l'attivazione dell'avviso non sono soddisfatte;
- **pending**, indica che le metriche monitorate stanno iniziando a deviare dalle condizioni normali ma non hanno ancora soddisfatto completamente le condizioni per attivare l'alert;
- **firing**, significa che le condizioni definite per l'avviso sono state soddisfatte e l'alert è attivo.

3.8.4.2 Configurazione canale di notifica

Per configurare un canale di notifica è necessario:

1. nel menù di sinistra, cliccare sull'icona "Alerting";
2. selezionare la voce "Notification channels";
3. cliccare sul pulsante "Add channel" per aggiungere un nuovo canale di notifica;
4. selezionare il tipo di canale di notifica desiderato tra quelli disponibili;
5. configurare le impostazioni del canale di notifica in base alle proprie esigenze;
6. cliccare sul pulsante "Save" per salvare le impostazioni del canale di notifica.

7Last ha deciso di rendere risponibile il server *Discord* configurato a questo scopo e raggiungibile a questo link:

`https://discord.com/channels/1214553333113556992/1241974479345942568`

3.8.5 Altri plugin

3.8.5.1 Orchestra Cities Map plugin

Progettato per facilitare la visualizzazione e l'analisi dei dati geospaziali all'interno di piattaforme di pianificazione urbana e sviluppo territoriale.
Le principali funzionalità offerte da questo plugin sono:

- **visualizzazione dei dati geospaziali**: consente agli utenti di visualizzare dati geografici, come mappe, strati di dati GIS (Geographic Information System), punti di interesse e altre informazioni territoriali;
- **interfaccia interattiva**: offre un'interfaccia utente intuitiva e interattiva che consente agli utenti di esplorare e interagire con i dati geospaziali in modo dinamico;
- **personalizzazione**: offre opzioni di personalizzazione per adattarsi alle esigenze specifiche dell'utente o dell'applicazione;
- **analisi dei dati**: oltre alla semplice visualizzazione dei dati geospaziali, il plugin può anche supportare funzionalità avanzate di analisi dei dati, come l'identificazione di cluster, la creazione di heatmap e l'esecuzione di analisi spaziali per identificare tendenze o pattern significativi nei dati territoriali;

- **integrazione**: è progettato per integrarsi facilmente con altre componenti dell'ecosistema Orchestra Cities e con altre piattaforme software di pianificazione urbana e sviluppo territoriale.

Documentazione

`https://grafana.com/grafana/plugins/orchestracities-map-panel/?tab=installation`
[Ultima consultazione 2024-06-05]

4 Architettura di deployment

Per implementare ed eseguire l'intero stack tecnologico e i livelli del modello architetturale, viene creato un ambiente *Docker* che riproduce la suddivisione e la distribuzione dei servizi. In particolare, per l'ambiente di produzione, sono stati creati i seguenti container:

- **Data feed**
  - Container: **Simulator**;
  - Descrizione: simula la generazione di dati;
- **Streaming layer**
  - Container: **Redpanda**;
  - Descrizione: definisce il flusso di dati in tempo reale;
  - Componenti di supporto: schema registry;
  - Porta: 18082.
- **Processing Layer**
  - Container: **Flink**;
  - Descrizione: pianifica, assegna e coordina l'esecuzione dei task di elaborazione dei dati su un cluster di nodi, garantendo prestazioni elevate, scalabilità e affidabilità nell'elaborazione dei dati.
- **Storage Layer**
  - Container: **Clickhouse**;
  - Descrizione: memorizza i dati;
  - Porta: 8123.
- **Data Visualization Layer**
  - Container: **Grafana**;
  - Descrizione: visualizza i dati;
  - Porta: 3000.

5 Requisiti

5.1 Requisiti funzionali

Codice	Importanza	Stato	Descrizione
RF-1	Obbligatorio	Soddisfatto	La parte *IoT* dovrà essere simulata attraverso tool di generazione di dati casuali che tuttavia siano verosimili.
RF-2	Obbligatorio	Soddisfatto	Il sistema dovrà permettere la visualizzazione dei dati in tempo reale.
RF-3	Obbligatorio	Soddisfatto	Il sistema dovrà permettere la visualizzazione dei dati storici.
RF-4	Obbligatorio	Soddisfatto	L'utente deve poter accedere all'applicativo senza bisogno di autenticazione.
RF-5	Obbligatorio	Soddisfatto	L'utente dovrà poter visualizzare su una mappa la posizione geografica dei sensori.
RF-6	Obbligatorio	Soddisfatto	I tipi di dati che il sistema dovrà visualizzare sono: temperatura, umidità, qualità dell'aria, precipitazioni, traffico, stato delle colonnine di ricarica, stato di occupazione dei parcheggi, stato di riempimento delle isole ecologiche e livello di acqua.
RF-7	Obbligatorio	Soddisfatto	I dati dovranno essere salvati su un database OLAP.
RF-8	Obbligatorio	Soddisfatto	I sensori di temperatura rilevano i dati in gradi Celsius
RF-9	Obbligatorio	Soddisfatto	I sensori di umidità rilevano la percentuale di umidità nell'aria.

Codice	Importanza	Stato	Descrizione
RF-10	Obbligatorio	Soddisfatto	I sensori livello acqua rilevano il livello di acqua nella zona di installazione
RF-11	Obbligatorio	Soddisfatto	I dati provenienti dai sensori dovranno contenere i seguenti dati: id sensore$_G$, data, ora e valore.
RF-12	Obbligatorio	Soddisfatto	Sviluppo di componenti quali widget$_G$ e grafici per la visualizzazione dei dati nelle dashboard$_G$.
RF-13	Obbligatorio	Soddisfatto	Il sistema deve permettere di visualizzare una dashboard$_G$ generale con tutti i dati dei sensori.
RF-14	Obbligatorio	Soddisfatto	Il sistema deve permettere di visualizzare una dashboard$_G$ specifica per ciascuna categoria di sensori.
RF-15	Obbligatorio	Soddisfatto	Nella dashboard$_G$ dei dati grezzi dovranno essere presenti: una mappa interattiva, un widget$_G$ con il conteggio totale dei sensori divisi per tipo, una tabella contente tutti i sensori e la data in cui essi hanno trasmesso l'ultima volta. Inoltre verranno mostrate delle tabelle con i dati filtrabili suddivisi per sensore$_G$ e un grafico time series$_G$ con tutti i dati grezzi.

Codice	Importanza	Stato	Descrizione
RF-16	Obbligatorio	Soddisfatto	Nella dashboard$_G$ della temperatura dovranno essere visualizzati: un grafico time series$_G$, una mappa interattiva, la temperatura media, minima e massima di un certo periodo di tempo, la temperatura in tempo reale e la temperatura media per settimana e mese.
RF-17	Obbligatorio	Soddisfatto	Nella dashboard$_G$ dell'umidità dovranno essere visualizzati: un grafico time series$_G$, una mappa interattiva, l'umidità media, minima e massima di un certo periodo di tempo e l'umidità in tempo reale.
RF-18	Obbligatorio	Soddisfatto	Nella dashboard$_G$ della qualità dell'aria dovranno essere visualizzati: un grafico time series$_G$, una mappa interattiva, la qualità media dell'aria in un certo periodo e in tempo reale, i giorni con la qualità dell'aria migliore e peggiore in un certo periodo di tempo.
RF-19	Obbligatorio	Soddisfatto	Nella dashboard$_G$ delle precipitazioni dovranno essere visualizzati: un grafico time series$_G$, una mappa interattiva, la quantità media di precipitazioni in un certo periodo e in tempo reale, i giorni con la quantità di precipitazioni maggiore e minore in un certo periodo di tempo.

Codice	Importanza	Stato	Descrizione
RF-20	Obbligatorio	Soddisfatto	Nella dashboard$_G$ del traffico dovranno essere visualizzati: un grafico time series$_G$, il numero di veicoli e la velocità media in tempo reale, il calcolo dell'ora di punta sulla base del numero di veicoli e velocità media.
RF-21	Obbligatorio	Soddisfatto	Nella dashboard$_G$ delle colonnine di ricarica dovranno essere visualizzati: una mappa interattiva contenente anche lo stato e il numero di colonnine di ricarica suddivise per stato in tempo reale.
RF-22	Obbligatorio	Soddisfatto	Nella dashboard$_G$ dei parcheggi dovranno essere visualizzati: una mappa interattiva con il rispettivo stato di occupazione e il conteggio di parcheggi suddivisi per stato di occupazione in tempo reale.
RF-23	Obbligatorio	Soddisfatto	Nella dashboard$_G$ delle isole ecologiche dovranno essere visualizzati: una mappa interattiva con il rispettivo stato di riempimento e il conteggio di isole ecologiche suddivise per stato di riempimento in tempo reale.
RF-24	Obbligatorio	Soddisfatto	Nella dashboard$_G$ del livello di acqua dovranno essere visualizzati: un grafico time series$_G$, una mappa interattiva, il livello medio di acqua in un certo periodo e in tempo reale.

Codice	Importanza	Stato	Descrizione
RF-25	Obbligatorio	Soddisfatto	Nel caso in cui non ci siano dati visualizzabili, il sistema deve notificare l'utente mostrando un opportuno messaggio.
RF-26	Obbligatorio	Soddisfatto	I sensori di qualità dell'aria inviano i seguenti dati: *PM10*, *PM2.5*, *NO2*, *CO*, *O3*, *SO2* in $\mu g/m^3$.
RF-27	Obbligatorio	Soddisfatto	I sensori di precipitazioni inviano la quantità di pioggia caduta in mm.
RF-28	Obbligatorio	Soddisfatto	I sensori di traffico inviano il numero di veicoli rilevati e la velocità in km/h.
RF-29	Obbligatorio	Soddisfatto	Le colonnine di ricarica inviano lo stato di occupazione e il tempo mancante alla fine della ricarica (se occupate) o il tempo passato dalla fine dell'ultima ricarica (se libere).
RF-30	Obbligatorio	Soddisfatto	I sensori di parcheggio inviano lo stato di occupazione del parcheggio (1 se occupato, 0 se libero) e il timestamp dell'ultimo cambiamento di stato.
RF-31	Obbligatorio	Soddisfatto	Le isole ecologiche inviano lo stato di riempimento come percentuale.
RF-32	Obbligatorio	Soddisfatto	I sensori di livello di acqua inviano il livello di acqua in cm.
RF-33	Obbligatorio	Soddisfatto	Il sistema deve permettere di filtrare i dati visualizzati in base a un intervallo di tempo.
RF-34	Obbligatorio	Soddisfatto	Il sistema deve permettere di filtrare i dati visualizzati in base al sensore_G che li ha generati.

Codice	Importanza	Stato	Descrizione
RF-37	Obbligatorio	Soddisfatto	Deve essere implementato almeno un simulatore di dati.
RF-39	Obbligatorio	Soddisfatto	I simulatori devono produrre dei dati verosimili.
RF-40	Obbligatorio	Soddisfatto	Per ciascuna tipologia di sensore$_G$ dev'essere sviluppata almeno una dashboard$_G$.
RF-50	Obbligatorio	Soddisfatto	Il sistema deve permettere di filtrare i dati visualizzati in base al tipo di sensore che li ha prodotti.

Tabella 3: Requisiti funzionali

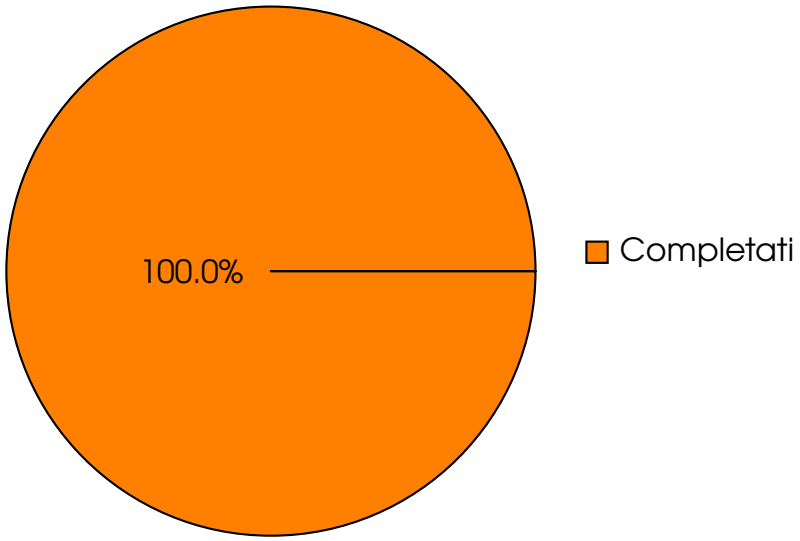

Figura 4: Percentuale di soddisfacimento dei requisiti funzionali

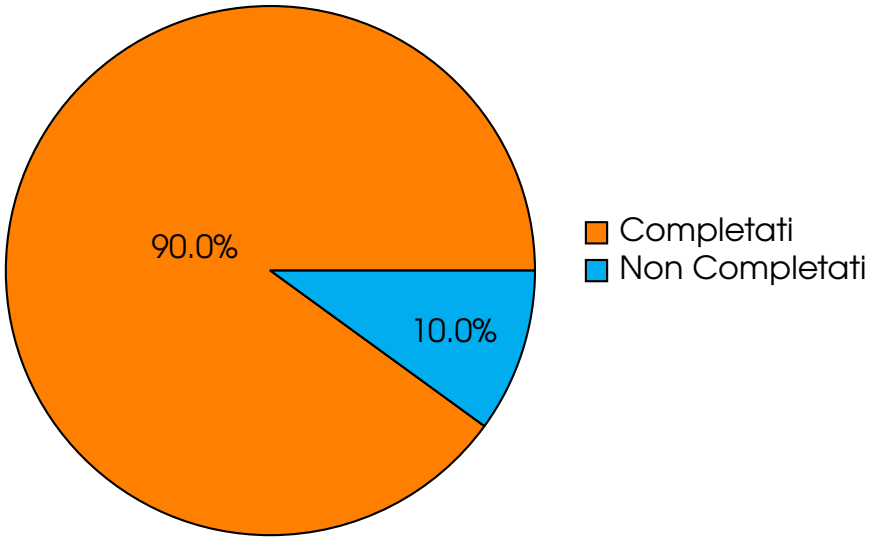

Figura 5: Percentuale di soddisfacimento dei requisiti totale