ByteOps.swe@gmail.com

Specifica tecnica

Informazioni documento

Redattori	A. Barutta R. Smanio L. Skenderi F. Pozza D. Diotto N. Preto
Verificatori	E. Hysa A. Barutta D. Diotto L. Skenderi R. Smanio F. Pozza
Destinatari	T. Vardanega R. Cardin

Versione	Data	Autore	Verificatore	Dettaglio
1.0.0	22/03/2024	N.Preto	A. Barutta	Grafici requisiti e revisione finale
0.7.0	22/03/2024	N.Preto	A. Barutta	Progettazione dashboard Grafana
0.6.0	20/03/2024	N.Preto	A. Barutta	Stesura Grafana plugin
0.5.2	19/03/2024	N.Preto	A. Barutta	Correzione sezione Database
0.5.1	18/03/2024	N.Preto	A. Barutta	Architettura di deploy completa
0.5.0	17/03/2024	N.Preto	A. Barutta	Aggiunta schema Registry
0.4.0	16/03/2024	F. Pozza	L. Skenderi	Conclusione sezione Tabella dei requisiti soddisfatti
0.3.1	15/03/2024	E. Hysa	A. Barutta	Iniziale stesura sezione Tabella dei requisiti soddisfatti
0.3.0	12/03/2024	N.Preto	D. Diotto	Conclusione sezione Architettura di deployment
0.2.1	08/03/2024	R. Smanio	A. Barutta	Iniziale stesura sezione Architettura di deployment
0.2.0	07/03/2024	E. Hysa	A. Barutta	Completamento scrittura sottosezione 3.7, correzioni ulteriori sezione Architettura di sistema
0.1.5	05/03/2024	N. Preto	R. Smanio	Completamento scrittura sottosezione 3.5 e 3.6
0.1.4	04/03/2024	N. Preto	R. Smanio	Completamento scrittura sottosezione 3.4
0.1.3	01/03/2024	F. Pozza	A. Barutta	Completamento scrittura sottosezione 3.3

0.1.2	24/02/2024	F. Pozza	E. Hysa	Completamento scrittura sottosezioni 3.1 e 3.2
0.1.1	22/02/2024	D. Diotto	R. Smanio	Iniziale stesura sezione Architettura di sistema
0.1.0	16/02/2024	E. Hysa	R. Smanio	Conclusione sezione Tecnologie
0.0.2	08/02/2024	A. Barutta	F. Pozza	Continuazione stesura Tecnologie
0.0.2	05/02/2024	A. Barutta	F. Pozza	Stesura sezione Introduzione, inizio stesura sezione Tecnologie
0.0.1	01/02/2024	E. Hysa	A. Barutta	Impostazione sezioni

Indice

ByteOps

Contents

1 Introduzione **8**

1.1 Scopo del documento . . . 8

1.2 Scopo del progetto . . . 8

1.3 Glossario . . . 9

1.4 Riferimenti . . . 9

1.4.1 Riferimenti normativi . . . 9

1.4.2 Riferimenti informativi . . . 9

2 Tecnologie **12**

2.1 Docker . . . 12

2.1.1 Ambienti . . . 12

2.1.2 Docker images . . . 13

2.2 Linguaggi e formato dati . . . 15

2.2.1 Python . . . 15

2.2.2 SQL (Structured Query Language) . . . 17

2.2.3 JSON (JavaScript Object Notation) . . . 18

2.2.4 YAML (YAML Ain't Markup Language) . . . 18

2.3 Database e servizi . . . 18

2.3.1 Apache Kafka . . . 18

2.3.2 Schema Registry . . . 20

2.3.3 Zookeper . . . 21

2.3.4 Clickhouse . . . 22
2.3.5 Grafana . . . 24

3 Architettura di sistema 25
3.1 Modello architetturale . . . 25
3.1.1 κ-architecture . . . 25
3.1.2 Componenti di sistema . . . 26
3.2 Data-flow . . . 27
3.3 Architettura dei simulatori . . . 28
3.3.1 Modulo simulatori sensori . . . 29
3.3.2 Modulo Writers . . . 39
3.3.3 Modulo Threading/Scheduling . . . 44
3.3.4 Progettazione - Panoramica UML . . . 50
3.4 Apache Kafka . . . 51
3.4.1 Kafka topic . . . 51
3.4.2 Formato messaggi . . . 51
3.5 Schema Registry . . . 52
3.5.1 Schema dei messaggi . . . 54
3.6 Faust - Processing Layer . . . 56
3.6.1 Introduzione . . . 56
3.6.2 Componenti Faust & Processing Layer . . . 57
3.6.3 Processing layer data-flow . . . 58
3.6.4 Modello per il calcolo del punteggio di salute . . . 60
3.6.5 Modulo Writer . . . 67
3.6.6 Modulo Threading/Scheduling . . . 68
3.6.7 Modulo Processing . . . 70
3.7 Configurazione Database . . . 72
3.7.1 Funzionalità Clickhouse utilizzate . . . 73
3.7.2 Integrazione Kafka tramite Kafka Engine in ClickHouse . . . 79
3.7.3 Trasferimento dati tramite Materialized View . . . 80
3.7.4 Tabella Kafka Engine per un sensore generico . . . 81

3.7.5 Misurazioni temperatura . . . 83
3.7.6 Misurazioni umidità . . . 85
3.7.7 Misurazioni di polveri sottili . . . 86
3.7.8 Misurazioni isole ecologiche . . . 88
3.7.9 Misurazioni guasti elettrici . . . 89
3.7.10 Misurazioni stazioni di ricarica . . . 90
3.7.11 Misurazioni sensori di rilevameno dell'acqua . . . 90
3.7.12 Punteggi di salute . . . 91
3.8 Grafana . . . 92
3.8.1 Utenti . . . 92
3.8.2 Dashboards . . . 93
3.8.3 ClickHouse data source plugin . . . 96
3.8.4 Variabili Grafana . . . 97
3.8.5 Grafana alerts . . . 98
3.8.6 Altri plugin utilizzati . . . 100

4 Architettura di deployment **101**

5 Stato dei requisiti funzionali **102**
5.1 Grafici riassuntivi requisiti soddisfatti . . . 115

List of Figures

1 Componenti dell'architettura - InnovaCity . . . 26
2 Data-flow - InnovaCity . . . 27
3 Modulo simulatori sensori - InnovaCity . . . 29
4 Modulo writers - InnovaCity . . . 39
5 Modulo Threading/Scheduling simulatori sensori - InnovaCity . . . 44
6 Panoramica progettazione simulatori sensori UML - InnovaCity . . . 50
7 Schema Registry overview - Confluent documentation . . . 53
8 Faust data-flow . . . 59
9 Modello per il calcolo del punteggio di salute - InnovaCity . . . 60

10 Modulo Writer - InnovaCity . . . 67
11 Modulo Threading/Scheduling health Model - InnovaCity . . . 68
12 Modulo Processing - InnovaCity . . . 70
13 Clickhouse data pipeline . . . 74
14 Query esempio senza projection - ClickHouse . . . 78
15 Query esempio projection - ClickHouse . . . 79
16 Architettura di Kafka Engine in ClickHouse . . . 80
17 Tabella sensore generico per il reperimento da Kafka - ClickHouse . . . 81
18 Tabella temperatures_kafka e temperatures . . . 84
19 Tabella humidity_kafka e humidity . . . 86
20 Tabella dust_PM10_kafka e dust_PM10 . . . 87
21 Tabella ecoIslands_kafka e ecoIslands . . . 88
22 Tabella electricalFault_kafka e electricalFault . . . 89
23 Tabella chargingStation_kafka e chargingStation . . . 90
24 Tabella waterPresence_kafka e waterPresence . . . 91
25 Tabella healthScore_kafka e healthScore . . . 91
26 Stato dei requisiti funzionali obbligatori . . . 115
27 Stato dei requisiti funzionali totali . . . 116

1 Introduzione

1.1 Scopo del documento

Il presente documento si propone come una risorsa esaustiva per la comprensione degli aspetti tecnici chiave del progetto "InnovaCity". La sua finalità principale è fornire una descrizione dettagliata e approfondita di due aspetti centrali: l'*architettura*$_G$ implementativa e l'*architettura*$_G$ di deployment.
Nel contesto dell'*architettura*$_G$ implementativa, è prevista un'analisi approfondita che si estenda anche al livello di design più dettagliato. Ciò include la definizione e la spiegazione dettagliata dei design *pattern*$_G$ e degli idiomi utilizzati nel contesto del progetto.
Gli obbiettivi del presente documento sono: motivare le scelte di sviluppo adottate, fungere da guida fondamentale per l'*attività*$_G$ di codifica e manutenzione ed infine garantire una completa copertura dei requisiti identificati nel documento *Analisi dei Requisiti v2.0.0 - Sez. Requisiti*.

1.2 Scopo del progetto

Sviluppare una *piattaforma*$_G$ di monitoraggio di una "*Smart City*$_G$" che consenta di avere sotto controllo lo stato di salute della città in modo tale da prendere decisioni veloci, efficaci ed analizzare poi gli effetti conseguenti.
A tale scopo il *proponente*$_G$ richiede di simulare dei sensori posti in diverse aree per reperire informazioni relative alle condizioni della città come, ad esempio, temperatura, umidità, quantità di polveri sottili nell'aria, livelli dell'acqua in punti strategici, stato di riempimento delle isole ecologiche, guasti elettrici e stato di occupazione delle colonnine di ricarica per macchine elettriche.
I dati trasmessi in tempo reale dai sensori devono poter essere memorizzati in un *database*$_G$ in modo tale da renderli disponibili per la visualizzazione tramite una *dashboard*$_G$, composta da *widget*$_G$, quali mappe, tabelle e grafici, per una visione d'insieme delle condizioni della città in tempo reale.
L'applicativo potrà consentire alle autorità locali di prendere decisioni informate e tempestive sulla gestione delle risorse e sull'implementazione di servizi.
L'implementazione di una città monitorata da sensori rappresenta un approccio promettente nell'ottica di ottimizzare l'efficienza e la qualità della vita urbana. Tale *sistema*$_G$ consente una raccolta continua di dati e informazioni cruciali, fornendo una base solida per l'ottimizzazione dei servizi pubblici, la gestione del traffico, la sicurezza e la sostenibilità ambientale.

1.3 Glossario

Per evitare possibili ambiguità che potrebbero sorgere durante la lettura dei documenti, alcuni termini utilizzati sono stati inseriti nel documento *Glossario v 2.0.0* .
Sarà possibile individuare il riferimento al Glossario per mezzo di una G a pedice del termine considerato ambiguo.

1.4 Riferimenti

1.4.1 Riferimenti normativi

- *Norme di progetto v2.0.0*;
- Capitolato d'appalto C6 - InnovaCity:
  - `https://www.math.unipd.it/tullio/IS-1/2023/Progetto/C6p.pdf` (Consultato: 19/03/2024);
- Regolamento di progetto:
  - `https://www.math.unipd.it/~tullio/IS-1/2023/Dispense/PD2.pdf` (Consultato: 19/03/2024).

1.4.2 Riferimenti informativi

- "Clean Code" di Robert C. Martin;
- *Analisi dei Requisiti v2.0.0*;
- *Glossario v2.0.0*;
- Documentazione Confluent *Kafka*$_G$ *Python*$_G$:
  - `https://developer.confluent.io/get-started/python/` (Consultato: 19/03/2024)
- Documentazione Faust:
  - `https://faust.readthedocs.io/en/latest/` (Consultato: 19/03/2024)
- Documentazione Pytest:
  - `https://docs.pytest.org/en/7.1.x/contents.html` (Consultato: 19/03/2024)
- Documentazione Pylint:

  - https://pylint.readthedocs.io/en/stable/ (Consultato: 19/03/2024)
- Documentazione *Clickhouse*$_G$-connect:
  - https://clickhouse.com/docs/en/integrations/python (Consultato: 19/03/2024)
- Documentazione *Kafka*$_G$:
  - https://kafka.apache.org/20/documentation.html (Consultato: 19/03/2024)
- Documentazione Zookeeper:
  - https://zookeeper.apache.org/documentation.html (Consultato: 19/03/2024)
- Documentazione *Clickhouse*$_G$:
  - https://clickhouse.com/docs/en/intro (Consultato: 19/03/2024)
- Documentazione Schema Registry:
  - https://docs.confluent.io/platform/current/schema-registry/index.html (Consultato: 19/03/2024)
- Documentazione *Grafana*$_G$:
  - https://grafana.com/docs/grafana/latest/ (Consultato: 19/03/2024)
- Documentazione Materialized View - *Clickhouse*$_G$:
  - https://clickhouse.com/docs/en/guides/developer/cascading-materialized-views (Consultato: 19/03/2024)
- Documentazione Merge Tree - *Clickhouse*$_G$:
  - https://clickhouse.com/docs/en/engines/table-engines/mergetree-family/mergetree#mergetree (Consultato: 19/03/2024)
- Documentazione TTL - *Clickhouse*$_G$:
  - https://clickhouse.com/docs/en/guides/developer/ttl#implementing-a-rollup (Consultato: 19/03/2024)
- Documentazione Partition - *Clickhouse*$_G$:

  - https://clickhouse.com/docs/en/engines/table-engines/mergetree-family/mergetree#partition-by (Consultato: 19/03/2024)

- Documentazione Projection - *Clickhouse*$_G$:
  - https://clickhouse.com/docs/en/sql-reference/statements/alter/projection (Consultato: 19/03/2024)

- Altra fonte utile Projection - *Clickhouse*$_G$
  - https://presentations.clickhouse.com/percona2021/projections.pdf (Consultato: 19/03/2024)

- Variable Panel plugin - *Grafana*$_G$
  - https://volkovlabs.io/plugins/volkovlabs-variable-panel/ (Consultato: 19/03/2024)

- Alerts - *Grafana*$_G$
  - https://grafana.com/docs/grafana/latest/alerting/ (Consultato: 19/03/2024)

- Variables - *Grafana*$_G$
  - https://grafana.com/docs/grafana/latest/dashboards/variables/ (Consultato: 19/03/2024)

- Notification policies - *Grafana*$_G$
  - https://grafana.com/docs/grafana/latest/alerting/alerting-rules/create-notification-policy/ (Consultato: 19/03/2024)

- Orchestra Cities Map plugin - *Grafana*$_G$
  - https://grafana.com/grafana/plugins/orchestracities-map-panel/ (Consultato: 19/03/2024)

- Documentazione *Docker*$_G$:
  - https://docs.docker.com/ (Consultato: 19/03/2024)

- Documentazione *Docker*$_G$ Compose:
  - https://docs.docker.com/compose/ (Consultato: 19/03/2024)

- Kappa Architecture: Stream Processing in Big Data Analytics:

- https://medium.com/data-and-beyond/kappa-architecture-stream-processing-in-big-data-analytics-e539f4bf63cd (Consultato: 19/03/2024)

2 Tecnologie

In questa sezione sono definiti gli strumenti e le tecnologie impiegati per lo sviluppo e l'implementazione del *software*$_G$ relativo al progetto InnovaCity.
Si procederà quindi con la descrizione delle tecnologie e dei linguaggi di programmazione utilizzati, delle *librerie*$_G$ e dei *framework*$_G$ necessari, nonché delle infrastrutture richieste.
L'obiettivo principale è garantire che il *software*$_G$ sia sviluppato utilizzando le tecnologie più appropriate in termini di efficienza, sicurezza e affidabilità.

2.1 Docker

Per lo sviluppo, il testing e il rilascio del prodotto sono stati utilizzati *container*$_G$ *Docker*$_G$ in modo tale da garantire ambienti consistenti e riproducibili.

2.1.1 Ambienti

- **Ambiente di sviluppo:**
  - È l'ambiente dove i *software*$_G$ developer scrivono, testano e modificano il codice sorgente;
  - Può includere strumenti di debug e monitoraggio per facilitare lo sviluppo e la correzione di errori;
  - Non è accessibile agli utenti finali.
- **Ambiente di test:**
  - Simula l'ambiente di produzione;
  - Viene utilizzato per testare il *software*$_G$ in modo completo e realistico prima del rilascio in produzione;
  - I *test*$_G$ vengono eseguiti automaticamente tramite *Github*$_G$ *workflow*$_G$ oppure manualmente in locale tramite profilo di testing *docker*$_G$ e includono *test di unità*$_G$, *integrazione*$_G$, *sistema*$_G$ (sicurezza, carico e prestazioni).
- **Ambiente di produzione:**

- È l'ambiente dove il *software*$_G$ viene rilasciato per poter essere utilizzato dagli utenti finali;
- Deve essere stabile, sicuro e performante per garantire un'esperienza utente ottimale;
- Le modifiche al *software*$_G$ in produzione sono controllate rigorosamente per minimizzare i rischi di errori o downtime.

2.1.2 Docker images

Di seguito sono elencate le immagini *Docker*$_G$ utilizzate:

- **Simulators - Python**
  - **Image:** *Python*$_G$: 3.9;
  - **Riferimento:** `https://hub.docker.com/layers/library/python/3.9/images/sha256-1023bd4c5e0e6b7f4f612b034627826d91ec78ae0439313450ec30c0ad60c908?context=explore` (Consultato: 19/03/2024);
  - **Ambiente:**
    - Develop;
    - Production.
- **Broker - Apache Kafka**
  - **Image:** confluentinc/cp-*kafka*$_G$: 7.6.0;
  - **Riferimento:** `https://hub.docker.com/layers/confluentinc/cp-kafka/7.6.0/images/sha256-8fc15a671986983b83beecae14e013a91adcd3999f687de8b6b8153fd47e8f67?context=explore` (Consultato: 19/03/2024);
  - **Ambiente:**
    - Develop;
    - Production;
    - Testing;
- **Zookeeper**
  - **Image:** confluentinc/cp-zookeeper: 7.6.0;
  - **Riferimento:** `https://hub.docker.com/layers/confluentinc/cp-zookeeper/7.6.0/images/sha256-6a0822643ceb4725db4f24bf2d228eee39bb5ade88f586449d87263cbc81bc97?context=explore` (Consultato: 19/03/2024);

  - **Ambiente:**
    - Develop;
    - Production;
    - Testing;

- **Apache Kafka UI**
  - **Image:** provectuslabs/*kafka*$_G$-ui: 53a6553765a806eda9905c43bfcfe09da6812035;
  - **Riferimento:** https://hub.docker.com/layers/provectuslabs/kafka-ui/53a6553765a806eda9905c43bfcfe09da6812035/images/sha256-633606ca07677d1c4b9405c5df1b6f0087aa75b36528a17eed142d06f65d0881?context=explore (Consultato: 19/03/2024);
  - **Ambiente:**
    - Develop.

- **Schema registry**
  - **Image:** confluentinc/cp-schema-registry: 7.6.0;
  - **Riferimento:** https://hub.docker.com/layers/confluentinc/cp-schema-registry/7.6.0/images/sha256-7cea5369377b52823d3101dd22073a235a501256f6f140c66d2111224803af0b?context=explore (Consultato: 19/03/2024);
  - **Ambiente:**
    - Develop;
    - Production;
    - Testing;

- **Schema registry UI**
  - **Image:** landoop/schema-registry-ui: latest;
  - **Riferimento:** https://hub.docker.com/layers/landoop/schema-registry-ui/latest/images/sha256-c8b7baf7c53224eaa066937410adae388384e3f7c6f26296ba6a98cc5880f866?context=explore (Consultato: 19/03/2024);
  - **Ambiente:**
    - Develop.

- **Faust processing - Python**

  - **Image:** *Python*$_G$: 3.9;
  - **Riferimento:** https://hub.docker.com/layers/library/python/3.9/images/sha256-1023bd4c5e0e6b7f4f612b034627826d91ec78ae0439313450ec30c0ad60c908?context=explore (Consultato: 19/03/2024);
  - **Ambiente:**
    * Develop;
    * Production;
    * Testing;

- **ClickHouse**
  - **Image:** *clickhouse*$_G$/*clickhouse*$_G$-server: 24.2.1.2248;
  - **Riferimento:** https://hub.docker.com/layers/clickhouse/clickhouse-server/24.2.1.2248/images/sha256-a5921f08bc3ab230e20db5970698b300279b29a353620e62729325fa8d1dc601?context=explore (Consultato: 19/03/2024);
  - **Ambiente:**
    * Develop;
    * Production;
    * Testing;

- **Grafana**
  - **Image:** grafana/grafana-oss: 10.4.0;
  - **Riferimento:** https://hub.docker.com/layers/grafana/grafana-oss/10.4.0/images/sha256-c7ae30e06ee76656f4faf37df1f0d0dfb6941a706b66800a7b289a304d31d771?context=explore (Consultato: 19/03/2024);
  - **Ambiente:**
    * Develop;
    * Production.

2.2 Linguaggi e formato dati

2.2.1 Python

Linguaggio di programmazione ad alto livello, interpretato e multi-*paradigma*$_G$.

2.2.1.1 Versione:

Versione utilizzata: 3.9

2.2.1.2 Documentazione:

https://docs.python.org/release/3.9.0/ (Consultato: 19/03/2024).

2.2.1.3 Utilizzo nel progetto

- Creazione delle simulazioni dei sensori, incluse le logiche di scrittura e invio dei dati registrati;
- Modello per il calcolo del punteggio di salute della città;
- Testing.

2.2.1.4 Librerie o framework

- **Confluent Kafka**
  - **Documentazione:** https://developer.confluent.io/get-started/python/ (Consultato: 19/03/2024);
  - **Versione:** 2.3.0;
  - Libreria *Python*$_G$ che fornisce un insieme completo di strumenti per agevolare la produzione e il consumo di messaggi da *Apache Kafka*$_G$.
- **Faust**
  - **Documentazione:** https://faust.readthedocs.io/en/latest/ (Consultato: 19/03/2024);
  - **Versione:** 1.10.4;
  - *Framework*$_G$ *Python*$_G$ per la creazione di applicazioni di data streaming in tempo reale. Fornisce un'*API*$_G$ dichiarativa e funzionale per definire i flussi di dati e le trasformazioni, consentendo agli sviluppatori di scrivere facilmente applicazioni scalabili e affidabili per il trattamento di grandi volumi di dati in tempo reale.

    Faust si integra nativamente con *Apache Kafka*$_G$ e offre funzionalità avanzate come il bilanciamento del carico, la gestione dello stato, la gestione delle *query*$_G$, e la tolleranza ai guasti, rendendolo una scelta ottimale per lo sviluppo di sistemi di data streaming complessi e robusti.
- **Pytest**
  - **Documentazione:** https://docs.pytest.org/en/7.1.x/contents.html (Consultato: 19/03/2024);

- **Versione:** 8.0.2;
- *Framework*$_G$ di testing per *Python*$_G$, noto per la sua semplicità. Consente agli sviluppatori di scrivere *test*$_G$ chiari e concisi utilizzando una sintassi intuitiva e flessibile.

  Pytest supporta una vasta gamma di funzionalità, tra cui *test di unità*$_G$, *integrazione*$_G$ e accettazione, parametrizzazione dei *test*$_G$ e gestione delle *fixture*$_G$.

  Merita menzione anche l'utilizzo di *Pytest-asyncio* per testare codice asincrono e *Pytest-cov* per la copertura del codice.

- **Pylint**
  - **Documentazione:** https://pylint.readthedocs.io/en/stable/ (Consultato: 19/03/2024);
  - **Versione:** 3.1.0;
  - Strumento di analisi statica per il linguaggio di programmazione *Python*$_G$. Esamina il codice sorgente per individuare potenziali errori, conformità alle linee guida stilistiche e altre possibili fonti di bug nel codice *Python*$_G$. Inoltre, valuta anche la qualità del codice in termini di *good practice* di programmazione.

    Pylint fornisce un punteggio di qualità del codice e suggerimenti per migliorare la leggibilità, la manutenibilità, sicurezza e la correttezza del codice *Python*$_G$.

- **Clickhouse-connect**
  - **Documentazione:** https://clickhouse.com/docs/en/integrations/python (Consultato: 19/03/2024);
  - **Versione:** 0.7.2;
  - *ClickHouse*$_G$ Connect è una libreria open source sviluppata per semplificare l'interazione con il *database*$_G$ *ClickHouse*$_G$ tramite il linguaggio di programmazione *Python*$_G$, viene utilizzata nei *test*$_G$.

    Essa fornisce un'interfaccia per comunicare con *ClickHouse*$_G$, consentendo agli sviluppatori di eseguire *query*$_G$, inserire dati e gestire altri aspetti dell'interazione con il *database*$_G$ in modo efficiente e conveniente.

2.2.2 SQL (Structured Query Language)

Linguaggio *standard*$_G$ per la gestione e la manipolazione dei *database*$_G$.

2.2.2.1 Utilizzo nel progetto

Gestione e interrogazione *database*$_G$ *Clickhouse*$_G$.

2.2.3 JSON (JavaScript Object Notation)

JSON è un formato di scrittura leggibile dalle persone grazie alla sua sintassi semplice e chiara, inoltre è facilmente ed efficientemente interpretabile dai computer. È ampiamente impiegato in diversi contesti, tra cui lo sviluppo web, le *API*$_G$ di servizi web e lo scambio di dati tra applicazioni.
Il formato JSON si basa su due strutture di dati principali:

- **Oggetti**: Rappresentati da coppie chiave-valore racchiuse tra parentesi graffe { }, dove la chiave è una stringa e il valore può essere un altro oggetto, un array, una stringa, un numero, un booleano o `null`;
- **Array**: Una raccolta ordinata di valori, racchiusi tra parentesi quadre [ ], in cui ogni elemento può essere un oggetto, un array, una stringa, un numero, un booleano o `null`.

2.2.3.1 Utilizzo nel progetto

- Formato dei messaggi trasmessi dai simulatori dei sensori al *broker*$_G$ *Kafka*$_G$;
- Configurazione *dashboard*$_G$ *Grafana*$_G$.

2.2.4 YAML (YAML Ain't Markup Language)

Formato di serializzazione leggibile dall'uomo utilizzato per rappresentare dati strutturati in modo chiaro e semplice.

2.2.4.1 Utilizzo nel progetto

- Configurazione *Docker*$_G$ Compose;
- Configurazione pipeline *Github*$_G$ *workflow*$_G$ per Countinuous Integration;
- Configurazione provisioning *Grafana*$_G$ e politiche di notifica allerte.

2.3 Database e servizi

2.3.1 Apache Kafka

Apache Kafka$_G$ è una *piattaforma*$_G$ *open-source*$_G$ di streaming distribuito sviluppata dall'Apache *Software*$_G$ Foundation. Progettata per gestire flussi di dati in tempo reale in modo scalabile e affidabile, è ampiamente utilizzata nel data streaming e nell'*integrazione*$_G$ dei dati nelle moderne applicazioni.

2.3.1.1 Versione:

Versione utilizzata: 3.7.0

2.3.1.2 Documentazione:

https://kafka.apache.org/20/documentation.html (Consultato: 19/03/2024).

2.3.1.3 Funzionalità e vantaggi di Apache Kafka

Le principali funzionalità e vantaggi di *Apache Kafka*$_G$ includono:

- **Pub-Sub messaging:** *Kafka*$_G$ utilizza un modello di messaggistica publish-subscribe, dove i produttori di dati inviano messaggi ad uno o più topic e i consumatori possono sottoscriversi a tali topic per ricevere i messaggi;
- **Disaccoppiamento produttore - consumatore:** questo principio si realizza grazie al fatto che i produttori e i consumatori non necessitano di essere consapevoli l'uno dell'altro o di interagire direttamente. Invece, essi comunicano attraverso il *broker*$_G$ *Kafka*$_G$, che svolge il ruolo di intermediario per la trasmissione dei messaggi. Ciò consente una maggiore scalabilità e flessibilità nell'*architettura*$_G$ del *sistema*$_G$, facilitando la gestione e il mantenimento delle applicazioni;
- **Architettura distribuita:** *Kafka*$_G$ è progettato per essere distribuito su un cluster di nodi, consentendo una scalabilità orizzontale per gestire grandi volumi di dati e carichi di lavoro. Questo approccio distribuito offre resilienza e alta disponibilità, garantendo che il *sistema*$_G$ possa crescere in modo flessibile con l'aumentare delle richieste;
- **Persistenza e affidabilità:** *Kafka*$_G$ offre la possibilità di definire politiche specifiche per la conservazione dei dati, garantendo la durabilità dei messaggi. Questo non solo assicura la disponibilità dei dati anche in caso di eventuali interruzioni del *servizio*$_G$, ma consente anche ai consumatori di recuperare i messaggi dopo tali anomalie, garantendo un alto livello di affidabilità nel *sistema*$_G$;
- **Alta disponibilità:** *Kafka*$_G$ assicura un'elevata disponibilità e tolleranza ai guasti grazie alla sua *architettura*$_G$ distribuita e al meccanismo di replica dei dati. Anche in caso di malfunzionamenti dei nodi o delle componenti, i cluster di *Kafka*$_G$ mantengono la loro operatività, garantendo la continuità del *servizio*$_G$.

2.3.1.4 Casi d'uso di Apache Kafka

Apache Kafka$_G$ è utilizzato in una vasta gamma di casi d'uso, tra cui:

- **Data integration:** *Kafka*$_G$ viene utilizzato per integrare dati provenienti da diverse fonti e sistemi, consentendo lo scambio di dati in tempo reale tra applicazioni e sistemi eterogenei;
- **Streaming di eventi:** Molte applicazioni moderne, come le applicazioni IoT (Internet of Things) e le applicazioni di monitoraggio in tempo reale, utilizzano *Kafka*$_G$ per lo streaming di eventi in tempo reale e l'analisi dei dati;

- **Analisi dei log:** *Kafka*$_G$ è spesso utilizzato per l'analisi dei *log*$_G$ di *sistema*$_G$ di applicativi in tempo reale, consentendo il monitoraggio delle prestazioni, la rilevazione degli errori e l'analisi dei *pattern*$_G$ di utilizzo;
- **Elaborazione di big data:** *Kafka*$_G$ è integrato con tecnologie di *big data*$_G$ come Apache Hadoop e Apache Spark, consentendo l'elaborazione di grandi volumi di dati in tempo reale;
- **Messaggistica Real-time:** *Kafka*$_G$ è ampiamente utilizzato per la messaggistica real-time in applicazioni di social media, e-commerce e finanziarie, dove la velocità e l'affidabilità della messaggistica sono cruciali.

2.3.1.5 Utilizzo nel progetto

All'interno del nostro prodotto *Kafka*$_G$ funge da *broker*$_G$, ricevendo i dati dai produttori e rendendoli disponibili ai consumatori. Nel contesto del progetto, i dati provenienti dalle simulazioni di sensori vengono inviati a *Kafka*$_G$ come messaggi in formato JSON.

Consumatori di dati:

- **ClickHouse:** *Kafka*$_G$ rende disponibili i dati al *database*$_G$ *ClickHouse*$_G$, dove i dati vengono salvati per l'analisi e l'archiviazione a lungo termine;
- **Faust:** per soddisfare il requisito opzionale del calcolo del punteggio di salute, *Kafka*$_G$ rende disponibili i dati in tempo reale a un'applicazione Faust. Quest'ultima elabora i dati utilizzando una funzione di aggregazione per calcolare il punteggio e quindi mette a disposizione il risultato in una coda dedicata di *Kafka*$_G$ per i servizi interessati.

In breve, *Kafka*$_G$ funge da ponte tra i produttori di dati (simulazioni di sensori) e i consumatori di dati (*ClickHouse*$_G$, Faust o altri servizi futuri). Gestisce il flusso dei dati in tempo reale e garantisce che i dati siano disponibili per l'elaborazione e la visualizzazione in modo efficiente e scalabile.

2.3.2 Schema Registry

Schema Registry è un componente importante nell'ecosistema di *Apache Kafka*$_G$, progettato per la gestione e la convalida degli schemi dei dati utilizzati all'interno di un *sistema*$_G$ di messaggistica distribuita.

2.3.2.1 Versione

Versione utilizzata: 7.6.0

2.3.2.2 Documentazione

https://docs.confluent.io/platform/current/schema-registry/index.html (Consultato: 19/03/2024).

2.3.2.3 Funzionalità e Vantaggi di Schema Registry

Le funzionalità principali di Schema Registry includono:

- **Gestione centralizzata degli schemi**: Fornisce un *repository*$_G$ centralizzato per la gestione degli schemi dei dati. Contribuisce alla governance dei dati garantendo la qualità, la conformità agli *standard*$_G$ e la tracciabilità dei dati;
- **Convalida degli schemi**: Assicura la validità e la compatibilità degli schemi dei dati;
- **Serializzazione e deserializzazione**: Supporta la serializzazione e la deserializzazione dei dati basati sugli schemi su reti distribuite.

2.3.2.4 Utilizzo nel progetto

Nell'ambito del progetto didattico schema registry permette di validare i messaggi nell'ambito del topic kakfa di appartenenza definendo un contratto che i produttori, ovvero i sensori, dovranno rispettare nell'invio delle misurazioni.

2.3.3 Zookeper

Apache Zookeeper è un *servizio*$_G$ di coordinamento *open-source*$_G$ sviluppato dalla Apache *Software*$_G$ Foundation. È progettato per fornire funzionalità di coordinazione affidabili e scalabili per applicazioni distribuite.

2.3.3.1 Versione:

Versione utilizzata: 7.6.0

2.3.3.2 Documentazione:

https://zookeeper.apache.org/documentation.html (Consultato: 19/03/2024).

2.3.3.3 Funzionalità e vantaggi di Apache Zookeeper:

Le principali funzionalità e vantaggi di Apache Zookeeper includono:

- **Servizio di coordinazione centralizzato:** Zookeeper fornisce un *servizio*$_G$ centralizzato per la gestione delle configurazioni, l'elezione del leader, la sincronizzazione dei dati e la notifica di eventi;
- **Affidabilità e scalabilità:** Zookeeper è progettato per essere affidabile e scalabile, in grado di gestire grandi cluster di applicazioni distribuite;

- **Integrazione con altri software:** Zookeeper è integrato con molti altri *software*$_G$ *open-source*$_G$, tra cui *Apache Kafka*$_G$.

2.3.3.4 Utilizzo nel progetto:

Zookeeper è utilizzato principalmente:

- **Sincronizzazione dei nodi Kafka:** Memorizza la configurazione del cluster *Kafka*$_G$, inclusa la lista dei *broker*$_G$ attivi. Quando un nuovo *broker*$_G$ viene aggiunto, Zookeeper aggiorna la configurazione e notifica gli altri *broker*$_G$. Questo garantisce che tutti i *broker*$_G$ abbiano una visione coerente del cluster e possano comunicare correttamente;
- **Coordinamento dello schema registry:** Memorizza lo schema per tutti i topic *Kafka*$_G$ utilizzati nel progetto. Quando un client tenta di produrre un messaggio su un topic, lo schema registry verifica lo schema con Zookeeper. Se lo schema è compatibile, il messaggio viene accettato,in caso contrario, il messaggio viene rifiutato. Questo garantisce che solo messaggi con schemi validi vengano pubblicati sui topic.

2.3.4 Clickhouse

Clickhouse$_G$ è un *sistema*$_G$ di gestione di *database*$_G$ (DBMS) di tipo column-oriented, progettato principalmente per l'analisi di grandi volumi di dati in tempo reale. È un progetto *open-source*$_G$ creato per rispondere alle esigenze di elaborazione analitica ad alte prestazioni.

2.3.4.1 Versione

Versione utilizzata: 24.2.1.2248

2.3.4.2 Documentazione:

`https://clickhouse.com/docs/en/intro` (Consultato: 19/03/2024);

2.3.4.3 Funzionalità e Vantaggi di Clickhouse

- **Modello di dati column-oriented:** a differenza dei tradizionali DBMS che memorizzano i dati in modo row-oriented, dove le righe complete sono memorizzate in sequenza, *clickhouse*$_G$ memorizza i dati in modo column-oriented. Questo significa che i dati di una stessa colonna vengono memorizzati contiguamente, permettendo una maggiore compressione e velocità di *query*$_G$ per le analisi che coinvolgono molte colonne;
- **Architettura distribuita e scalabilità:** *Clickhouse*$_G$ è progettato per funzionare in un ambiente distribuito, consentendo la scalabilità orizzontale per gestire grandi carichi di lavoro;
- **Compressione dei dati:** utilizza algoritmi efficienti per ridurre lo spazio di archiviazione richiesto per i dati, riducendo i costi di archiviazione;

- **Alte prestazioni:** *Clickhouse*$_G$ è ottimizzato per eseguire *query*$_G$ analitiche su grandi volumi di dati in tempo reale, garantendo tempi di risposta bassi anche con carichi di lavoro elevati.
- **Supporto per SQL:** *Clickhouse*$_G$ supporta un sottoinsieme del linguaggio SQL;
- **Integrazione con Strumenti di Business Intelligence (BI):** può essere integrato con strumenti di BI popolari come *Grafana*$_G$ per la visualizzazione e l'analisi dei dati.

2.3.4.4 Casi d'uso di Clickhouse

Clickhouse$_G$ è adatto per una vasta gamma di casi d'uso, tra cui:

- **Analisi dei log:** *Clickhouse*$_G$ può essere utilizzato per analizzare i *log*$_G$ di grandi dimensioni generati da server, applicazioni web e dispositivi IoT;
- **Analisi dei dati in tempo reale:** *Clickhouse*$_G$ è ideale per l'analisi dei dati in tempo reale, consentendo agli utenti di eseguire *query*$_G$ complesse su flussi di dati in continua evoluzione;

2.3.4.5 Utilizzo nel progetto

Nel contesto del progetto, *Clickhouse*$_G$ svolge una serie di ruoli cruciali per garantire l'efficacia e l'efficienza dell'analisi e delle persistenza dei dati provenienti dai sensori IoT:

- **Integrazione con Kafka**: *Clickhouse*$_G$ viene utilizzato per recuperare in tempo reale i dati dal server *Kafka*$_G$, consentendo una continua acquisizione dei dati dai sensori IoT. Questa *integrazione*$_G$ permette di assicurare che le informazioni più recenti siano immediatamente disponibili per l'analisi.
- **Organizzazione efficiente dei dati:** *Clickhouse*$_G$ è in grado di organizzare grandi volumi di dati in modo ottimale grazie alla sua *architettura*$_G$ columnar che consente una compressione dei dati efficace e un accesso rapido alle informazioni, migliorando le prestazioni complessive del *sistema*$_G$.
- **Aggregazione rapida dei dati:** *Clickhouse*$_G$ offre potenti funzionalità per eseguire operazioni di aggregazione sui dati in modo rapido e incrementale. Ciò significa che è possibile ottenere risposte rapide alle *query*$_G$ di aggregazione anche su enormi quantità di dati, consentendo analisi in *quasi-real-time* delle misurazioni dei sensori IoT.
- **Integrazione con Grafana:** I dati elaborati e aggregati da *Clickhouse*$_G$ sono resi disponibili per il reperimento tramite *Grafana*$_G$. *Grafana*$_G$ consente di creare *dashboard*$_G$ interattive e *report*$_G$ visivi basati sui dati ricevuti offrendo agli utenti un'interfaccia intuitiva per l'analisi e la visualizzazione dei dati.

2.3.5 Grafana

Grafana$_G$ è una *piattaforma*$_G$ *open-source*$_G$ per la visualizzazione e l'analisi dei dati, utilizzata per creare *dashboard*$_G$ interattive e grafici da fonti di dati eterogenee.

2.3.5.1 Versione

Versione utilizzata: 10.4.0

2.3.5.2 Documentazione

`https://grafana.com/docs/grafana/v10.4/` (Consultato: 19/03/2024).

2.3.5.3 Funzionalità e Vantaggi di Grafana

- **Dashboard interattive:** Creazione di *dashboard*$_G$ personalizzate e interattive per visualizzare dati provenienti da diverse fonti in un'unica interfaccia;
- **Ampia varietà di visualizzazioni:** Selezione di pannelli e visualizzazioni, tra cui grafici a linea, a barre, a torta, termometri, mappe geografiche e altro ancora, per adattarsi alle esigenze specifiche di visualizzazione dei dati;
- **Query e aggregazioni:** Esecuzione di *query*$_G$ e aggregazione dei dati in modi personalizzati per ottenere insight approfonditi dai dati;
- **Notifiche e allarmi:** Impostazione di avvisi in base a criteri predefiniti, come soglie di performance, e ricezione di notifiche tramite diversi canali di comunicazione;
- **Gestione degli accessi e dei permessi:** Controllo degli accessi e dei permessi degli utenti in modo granulare, gestendo chi può visualizzare, modificare o creare *dashboard*$_G$ e pannelli;
- **Integrazione con altre applicazioni e strumenti:** Integrazione con una vasta gamma di applicazioni e strumenti, tra cui sistemi di *log*$_G$ management, strumenti di monitoraggio delle prestazioni, sistemi di allerta e altro ancora.

2.3.5.4 Casi d'uso di Grafana

- **Monitoraggio delle prestazioni:** Monitoraggio in tempo reale delle metriche di *sistema*$_G$ come CPU, memoria e *rete*$_G$ per identificare e risolvere rapidamente problemi di prestazioni;
- **Analisi dei log:** Analisi e visualizzazione dei *log*$_G$ delle applicazioni e dell'infrastruttura per individuare *pattern*$_G$ e risolvere problemi operativi;
- **DevOps e CI/CD:** Monitoraggio dei *processi*$_G$ di sviluppo, *test*$_G$ e distribuzione del *software*$_G$ per migliorare la collaborazione e l'efficienza del team;

- **Monitoraggio di dispositivi IoT:** Monitoraggio dei dispositivi IoT per raccogliere e visualizzare dati di sensori e dispositivi connessi, consentendo una gestione efficiente degli ambienti IoT.

2.3.5.5 Utilizzo nel progetto

Nel contesto del nostro progetto che coinvolge la visualizzazione e l'analisi di grandi quantità di misurazioni, *Grafana*$_G$ viene utilizzato principalmente per:

- **Visualizzazione dei dati:** *Grafana*$_G$ consente agli utenti di visualizzare tramite *dashboard*$_G$ grafici interattivi che mostrano i dati provenienti dai sensori IoT in modo chiaro e comprensibile. Questi grafici consentendo agli utenti di monitorare facilmente le prestazioni dei sensori e rilevare eventuali *pattern*$_G$ o anomalie nei dati.
- **Analisi dei dati:** *Grafana*$_G$ offre agli utenti la possibilità di analizzare i dati visualizzati in modo approfondito fornendo opzioni di filtraggio spaziale e temporale e aggregazioni temporali.
- **Monitoraggio in tempo reale:** *Grafana*$_G$ supporta il monitoraggio in tempo reale dei dati, consentendo agli utenti di visualizzare aggiornamenti istantanei sui valori dei sensori e le metriche correlate. Ciò è particolarmente utile per la rilevazione immediata di problemi o anomalie nei dati dei sensori.
- **Allerta e notifica:** *Grafana*$_G$ permette agli utenti di ricevere avvisi basati su condizioni specifiche delle misurazioni. In particolare, nel nostro caso, invia una notifica tramite discord quando un determinato *sensore*$_G$ supera una soglia prestabilita o quando si verifica un'anomalia nei dati.

3 Architettura di sistema

3.1 Modello architetturale

Il *sistema*$_G$ richiede la capacità di elaborare dati provenienti da diverse fonti in tempo reale e di fornire una visualizzazione immediata e continua di tali dati, permettendo di monitorarne gli andamenti e di rilevare eventuali anomalie. Per tale scopo, l'*architettura*$_G$ di *sistema*$_G$ adottata è la κ-*architecture*.

3.1.1 κ-architecture

L'*architettura*$_G$ Kappa è un modello di elaborazione dati in streaming che offre un'alternativa all'*architettura*$_G$ Lambda. Il suo obiettivo principale è unificare lo *stream processing* e il *batch processing* all'interno di un unico *stack tecnologico*$_G$, mantenendo il *sistema*$_G$ in real time.

3.1.1.1 Vantaggi

- Semplice da implementare e gestire, costi di manutenzione ridotti;
- Assicura coerenza tra l'analisi in tempo reale e batch.

3.1.1.2 Svantaggi

- Potenziale rallentamento dell'analisi in tempo reale, meno flessibile rispetto a Lambda.

3.1.2 Componenti di sistema

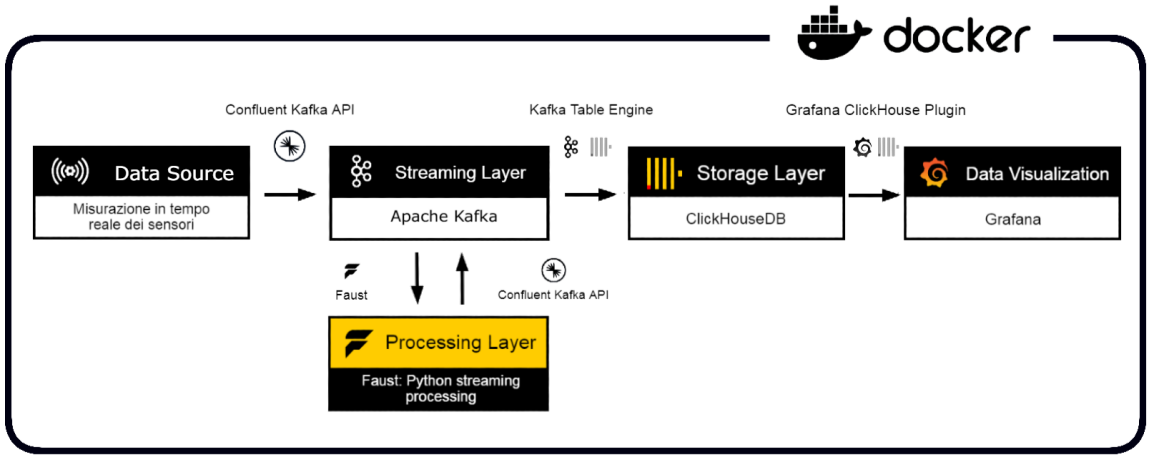

Figure 1: Componenti dell'architettura - InnovaCity

- **Data source:** Le sorgenti dati sono costituite da simulatori di sensori IoT dislocati sul territorio cittadino. Questi sensori sono in grado di inviare, ad intervalli regolari, messaggi contenenti misurazioni allo streaming layer;
- **Streaming layer:** Lo streaming layer gestisce i dati in arrivo in tempo reale, per poi archiviarli sistematicamente nello *storage*$_G$ layer;

  Lo streaming layer è composto da:

  - **Apache Kafka:** *Kafka*$_{GG}$ è un *sistema*$_G$ di messaggistica distribuito che consente di pubblicare, sottoscrivere e archiviare messaggi in tempo reale. *Kafka*$_{GG}$ è utilizzato per ricevere i dati dai sensori e renderli disponibili per l'elaborazione in tempo reale e batch;
  - **Zookeeper:** Per il coordinamento e la gestione dei cluster *Kafka*$_G$;

  - **Schema Registry:** Definizione e gestione degli schemi per i dati in streaming.

- **Processing layer:** Il processing layer è costituito da Faust che consuma i dati dallo streaming layer e li processa in tempo reale. Faust è una libreria *Python*$_G$ che permette l'elaborazione di *stream*$_G$ di dati in tempo reale. Faust è utilizzato per elaborare i dati in arrivo tramite un modello per il calcolo del punteggio di salute che poi viene reso nuovamente disponibili allo streaming layer;
- **Storage layer:** Lo *storage*$_G$ layer è costituito da un *database*$_G$ column-oriented, *ClickHouse*$_G$, che archivia i dati in arrivo dallo streaming layer. Questi dati sono disponibili per l'analisi e la visualizzazione in tempo reale e batch;
- **Data visualization layer:** Composto da *Grafana*$_G$, si occupa della visualizzazione dei dati elaborati ottenuti dallo *storage*$_G$ layer e della gestione delle notifiche in caso di anomalie rilevate.

3.2 Data-flow

Il diagramma rappresenta il percorso dei dati all'interno del *sistema*$_G$ e le relative elaborazioni. Vengono identificate le diverse entità coinvolte nel processo e le relazioni tra di esse, fornendo una panoramica dettagliata di come i dati vengono acquisiti, elaborati, archiviati e visualizzati.

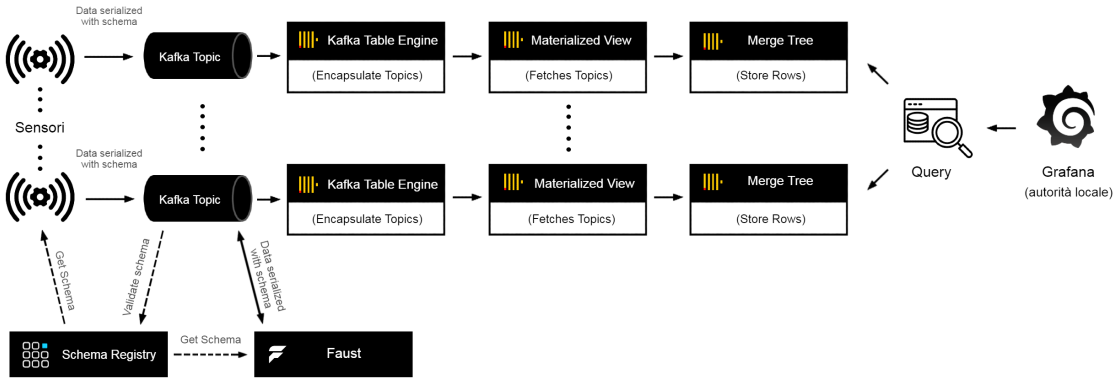

Figure 2: Data-flow - InnovaCity

1. **Generazione e invio dati:**
   - I dati vengono generati dai simulatori dei sensori IoT e inviati al *broker*$_G$ *Kafka*$_G$.

2. **Serializzazione e diramazione:**

   - I dati vengono serializzati secondo lo schema definito nello schema registry per il relativo topic. Nel caso in cui il formato del messaggio non sia conforme a quanto definito nello schema registry questo viene scartato. A questo punto, il flusso si divide in due percorsi paralleli:

     (a) **Calcolo del punteggio di salute:**
     - I dati di temperatura, umidità e polveri sottili vengono acquisiti dal processing layer per l'elaborazione;
     - Il processing layer applica il modello di calcolo del punteggio di salute e reinvia i dati processati a un topic dedicato in *Kafka*$_G$;
     - Lo schema registry verifica la validità del messaggio prima dell'invio.

     (b) **Archiviazione:**
     - Le misurazioni (comprese sia quelle generate che quelle elaborate dallo *stream processing*$_G$) fluiscono verso *ClickHouse*$_G$ tramite *Kafka*$_G$ engine e materialized view per l'archiviazione.

3. **Visualizzazione:**

   - I dati archiviati in *ClickHouse*$_G$ vengono interrogati da *Grafana*$_G$, che genera visualizzazioni per la loro presentazione.

3.3 Architettura dei simulatori

Nonostante i simulatori non siano ufficialmente considerati come parte fondamentale del prodotto dalla *proponente*$_G$, ma necessari solamente per dimostrare il corretto funzionamento del *sistema*$_G$, il nostro team ha comunque scelto di dedicare alcune risorse alla progettazione di questa componente nell'ambito del progetto didattico.
Inoltre, abbiamo deciso di implementare e tenere conto delle possibili logiche dei microcontrollori associati ai sensori IoT, che possono effettuare operazioni per rendere più efficiente l'intero *sistema*$_G$.
Nei paragrafi successivi, verrà presentata l'*architettura*$_G$ individuata mediante l'utilizzo di diagrammi delle classi e relative descrizioni. Inoltre, saranno motivate le scelte dei design *pattern*$_G$ individuati e le decisioni progettuali rilevanti. Successivamente, per ogni classe, saranno illustrati metodi e attributi.

3.3.1 Modulo simulatori sensori

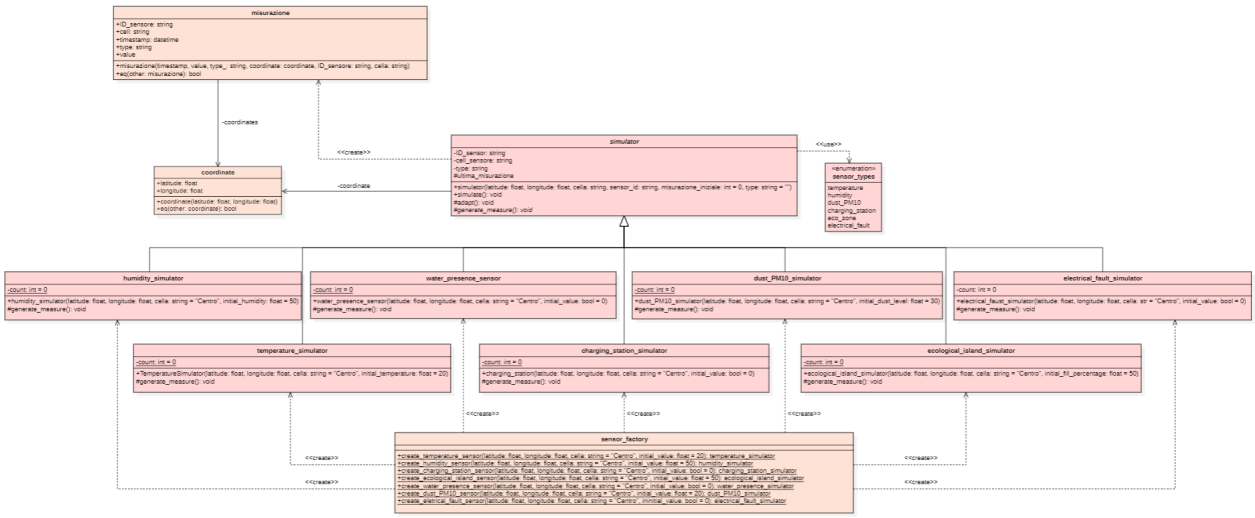

Figure 3: Modulo simulatori sensori - InnovaCity

Questo modulo si occupa della generazione di dati per diverse tipologie di sensori. In particolare sono stati implementati simulatori per i seguenti tipi di sensori:

- Sensori di temperatura;
- Sensori di umidità;
- Sensori di polveri sottili PM10;
- Sensori di stato occupazione colonnine di ricarica;
- Sensori di stato riempimento isole ecologiche;
- Sensori di presenza d'acqua;
- Sensori di guasto elettrico.

3.3.1.1 Design pattern Template Method

La classe astratta *simulator* implementa il design *pattern*$_G$ *Template Method*. Il metodo *simulate()* fornisce lo scheletro dell'algoritmo per la generazione delle misurazioni e non può essere ridefinito nelle implementatazioni. Le classi che estendono *simulator* implementano i metodi:

- ***generate_measure()*:** Per la generazione semi-randomica della misurazione associata al tipo di $sensore_G$, di fatto non è richiesta una generazione di misurazioni realistiche dal prodotto, ma per poter avere una visualizzazione finale delle misurazioni non altalenanti ogni misurazione viene generata sulla base di quella precedente con una variazione limitata;
- ***adapt()* - "hook method":** L'implementazione di default non modifica in alcun modo la misurazione generata. Può essere ridefinito per implementare la logica di adattamento della misurazione.

  Ad esempio:

  - Per i sensori di polveri sottili PM10, la misurazione può essere adattata cambiando il valore dell'unità di misura $\mu g/m^3$ in mg/m^3 senza modifcare la logica di generazione in *generate_measure()*;
  - Per i sensori di temperatura, la misurazione può essere adattata cambiando il valore dell'unità di misura da °C a Fahrenheit senza modifcare la logica di generazione in *generate_measure()*;
    - Fahrenheit = (Celsius × 9/5) + 32
  - Questo apre le porte alla possibilità di creare per una stessa tipoligia di $sensore_G$ diverse implementazioni, che generano misurazioni con unità di misura diverse, senza dover modificare la logica di generazione che dichiara l'unità di misura generata di default, ma semplicemente estendendo la classe e ridefinendo il metodo *adapt()* inserendo la formula di trasformazione.
- Per garantire il rispetto del **Liskov Substitution Principle (LSP)** *"Objects of a superclass should be replaceable with objects of its subtypes without altering the program's correctness."*:

  - Il metodo *adapt()* si limita a convertire le misurazioni senza alterare il comportamento generale del metodo *simulate()* e senza modificare il tipo della misurazione generata da *generate_measure()*;
  - Il metodo *generate_measure()* viene reso **final** nelle concretizzazioni di *simulator* impedendo la ridefinizione del comportamento.

  In generale le postcondizioni sono piu forti nelle classi derivate, mentre non variano le precondizioni garantendo il rispetto di *LSP*.

  Ad esempio:

  - In *simulator*:

    - La postcondizione del metodo *simulate()* in simulator è la generazione di un oggetto di tipo *misurazione*;
    - La postcondizione del metodo *generate_measure* è la generazione del valore della misurazione.
    - La postcondizione del metodo *adapt()* è la possibilità di convertire il valore della misurazione.
  - In *temperature_simulator*, che eredita da *simulator*:
    - La postcondizione del metodo *simulate()* è la generazione di un oggetto di tipo *misurazione* di temperatura;
    - La postcondizione del metodo *generate_measure* è la generazione del valore di una misurazione di temperatura;
    - La postcondizione del metodo *adapt()* è la possibilità di convertire il valore della misurazione ad un altra unità di misura (Kelvin, Fahrenheit).
  - In una possibile estensione futura *temperature_simulator_fahrenheit*, che eredita da *temperature_simulator*:
    - La postcondizione del metodo *simulate()* è la generazione di un oggetto di tipo *misurazione* di temperatura espresso in gradi Fahrenheit;
    - La postcondizione del metodo *generate_measure* rimane invariata;
    - La postcondizione del metodo *adapt()* è di convertire il valore della misurazione dall'unità di misura di default a Fahrenheit.

Al termine delle operazioni di generazione e adattamento, il metodo *simulate()* crea e restituisce un oggetto di tipo *misurazione*.
Il design *pattern*$_G$ *Template Method* è stato scelto per:

- Permettere una facile estensione del *sistema*$_G$ con nuovi tipi di sensori che dovranno unicamente implementare la loro logica di generazione delle misurazioni e di adapting se necessario;
- Standardizzare i passi per la generazione delle misurazioni, garantendo coerenza e manutenibilità del codice;
- Ridurre la duplicazione del codice.

Come già esposto, una volta ottenuto lo stato del *sensore*$_G$, esso viene inserito in un oggetto di tipo *misurazione*. Questo oggetto contiene informazioni di contesto come:

- Identificativo del *sensore*$_G$;
- Cella della città in cui è presente il *sensore*$_G$;

- Timestamp della misurazione;
- Valore della misurazione;
- Coordinate;
- Tipologia di misurazione.

L'oggetto *misurazione* viene poi ritornato al chiamante che si occuperà di inviarlo al server *Kafka*$_G$. Un oggetto di tipo *simulator* infatti, verrà assegnato ad ogni *simulator_thread*. Esso chiamerà ad intervalli regolari il metodo *simulate()* ottenendo appunto la misurazione che invierà poi al server *Kafka*$_G$ tramite un modulo apposito e indipendendente.

3.3.1.2 Design pattern Factory

sensor_factory implementa il design *pattern*$_G$ *Factory* per la creazione di simulatori dei sensori. Il *pattern*$_G$ *Factory* è un *pattern*$_G$ di tipo "Creazionale" secondo la classificazione della GoF.
I *pattern*$_G$ di tipo creazionali si occupano della costruzione delle simulazioni dei sensori e delle problematiche che si possono originare, astraggono il processo di creazione degli oggetti, nascondono i dettagli della creazione e rendono i sistemi indipendenti da come gli oggetti sono creati e composti.
Il *pattern*$_G$ *Factory* incapsula la creazione concreta dei sensori, consentendo al client (l'utilizzatore) di non conoscere i dettagli.

3.3.1.3 Classi, interfacce metodi e attributi:

- **Classe astratta: *simulator***
  - **Attributi**:
    - **ID_sensor: string [private]** - Identificatore univoco del *sensore*$_G$;
    - **cella_sensore: string [private]** - Identificatore della cella del *sensore*$_G$;
    - **coordinate: coordinate [private]** - Coordinate geografiche del *sensore*$_G$;
    - **misurazione: T [protected]** - Misurazione corrente del *sensore*$_G$;
    - **type: string [private]** - Tipo di *sensore*$_G$.
  - **Metodi**:
    - **simulate(): misurazione [public]** - Metodo principale per simulare la generazione di una misurazione.
    - Si basa sul design *pattern*$_G$ Template Method:
      1. Chiama generate_measure() per generare un valore di misurazione;
      2. Chiama adapt() per effettuare adattamenti se necessari;

3. Restituisce un oggetto *misurazione* con data e ora corrente, valore misurato, tipo di $sensore_G$, coordinate e identificativo del $sensore_G$.

* **generate_measure(): None [protected]** - Metodo astratto da implementare nelle classi concrete per generare un valore di misurazione semi-casuale coerente con la tipolgia di $sensore_G$ da salvare nell'attributo *misurazione*;
* **adapt(): void [protected]** - Fornisce un'implementazione di default che non modifica in alcun modo la misurazione generata. Può essere ridefinito per implementare la logica di adattamento della misurazione come conversioni di unità di misura.

– **Note**:

* La classe *simulator* è astratta e definisce il comportamento generale della simulazione della misurazione, $pattern_G$ *Template method*;
* Le classi concrete che ereditano da *simulator* devono implementare il metodo astratto generate_measure();
* Il metodo adapt() può essere ridefinito nelle classi concrete per implementare conversioni o adattamenti necessari;
* Il metodo *simulate()* è final e non può essere ridefinito;
* Spiegazioni esaustive sono state presentate in: 3.3.1.1.

- **Enumerazione: *sensor_types***

– **Costanti**:

* **TEMPERATURE: string [public]** - Rappresenta la nomenclatura dei $sensore_G$ di temperatura;
* **HUMIDITY: string [public]** - Rappresenta la nomenclatura dei $sensore_G$ di umidità;
* **DUST_PM10: string [public]** - Rappresenta la nomenclatura dei $sensore_G$ di "polvere PM10";
* **CHARGING_STATION: string [public]** - Rappresenta la nomenclatura dei $sensore_G$ di stato delle colonnine di ricarica;
* **ECOLOGICAL_ISLAND: string [public]** - Rappresenta la nomenclatura dei $sensore_G$ di stato riempimento isole ecologica;
* **WATER_PRESENCE: string [public]** - Rappresenta la nomenclatura dei $sensore_G$ di presenza d'acqua;
* **ELECTRICAL_FAULT: string [public]** - Rappresenta la nomenclatura dei $sensore_G$ di guasti elettrici.

– **Note**:

        - L'enumerazione viene utilizzata per centralizzare la gestione della nomenclatura dei tipi di sensori che verrà salvata nelle misurazioni.

- **Classe: *temperature_simulator***
  - **Attributi:**
    - **count: int [private, static]** - Contatore statico per generare un ID_G univoco per ogni istanza.
  - **Metodi**:
    - **generate_measure(): None [protected,final]** - Genera una misurazione di temperatura in gradi Celsius semi-casuale e aggiorna lo stato interno con il valore della misurazione corrente.
  - **Note**:
    - La classe *temperature_simulator* è una classe concreta che eredita dalla classe astratta *simulator*;
    - Il costruttore genera automaticamente un ID_G $sensore_G$ univoco per ogni istanza;
    - Dichiara di generare misurazioni di temperatura con unità di default (Gradi Celsius), possibili classi derivate possono effettuare conversioni ad altre unità di misura (Kelvin,Fahrenheit) tramite il metodo *adapt()*, senza dover modificare la logica di generazione.

- **Classe: *humidity_simulator***
  - **Attributi:**
    - **count: int [private, static]** - Contatore statico per generare un ID_G univoco per ogni istanza.
  - **Metodi**:
    - **generate_measure(): None [protected,final]** - Genera una misurazione di umidità in percentuale semi-casuale e aggiorna lo stato interno con il valore della misurazione corrente.
  - **Note**:
    - La classe *humidity_simulator* è una classe concreta che eredita dalla classe astratta *simulator*;
    - Il costruttore genera automaticamente un ID_G $sensore_G$ univoco per ogni istanza;

    - Dichiara di generare misurazioni di umidità con unità di default (Percentuale), possibili classi derivate possono effettuare conversioni ad altre unità di misura (g/m^3) tramite il metodo *adapt()*, senza dover modificare la logica di generazione.

- **Classe: *charging_station_simulator***
  - **Attributi**:
    - **count: int [private, static]** - Contatore statico per generare un ID_G univoco per ogni istanza.
  - **Metodi**:
    - **generate_measure(): None [protected,final]** - Genera lo stato della colonnina di ricarica (Occupato: TRUE, Libero: FALSE) basata su una probabilità di transizione e aggiorna lo stato interno con il valore della misurazione corrente.
  - **Note**:
    - La classe *charging_station_simulator* è una classe concreta che eredita dalla classe astratta *simulator*;
    - Il costruttore genera automaticamente un ID_G $sensore_G$ univoco per ogni istanza;
    - Implementa il metodo astratto *generate_measure()* per generare una misurazione basata sulla probabilità di transizione.

- **Classe: *dust_PM10_simulator***
  - **Attributi**:
    - **count: int [private, static]** - Contatore statico per generare un ID_G univoco per ogni istanza.
  - **Metodi**:
    - **generate_measure(): None [protected,final]** - Genera una variazione di quantità di polvere PM10 semi-casuale e aggiorna lo stato interno con il valore della misurazione corrente.
  - **Note**:
    - La classe *dust_PM10_simulator* è una classe concreta che eredita dalla classe astratta *simulator*;
    - Il costruttore genera automaticamente un ID_G $sensore_G$ univoco per ogni istanza;

        * Dichiara di generare misurazioni di polvere PM10 in con unità di default ($\mu g/m^3$), possibili classi derivate possono effettuare conversioni ad altre unità di misura (mg/m^3) tramite il metodo *adapt()*, senza dover modificare la logica di generazione.

- **Classe: *electrical_fault_simulator***
  - **Attributi**:
    * **count: int [private, static]** - Contatore statico per generare un ID_G univoco per ogni istanza.
  - **Metodi**:
    * **generate_measure(): None [protected,final]** - Genera lo stato di una centralina elettrica (Guasto verificato: TRUE, Operativa: FALSE) basandosi su una probabilità di guasto e aggiorna lo stato interno con il valore della misurazione corrente.
  - **Note**:
    * La classe *electrical_fault_simulator* è una classe concreta che eredita dalla classe astratta *simulator*;
    * Il costruttore genera automaticamente un ID_G $sensore_G$ univoco per ogni istanza.

- **Classe: *ecological_island_simulator***
  - **Attributi**:
    * **count: int [private, static]** - Contatore statico per generare un ID_G univoco per ogni istanza.
  - **Metodi**:
    * **generate_measure(): None [protected,final]** - Genera una misurazione della percentuale di riempimento di un isola ecologica e aggiorna lo stato interno con il valore della misurazione corrente.
  - **Note**:
    * La classe *ecological_island_simulator* è una classe concreta che eredita dalla classe astratta *simulator*;
    * Il costruttore genera automaticamente un ID_G $sensore_G$ univoco per ogni istanza.

- **Classe: *water_presence_sensor***
  - **Attributi**:

        - **count: int [private, static]** - Contatore statico per generare un ID_G univoco per ogni istanza.
    - **Metodi**:
        - **generate_measure(): None [protected,final]** - Genera una misurazione basata sulla soglia di presenza dell'acqua (Acqua rilevata: TRUE, Acqua non rilevata: FALSE) e aggiorna lo stato interno con il valore della misurazione corrente.
    - **Note**:
        - La classe *ecological_island_simulator* è una classe concreta che eredita dalla classe astratta *simulator*;
        - Il costruttore genera automaticamente un ID_G $sensore_G$ univoco per ogni istanza.

- **Classe: *misurazione***
    - **Attributi**:
        - **timestamp: datetime [private]** - Timestamp della misurazione;
        - **value: T [private]** - Valore della misurazione;
        - **type: string [private]** - Tipo della misurazione;
        - **coord: coordinate [private]** - Coordinate della misurazione;
        - **ID_sensore: string [private]** - ID_G del $sensore_G$ che ha effettuato la misurazione;
        - **cella: string [private]** - Cella in cui è stata effettuata la misurazione.
    - **Metodi**:
        - **__eq__(other: misurazione): bool [public]** - Ridefinizione dell'operatore di uguaglianza per confrontare due oggetti *misurazione*.

- **Classe: *coordinate***
    - **Attributi**:
        - **latitude: float [private]** - Latitudine della coordinata;
        - **longitude: float [private]** - Longitudine della coordinata.
    - **Metodi**:
        - **__eq__(other: coordinate):bool [public]** - Ridefinizione dell'operatore di uguaglianza per confrontare due oggetti *coordinate*.

- **Classe: *sensor_factory***
    - **Metodi**:

* **create_temperature_sensor(latitude: float, longitude: float, cella: string, initial_value: float): temperature_simulator [public, static]** - Crea un simulatore di temperatura;
* **create_humidity_sensor(latitude: float, longitude: float, cella: string, initial_value: float): humidity_simulator [public, static]** - Crea un simulatore di umidità;
* **create_charging_station_sensor(latitude: float, longitude: float, cella: string, initial_value: bool): charging_station_simulator [public, static]** - Crea un simulatore di stazione di ricarica;
* **create_ecological_island_sensor(latitude: float, longitude: float, cella: string, initial_value: float): ecological_island_simulator [public, static]** - Crea un simulatore di isola ecologica;
* **create_water_presence_sensor(latitude: float, longitude: float, cella: string, initial_value: bool): water_presence_sensor [public, static]** - Crea un *sensore*$_G$ di presenza d'acqua;
* **create_dust_PM10_sensor(latitude: float, longitude: float, cella: string, initial_value: float): dust_PM10_simulator [public, static]** - Crea un simulatore di polvere PM10;
* **create_eletrical_fault_sensor(latitude: float, longitude: float, cella: string, initial_value: bool): electrical_fault_simulator [public, static]** - Crea un simulatore di guasto elettrico.

Note:

* Implementazione del *pattern*$_G$ Factory;
* Fornisce metodi per la creazione di simulatori di sensori;
* Astrae il processo di creazione dei sensori, nascondendo i dettagli della creazione.

3.3.2 Modulo Writers

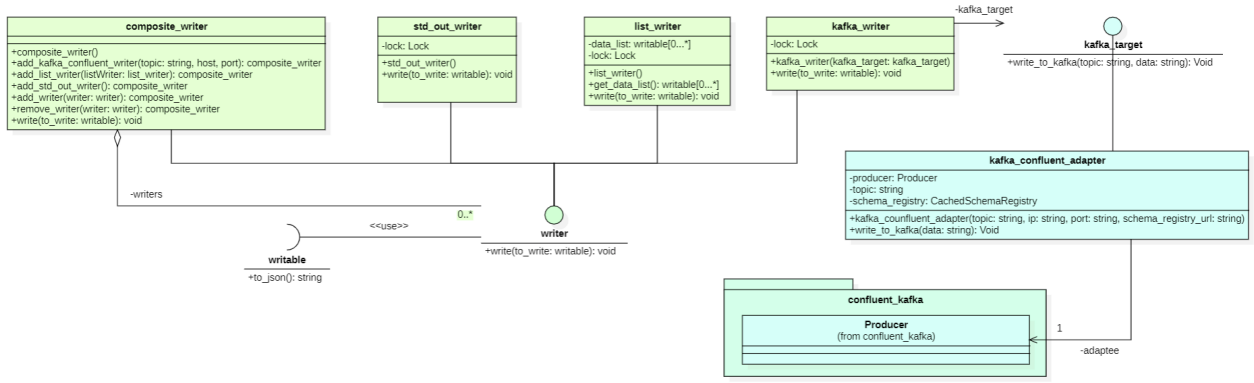

Figure 4: Modulo writers - InnovaCity

Questo modulo si occupa della scrittura e/o invio di informazioni a diverse tipologie di servizi e vuole essere completamente indipendente e non influenzato dal modulo della simulazione dei sensori, così da poter consentire un suo riutilizzo. Per quanto riguarda la scrittura in *Kafka*$_G$, l'impiego della connessione allo Schema Registry (vedi sezione §3.5) consente la convalida del formato del messaggio prima della sua scrittura nei topic *Kafka*$_G$. Questa pratica permette di ridurre il carico di *rete*$_G$ nel caso di messaggi malformati, consentendo un filtraggio "alla fonte".
Il modulo è stato progettato per rispettare il Dependecy Inversion Principle *(DIP)*, di conseguenza sia i moduli di alto livello che quelli di basso livello dipendono da astrazioni (interfacce o classi astratte).

3.3.2.1 Design pattern Strategy + Composite:

Il modulo presenta un'interfaccia *writer* che offre il metodo di scrittura *write()* di oggetti che implementano *writable*. Questo metodo è implementato da diverse classi concrete che rappresentano i vari servizi a cui è possibile inviare le informazioni. L'approccio adottato implementa il design *pattern*$_G$ *Strategy* per la scrittura/invio dei dati su diverse piattaforme/servizi e il design *pattern*$_G$ *Composite* per la scrittura ad uno o più servizi in modo uniforme. Nello specifico sono state implentate tre strategie di scrittura: la prima, (*kafka_writer*), atta a permettere al simulatore di inviare messaggi a topic *Kafka*$_G$, la seconda (*std_out_writer*) atta a permettere di stampare gli *writable* su terminale e la terza (*list_writer*) per il salvataggio su una lista degli *writable*. L'utilizzo del design *pattern*$_G$ Composite e Strategy in questo caso ha diverse motivazioni:

1. *kafka_writer*, atta a permettere al simulatore di inviare messaggi a *Kafka*$_G$;

2. *std_out_writer*, atta a permettere di stampare i *writable* su terminale;
3. *list_writer*, per il salvataggio su una lista degli oggetti di tipo *writable*.

L'utilizzo dei design *pattern*$_G$ Composite e Strategy in questo caso ha diverse motivazioni:

- **Gestione uniforme dei servizi**: Il *pattern*$_G$ Strategy consente di definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. In questo caso, i servizi di scrittura sono trattati come algoritmi intercambiabili, consentendo di scrivere informazioni su diversi servizi senza dover conoscere i dettagli di implementazione di ciascuno;
- **Gestione gerarchica dei servizi**: Il *pattern*$_G$ Composite consente di trattare gli oggetti singoli e le loro composizioni (gruppi di oggetti) allo stesso modo.
  Nel contesto del modulo, potrebbe esserci la necessità di gestire non solo singoli servizi, ma anche gruppi di servizi. Ad esempio, potrebbe essere utile inviare informazioni al *servizio*$_G$ *Kafka*$_G$ e contemporaneamente stamparle nel terminale e memorizzarle in un'apposita lista per il testing. Il Composite consente di comporre questi servizi in modo gerarchico e trattarli uniformemente.

3.3.2.2 Design pattern Object Adapter:

Nello specifico, la classe *kafka_writer* realizza la sua funzionalità attraverso l'utilizzo del design *pattern*$_G$ *Adapter*, nella sua variante *Object Adapter*. Tale scelta è stata motivata dall'impiego della classe *Producer* della libreria *confluent_kafka*, la quale potrebbe subire variazioni non controllabili da noi. Per garantire la capacità di rispondere prontamente a tali cambiamenti senza dover modificare la classe *kafka_writer* o altri parti di *sistema*$_G$, si è optato per l'utilizzo di questo *pattern*$_G$, trasferendo così la complessità derivante da tali modifiche proprio nell'adapter. Inoltre grazie all'interfaccia *kafka_target*, si è garantita la possibilità di estendere il *sistema*$_G$ con nuovi metodi di scrittura su *Kafka*$_G$ o l'utilizzo di nuove *librerie*$_G$ senza dover modificare la classe *kafka_writer* ma solamente aggiungendo una nuova classe adapter che implementi *kafka_target*.

3.3.2.3 Classi: metodi e attributi

- **Interfaccia: *writable***
  - **Metodi**:
    - **to_json(): string [public, abstract]** - Metodo astratto che deve essere implementato nelle sottoclassi per convertire l'oggetto in una stringa *JSON*$_G$.
  - **Note**:
    - L'interfaccia *writable* definisce un insieme di metodi che una classe deve implementare perchè possa essere utilizzata dalle strategie di scrittura.

- **Interfaccia: *writer***
  - **Metodi**:
    - **write(to_write: writable): None [public, abstract]** - Metodo astratto che deve essere implementato nelle sottoclassi per scrivere un oggetto writable.
  - **Note**:
    - L'interfaccia *writer* definisce il metodo che una classe deve implementare perchè possa essere utilizzata come strategia di scrittura. Rappresenta il componente "*strategy*" del *pattern*$_G$ "*Strategy*";
    - Rappresenta l'interfaccia "*Component*" del *pattern*$_G$ *Composite* che descrive le operazioni comuni sia agli elementi semplici che a quelli complessi dell'albero.
- **Classe: *composite_writer***
  - **Attributi**:
    - **writers: writer [private]** - Lista di oggetti *writer*.
  - **Metodi**:
    - **add_writer(writer: writer): composite_writer [public]** - Aggiunge un oggetto che implementa *writer* alla lista writers;
    - **add_kafka_confluent_writer(topic: string, host: string, port: int, schema_registry_url: string): composite_writer [public]** - Crea un *kafka_writer* con un *kafka_confluent_adapter* e lo aggiunge alla lista writers. Ritorna se stesso per permettere operazione concatenate;
    - **add_std_out_writer(): composite_writer [public]** - Crea un *std_out_writer* e lo aggiunge alla lista writers. Ritorna se stesso per permettere operazione concatenate;
    - **add_list_writer(writer_list: list_writer): composite_writer [public]** - Aggiunge un *list_writer* alla lista writers. Ritorna se stesso per permettere operazione concatenate;
    - **remove_writer(writer: writer): composite_writer [public]** - Rimuove un *writer* dalla lista writers. Ritorna se stesso per permettere operazione concatenate;
    - **write(to_write: writable): *composite_writer* [public]** - Chiama il metodo write su ogni writer nella lista writers passando come attributo il *writable* ricevuto.
  - **Note**:
    - La classe è la componente "Composite" del *pattern*$_G$ *Composite*, ovvero l'elemento che può avere sottoelementi;
    - Dopo aver ricevuto una richiesta, il contenitore (detto composite) delega il lavoro ai suoi sottoelementi: foglie o altri contenitori.

- **Classe: *std_out_writer***
  - **Attributi**:
    - **lock:threading.Lock [private]** - Lock per garantire l'accesso esclusivo alla stampa ed un esecuzione Thread safe.
  - **Metodi**;
    - **write(to_write: writable): None [public]** - Stampa l'oggetto writable come stringa *JSON*$_G$ nella console;
  - **Note**:
    - La classe è una strategia di scrittura del *pattern*$_G$ *Strategy* ma anche la componente "Leaf" del *pattern*$_G$ *Composite*, ovvero l'elemento base che non ha sottoelementi.
- **Classe: *list_writer***
  - **Attributi**:
    - **data_list:list [private]** - Lista per memorizzare gli oggetti writable;
    - **lock:threading.Lock [private]** - Lock per garantire l'accesso esclusivo alla lista ed un esecuzione Thread safe.
  - **Metodi**:
    - **write(to_write: writable): None [public]** - Aggiunge l'oggetto writable alla lista;
    - **get_data_list(): list [public]** - Restituisce la lista di oggetti writable.
  - **Note**:
    - La classe è una strategia di scrittura del *pattern*$_G$ *Strategy* ma anche la componente "Leaf" del *pattern*$_G$ *Composite*, ovvero l'elemento base che non ha sottoelementi.
- **Classe: *kafka_writer***
  - **Attributi**:
    - **lock:threading.Lock [private]** - Lock per garantire l'accesso esclusivo alla scrittura su *Kafka*$_G$ ed un esecuzione Thread safe;
    - **kafka_target:kafka_target [private]** - Riferimento ad un implementazione di *kafka*$_G$_target per effettuare l'effettiva scrittura in *Kafka*$_G$ tramite *librerie*$_G$.
  - **Metodi**:
    - **write(to_write: writable): None [public]** - Scrive l'oggetto *writable* come stringa *JSON*$_G$ su *Kafka*$_G$.

  - **Note**:
    - La classe è una strategia di scrittura del *pattern*$_G$ *Strategy* ma anche la componente "Leaf" del *pattern*$_G$ *Composite*, ovvero l'elemento base che non ha sottoelementi;
    - La costruzione dell'oggetto *kafka_writer* richiede un riferimento ad un oggetto che implementi l'interfaccia *kafka*$_G$_target.

- **Interfaccia: kafka_target**
  - **Metodi**:
    - **write_to_kafka(data: string): None [public, abstract]** - Metodo astratto che deve essere implementato nelle sottoclassi per scrivere dati su *Kafka*$_G$.
  - **Note**:
    - La classe è una interfaccia che fornisce un contratto per le operazioni di scrittura/invio a topic *Kafka*$_G$;
    - Rappresenta il componente Target del *pattern*$_G$ *Object Adapter*.

- **Classe: *KafkaConfluentAdapter***
  - **Attributi**:
    - **topic:string [private]** - Il topic su cui scrivere in *Kafka*$_G$;
    - **producer:Producer [private]** - Il producer *Kafka*$_G$ per inviare messaggi;
    - **schema_registry:CachedSchemaRegistryClient [private]** - Consente di interagire con Confluent Schema Registry in modo efficiente, memorizza in cache gli schemi recuperati da Schema Registry, riducendo le chiamate di *rete*$_G$ e migliorando la velocità di accesso agli schemi.

- **Metodi**:
  - **write_to_kafka(data: string): None [public]** - Scrive i dati su *Kafka*$_G$ dopo averli validati rispetto allo schema registrato per il topic di destinazione.
  - **Note**:
    - La classe è un'implementazione concreta dell'interfaccia *kafka_target*, utilizzando la libreria *confluent-kafka* per interagire con *Kafka*$_G$;
    - Rappresenta il componente "Adapter" del *pattern*$_G$ *Object Adapter*;
    - Il Producer *kafka*$_G$ rappresenta la componente "service" del *pattern*$_G$ *Object Adapter*;
    - Prima di inviare il messaggio (ovvero la misurazione) controlla che sia nel formato/schema corretto per il topic di destinazione evintando cosi sovraccarico inutile di *rete*$_G$;
    - Nel caso in cui il formato del messaggio sia scorretto questo viene scartato.

3.3.3 Modulo Threading/Scheduling

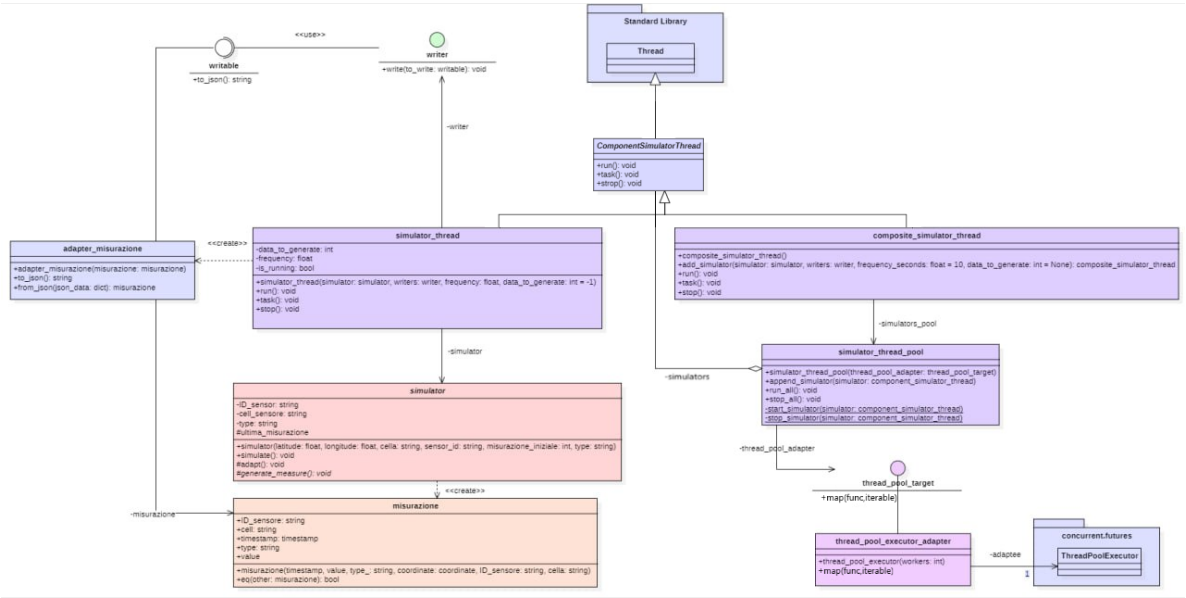

Figure 5: Modulo Threading/Scheduling simulatori sensori - InnovaCity

Questo modulo si propone di gestire la logica di pianificazione per il recupero dei dati dai simulatori dei sensori e di inviare/scrivere tali dati utilizzando il modulo Writer.
Funge da orchestratore per i due moduli appena descritti, offrendo la possibilità di configurare la frequenza di campionamento e il numero di misurazioni da eseguire.
Inoltre, incorpora una logica di ottimizzazione, simile a quella impiegata dai microcontrollori dei sensori nella realtà, al fine di evitare la trasmissione di dati ridondanti, inviando solo i cambiamenti di stato dei sensori.
Il modulo è stato progettato per rispettare il Dependecy Inversion Principle *(DIP)*, di conseguenza sia i moduli di alto livello che quelli di basso livello dipendono da astrazioni (interfacce o classi astratte).

3.3.3.1 Design pattern Composite:

Come il modulo di scrittura anche questo è sviluppato secondo il *pattern*$_G$ *Composite* che permette di gestire un singolo thread di esecuzione o un gruppo di thread in modo uniforme.

3.3.3.2 Design pattern Object Adapter:

Inoltre, considerando l'impiego di più thread per un esecuzione parallela, per delegare l'orchestrazione delle operazioni, si è deciso di utilizzare delle ThreadPool.
Al fine di evitare modifiche dirette al codice di *simulator_thread_pool*, è stato adottato il *pattern*$_G$ *Object Adapter* per adattare la ThreadPool di *Python*$_G$ a un'interfaccia comune con cui *simulator_thread_pool* possa interagire.
Questo approccio consente di modificare la logica o la libreria utilizzata per la gestione dei thread senza richiedere modifiche al codice di *simulator_thread_pool*, ma semplicemente aggiungendo una nuova classe adapter che implementi *thread_pool_target*.
Un'altra implementazione del *pattern*$_G$ *Object Adapter* viene impiegata per adattare gli oggetti *misurazione* del modulo dei Simulatori all'interfaccia *writable* del modulo Writers. La classe *adapter_misurazione*, implementando l'interfaccia *writable*, fornisce un'implementazione del metodo *to_json()* che consente di convertire un oggetto *misurazione* nel formato *JSON*$_G$, compatibile con il formato definito nello Schema Registry e riconosciuto da *Kafka*$_G$. 3.4.2

3.3.3.3 Classi, interfacce, metodi e attributi

- **Classe astratta: *component_simulator_thread***
  - **Metodi:**
    - **run(): None [public, abstract]** - Metodo astratto che deve essere implementato nelle sottoclassi per definire il comportamento del thread quando viene avviato;
    - **task(): None [public, abstract]** - Metodo astratto che deve essere implementato nelle sottoclassi per definire il compito specifico che il thread deve eseguire;
    - **stop(): None [public, abstract]** - Metodo astratto che deve essere implementato nelle sottoclassi per definire come fermare il thread.
  - **Note**:
    - Eredita le proprietà e i metodi della classe *Thread* della *Standard Library*;
    - *component_simulator_thread* è una classe astratta di threading per la simulazione dei sensori, fornisce un contratto per le operazioni di avvio, esecuzione del compito e arresto;
    - Rappresenta il componente "Component" del *pattern*$_G$ *Composite*, descrive le operazioni comuni sia ai singoli Thread sia a composizioni di questi;
    - L'utilizzatore dei simulatori può lavorare allo stesso modo con elementi semplici (singoli Thread) o complessi (insiemi di Thread in forma di albero).
- **Classe: *simulator_thread***
  - **Attributi**:

    - **simulator: *simulator* [private]** - Il simulatore da utilizzare per generare i dati;
    - **frequency: float [private]** - La frequenza con cui generare i dati;
    - **is_running: bool [private]** - Flag per controllare se il thread è in esecuzione;
    - **data_to_generate: int [private]** - Il numero di dati da generare;
    - **writers: writer [private]** - L'oggetto implementazione di *writer* per scrivere i dati generati. (Singolo o albero - Composite *pattern*$_G$)
- **Metodi:**
    - **run(): None [public]** - Avvia il thread del simulatore;
    - **task(): None [public]** - Definisce il compito specifico che il thread deve eseguire, contiene la logica per generare il numero di misurazioni richieste con l'intervallo specificato alla costruzione. Inoltre evita l'invio di misurazioni consecutive uguali cosi da ridurre il carico scartando dati ridondanti e deducibili inviando ai *Writers* solo i cambi di stato del *sensore*$_G$ da cui acquisisce la misurazione. All'interno del metodo, la misurazione restituita dal simulatore, viene adattata ad un oggetto *writable* tramite *adapter_misurazione* ed inviata ai *Writers*;
    - **stop(): None [public]** - Ferma il thread del simulatore.
- **Note**:
    - La classe è un'implementazione concreta della classe astratta *component_simulator_thread*;
    - Utilizza un oggetto *simulator* per generare dati a una certa frequenza e un oggetto che implementa*component_writer* per scrivere i dati generati;
    - Rappresenta il componente Leaf del *pattern*$_G$ *Composite*;
    - Se *data_to_generate* < 0, allora genera misurazioni finchè il thread non viene interroto dall'esterno;
    - Sebbene i simulatori non siano considerati dalla *proponente*$_G$ parte del prodotto, la logica di ottimizzazione per inviare solo i cambi di stato dei sensori viene implementata nella realtà IoT.
      Di conseguenza, è stata presa la decisione di replicarla. È importante notare che questa logica non è incorporata nel simulatore del *sensore*$_G$, il quale ha unicamente il compito semantico di generare dati come un vero *sensore*$_G$. Invece, essa è implementata in *simulator_thread*, il quale agisce in modo simile a un microcontrollore, responsabile sia della gestione dell'intervallo di campionamento che della logica per l'invio delle misurazioni;
    - Nel corso dello sviluppo futuro, potrebbe risultare vantaggioso considerare l'implementazione di un *pattern*$_G$ *Strategy* per gestire la strategia/criterio di

invio dei dati, che possa distinguere tra un invio continuo e la trasmissione solo in caso di cambiamenti di stato. Tuttavia, al momento della decisione, si è optato per non includerlo al fine di evitare un'eccessiva complessità nell'*architettura*$_G$, nota come sovraingegnerizzazione. Tale scelta è stata dettata dalla volontà di mantenere un equilibrio tra la completezza del *sistema*$_G$ e la sua semplicità, favorendo un'implementazione più diretta e immediata delle funzionalità richieste.

- **Classe: *adapter_misurazione***
  - **Attributi**:
    - **misurazione: *misurazione* [private]** - L'oggetto *misurazione* da adattare.
  - **Metodi:**
    - **to_json(): string [public]** - Converte l'oggetto *misurazione* in una stringa *JSON*$_G$ conforme a quanto definito in 3.4.2;
    - **from_json(json_data: dict): misurazione [staticmethod, public]** - Crea un oggetto *misurazione* da un dizionario *JSON*$_G$.
  - **Note**:
    - La classe è un'implementazione concreta dell'interfaccia *writable*. Fornisce metodi per convertire un oggetto *misurazione* in un formato *JSON*$_G$ e viceversa;
    - Rappresenta la componente "Adapter" del *pattern*$_G$ *Object Adapter*.
- **Classe: *composite_simulator_thread***
  - **Attributi**:
    - **simulator_executor: simulator_thread_pool [private]** - L'executor per gestire l'esecuzione di più thread dei simulatori.
  - **Metodi:**
    - **add_simulator(simulator: simulator, writers: component_writer, frequency: float, data_to_generate: int): composite_simulator_thread [public]** - Aggiunge un simulatore all'executor;
    - **add_simulator_thread(thread_simulator: component_simulator_thread): composite_simulator_thread [public]** - Aggiunge un *simulator_thread* all'executor;
    - **run(): None [public]** - Avvia tutti i *simulator_thread* nell'executor;
    - **stop(): None [public]** - Ferma tutti i *simulator_thread* nell'executor;
    - **task(): None [public]** - Avvia tutti i *simulator_thread* nell'executor.
  - **Note**:

    - La classe è un'implementazione concreta della classe astratta *component_simulator_thread*, utilizza un oggetto *simulator_thread_pool* per gestire l'esecuzione di vari simulatori;
    - Rappresenta il componente "Composite" del *pattern$_G$ Composite*.
    - Utilizzando il metodo *run()* viene creato un nuovo thread che permette di eseguire la funzione *task()* in modo sincrono.
    - Utilizzando il metodo *task()* direttamente non è possibile avere un'esecuzione asincrona. La funzione *task()* viene eseguita sul thread chiamante, bloccando l'esecuzione del codice fino al termine della sua esecuzione. Questa funzione viene utilizzata dalle ThreadPool che creano dei thread per l'esecuzione asincrona delle task.

- **Classe: *simulator_thread_pool***
  - **Attributi:**
    - **simulators: List[component_simulator_thread] [private]** - La lista dei *component_simulator_thread* da eseguire (Singoli thread o alberi di thread);
    - **thread_pool_adapter: thread_pool_target [private]** - Thread pool per gestire l'esecuzione parallela dei simulatori.
  - **Metodi:**
    - **run_all(): None [public]** - Avvia tutti i simulatori nella thread pool, utilizzando l'interfaccia fornita da *thread_pool_target* per l'esecuzione controllata di *attività$_G$* in parallelo. Per farlo viene utilizzato il metodo *map()* di *thread_pool_target* per mappare la funzione statica *start_simulator()* su ogni *component_simulator_thread* in *simulators*;
    - **stop_all(): None [public]** - Ferma tutti i simulatori nel thread pool, utilizzando l'interfaccia fornita da *thread_pool_target* per l'esecuzione controllata di *attività$_G$* in parallelo. Per farlo utilizza il metodo *map()* di *thread_pool_target* per mappare la funzione statica *stop_simulator()* su ogni *component_simulator_thread* in *simulators*;
    - **append_simulator(simulator: component_simulator_thread): None [public]** - Aggiunge un *component_simulator_thread* alla threadpool;
    - **start_simulator(simulator: component_simulator_thread): None [private, static]** - Avvia un *component_simulator_thread*;
    - **stop_simulator(simulator: component_simulator_thread): None [private, static]** - Ferma un *component_simulator_thread*.
  - **Note**:

    - La classe gestisce una pool di thread per l'esecuzione di vari simulatori, utilizzando un oggetto che implementa *thread_pool_target* per gestire l'esecuzione dei simulatori;
    - I metodi *run_all()* e *stop_all()* utilizzano l'interfaccia fornita da *thread_pool_target* per mappare rispettivamente la funzione statica *start_simulator()* e *stop_simulator()* per ogni *component_simulator_thread* in *simulators*;
    - Grazie all'utilizzo di *thread_pool_target* è possibile estendere il *sistema*$_G$ con nuovi metodi o utilizzare nuove *librerie*$_G$ senza dover modificare la classe *simulator_thread_pool*, ma solamente aggiungendo una nuova classe adapter che implementi *thread_pool_target*.

- **Interfaccia: *thread_pool_target***
  - **Metodi:**
    - **map(func, iterable): [abstractmethod]** - Un metodo astratto che deve essere implementato nelle classi derivate. Questo metodo applica la funzione *func* a ogni elemento nell'*iterable*.
  - **Note**:
    - L'interfaccia rappresenta la componente "Target" del *pattern*$_G$ *Object Adapter* fornendo un contratto per le operazioni di esecuzione controllata di *attività*$_G$ in parallelo.

- **Classe: *thred_pool_executor_adapter***
  - **Attributi:**
    - **executor: concurrent.futures.ThreadPoolExecutor [private]** - L'executor della thread pool per gestire l'esecuzione dei thread dalla libreria *concurrent.futures*.
  - **Metodi:**
    - **map(func, iterable): [public]** - Applica la funzione *func* a ogni elemento nell'*iterable* utilizzando l'executor del thread pool (*executor*).
  - **Note**:
    - La classe è un'implementazione concreta dell'interfaccia *thread_pool_target*, utilizzando un oggetto *concurrent.futures.ThreadPoolExecutor* per gestire l'esecuzione dei thread;
    - Rappresenta il componente "Adapter" del *pattern*$_G$ *Object Adapter*;
    - Adatta l'oggetto ThreadPoolExecutor dalla libreria *concurrent.futures*;
    - Al momento della costruzione deve essere fornito il parametro intero "workers" ovvero Il numero massimo di thread che è possibile utilizzare per eseguire le task indicate.

3.3.4 Progettazione - Panoramica UML

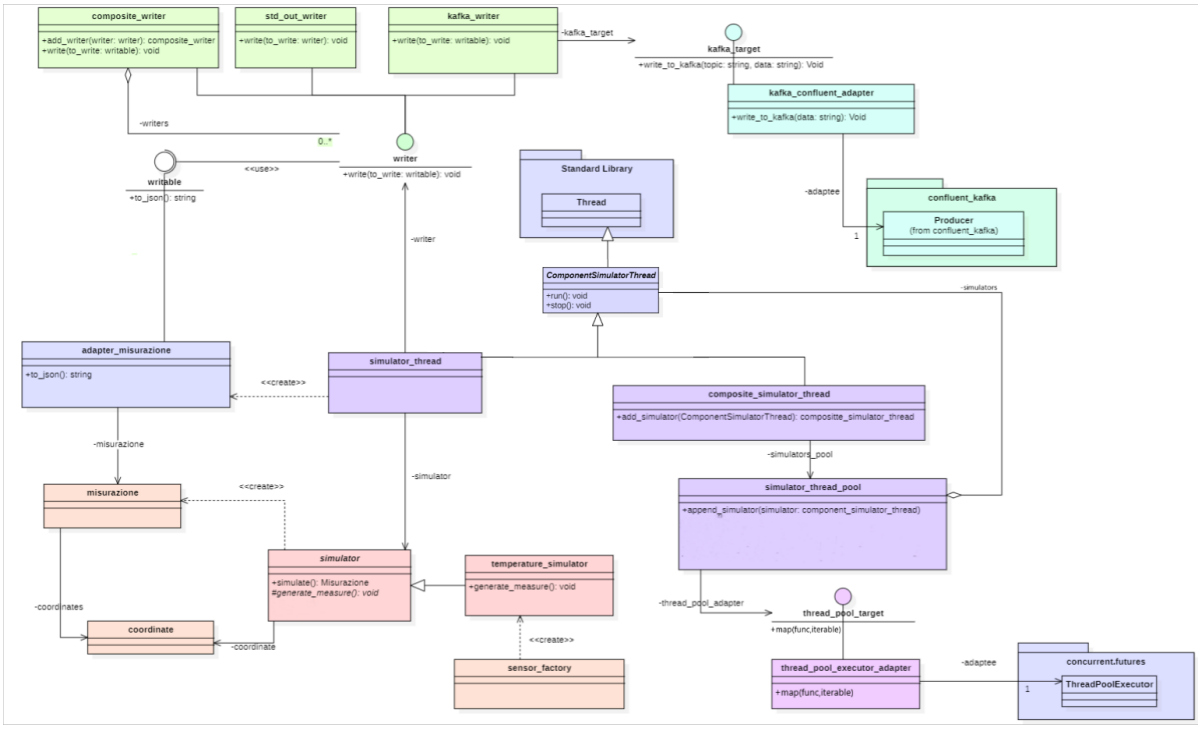

Figure 6: Panoramica progettazione simulatori sensori UML - InnovaCity

L'immagine vuole mostrare sinteticamente la struttura dei moduli Simulatori, Writers e Threading/Scheduling e le relazioni tra le classi principali di questi moduli.
In particolare, si evidenzia la presenza del *pattern*$_G$ *Composite* per la gestione di più servizi di scrittura e di più thread di esecuzione, il *pattern*$_G$ *Strategy* per la scrittura su diversi servizi e il *pattern*$_G$ *Object Adapter* per adattare le ThreadPool di *Python*$_G$ e gli oggetti *misurazione* ai *writer*.
Viene riportato solo il *sensore*$_G$ di temperatura come implementazione concreta di *simulator* in quanto le altre implementazioni sono analoghe.

3.4 Apache Kafka

3.4.1 Kafka topic

I topic in *Kafka*$_G$ possono essere considerati come le tabelle di un *database*$_G$, utili per separare logicamente diversi tipi di messaggi o eventi che vengono inseriti nel *sistema*$_G$. Nel nostro caso vengono utilizzati per separare le diverse misurazioni dei sensori, quindi per ogni tipologia di *sensore*$_G$ è presente un topic dedicato.
Ciò ci consente di creare all'interno di *ClickHouse*$_G$ delle "tabelle consumatrici" che acquisiscono automaticamente i dati.

3.4.2 Formato messaggi

I messaggi vengono trasmessi in formato JSON, la loro struttura contenente le informazioni della misurazione, rispetta il contratto definito nello Schema Registry (sez.3.5, in particolare 3.5.1) ed è la seguente:

```
1  {
2    "timestamp": "AAAA-MM-DD HH:MM:SS.sss",
3    "value": "Valore della misurazione",
4    "type": "Tipologia Simulatore",
5    "latitude": "Latitudine",
6    "longitude": "Longitudine",
7    "ID_sensore": "ID sensore",
8    "cella": "Partizione della citta‘ dove e‘ presente il sensore"
9   }
```

Mentre la struttura JSON di un messaggio contenente di una misurazione del punteggio di salute è la seguente:

```
1  {
2    "timestamp": "AAAA-MM-DD HH:MM:SS.sss",
3    "value": "Valore della misurazione",
4    "type": "Tipologia Simulatore",
5    "cella": "Cella relativa al punteggio di salute"
6  }
```

Sebbene le misurazioni vengano divise in topic diversi a seconda della tipoligia di *sensore*$_G$, si è comunque deciso di inviare e salvare il campo della tipoligia di misurazione per i seguenti motivi:

- **Backup e ripristino dei dati:** Se si dovesse perdere la struttura dei topic o fosse necessario ripristinare i dati in un altro *sistema*$_G$, il campo *type* aiutarebbe a identificare il tipo di *sensore*$_G$ che ha effettuato la misurazione, anche se i dati sono stati conservati in un unico topic.
- **Flessibilità futura:**
  - Potrebbero sorgere esigenze future che richiedono l'analisi dei dati provenienti da diversi tipi di sensori all'interno dello stesso topic. In questo caso, il campo *type* sarebbe utile per distinguere le misurazioni provenienti da sensori diversi;
  - Includere il campo *type* potrebbe essere particolarmente utile se si prevede di supportare diverse unità di misura per una stessa tipologia di *sensore*$_G$ in futuro. Ad esempio, potrebbe essere necessario gestire misurazioni di temperatura in gradi Celsius, Fahrenheit o Kelvin nello stesso topic. In tal caso, includendo il campo *type*, si può associare ad ogni misurazione l'unità di misura corretta.

3.5 Schema Registry

Documentazione
https://docs.confluent.io/platform/current/schema-registry/index.html (Consultato 25/03/2024).
Schema Registry fornisce un *repository*$_G$ centralizzato per la gestione e la convalida degli schemi relativi ai messaggi *Kafka*$_G$, nonché per la serializzazione e la deserializzazione dei dati sulla *rete*$_G$. I produttori e i consumatori degli argomenti *Kafka*$_G$ possono sfruttare gli schemi per garantire la coerenza e la compatibilità dei dati mentre questi ultimi si evolvono nel tempo.
Lo Schema Registry rappresenta un elemento chiave per la governance dei dati, poiché contribuisce ad assicurare la loro qualità, la tracciabilità dell'origine dei dati, la conformità agli *standard*$_G$, la collaborazione tra team, protocolli di sviluppo delle applicazioni efficienti e le prestazioni del *sistema*$_G$.

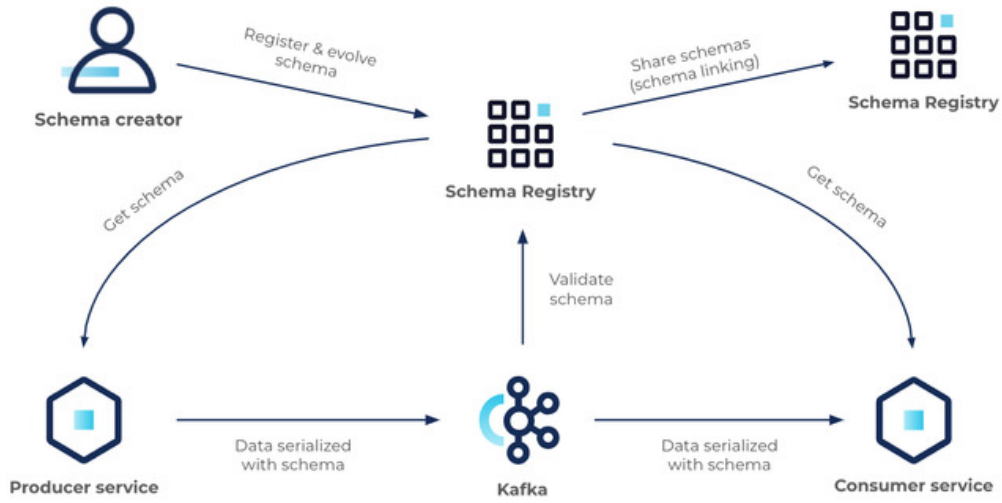

Figure 7: Schema Registry overview - Confluent documentation

Schema Registry permette la convalida del formato dei messaggi per garantire l'integrità e l'affidabilità dei dati in *Apache Kafka*$_G$.

1. **Convalida dei messaggi:** Schema Registry convalida i messaggi in base allo schema registrato per il topic. I messaggi non validi vengono scartati;
2. **Maggiore affidabilità:** La convalida dei messaggi aiuta a prevenire errori e a garantire che i dati siano conformi allo schema definito. Questo riduce il rischio di corruzione dei dati e di errori di elaborazione nei sistemi a valle;
3. **Interoperabilità:** Schema Registry facilita la comunicazione tra diversi sistemi che producono e consumano messaggi *Kafka*$_G$. Definendo un formato comune per i messaggi, i sistemi possono interoperare senza problemi, anche se sono sviluppati in linguaggi di programmazione diversi o utilizzano *librerie*$_G$ client differenti;
4. **Evolutività:** Schema Registry permette di evolvere gli schemi dei messaggi nel tempo in modo compatibile. Ciò significa che è possibile aggiungere nuovi campi o modificare la struttura dei messaggi senza interrompere i sistemi esistenti;
5. **Migliore debug:** La convalida dei messaggi fornisce informazioni utili in caso di errori, facilitando l'identificazione del problema e la sua risoluzione;
6. **Sicurezza:** La convalida dei messaggi può essere utilizzata per proteggere il *sistema*$_G$ da messaggi malformati o dannosi.

Convalida del formato dei messaggi con Schema Registry:

- **Definizione dello schema:** Vengono definiti e registrati gli schemi per i messaggi *Kafka*$_G$ utilizzando il formato JSON;
- **Produzione dei messaggi:** I producer *Kafka*$_G$ inviano messaggi conformi allo schema definito per il topic di destinazione;
- **Convalida dei messaggi:** Schema Registry convalida i messaggi in base allo schema registrato per il topic, i messaggi non validi vengono scartati;
- **Consumo dei messaggi:** I consumer *Kafka*$_G$ ricevono solo messaggi convalidati.

3.5.1 Schema dei messaggi

I messaggi nei topic dedicati allo streaming delle misurazioni dei sensori devono essere nel seguente formato per superare la convalida definita da contratto nello Schema Registry.

3.5.1.1 Misurazioni dei senori

```
{
    "type": "record",
    "name": "Misurazione",
    "fields": [
      {
        "name": "timestamp",
        "type": "string"
      },
      {
        "name": "value",
        "type": "float"
      },
      {
        "name": "type",
        "type": "string"
      },
      {
        "name": "latitude",
        "type": "float"
      },
      {
        "name": "longitude",
```

```
        "type": "float"
      },
      {
        "name": "ID_sensore",
        "type": "string"
      },
      {
        "name": "cella",
        "type": "string"
      }
    ]
  }
```

3.5.1.2 Misurazioni del punteggio di salute

I messaggi nel topic dedicato allo streaming delle misurazioni dei punteggi di salute devono essere nel seguente formato per superare la convalida definita da contratto nello Schema Registry.

```
{
    "type": "record",
    "name": "MisurazioneSalute",
    "fields": [
      {
        "name": "timestamp",
        "type": "string"
      },
      {
        "name": "value",
        "type": "float"
      },
      {
        "name": "type",
        "type": "string"
      },
      {
        "name": "cella",
        "type": "string"
      }
    ]
```

```
22      }
```

3.5.1.3 Implementazione Schema Registry

Per registrare uno schema nel registro degli schemi di *Kafka*$_G$ viene utilizzato il comando *curl* insieme alla *Rest API*$_G$ fornita dallo schema registry.
Il comando *curl* viene utilizzato tramite un'immagine *Docker*$_G$ dedicata.

3.6 Faust - Processing Layer

3.6.1 Introduzione

Per soddisfare il requisito opzionale del calcolo del punteggio di salute, si è scelto di utilizzare Faust, una libreria *Python*$_G$ ispirata al modello di *Kafka*$_{GG}$ Streams.
Faust facilita l'elaborazione di flussi di dati distribuiti in tempo reale, rendendola una libreria ideale per questo caso d'uso. Offre un'interfaccia di alto livello che astrae le complessità di *Kafka*$_{GG}$, rendendo la raccolta dati semplice e intuitiva. Inoltre Faust è progettato per essere scalabile e può essere utilizzato per gestire grandi volumi di dati.

3.6.1.1 Faust & Schema Registry

Faust deserializza i messaggi dei topic in conformità allo schema definito nello Schema Registry. Qualora si riscontrino messaggi non conformi, essi vengono eliminati. I messaggi validi vengono successivamente processati e instradati verso un topic *Kafka*$_G$ dedicato.
La gestione della connessione allo Schema Registry è integrata in Faust. Dopo aver fornito l'indirizzo dello Schema Registry nella configurazione dell'applicazione Faust, essa si occupa di validare i messaggi in base allo schema registrato per il relativo topic.

3.6.1.2 Calcolo del punteggio di salute

Il punteggio di salute rappresenta un indicatore sintetico del benessere generale di una città, viene calcolato sulla base di tre tipologie di misurazioni:

- Temperatura;
- Umidità;
- Livello di polveri sottili nell'aria (PM10).

Il calcolo del punteggio di salute avviene in due fasi:

1. **Incrementi:**
    - A intervalli regolari, per ciascuna tipoligia di misurazione, viene calcolato un incremento basato sulle sole misurazioni acquisite all'interno dell'intervallo;

- Ciascuna tipologia di misurazione ha un suo algoritmo di calcolo dell'incremento, basato su soglie predefinite di benessere.

2. **Punteggio Finale:**
   - Il punteggio di salute finale si ottiene sommando gli incrementi calcolati per le tre tipologie di misurazioni;
   - Punteggi più alti indicano un minore stato di benessere, con la necessità di interventi per migliorare la qualità della vita.

3.6.2 Componenti Faust & Processing Layer

- **Applicazione Faust:**

```
1   faust.App(<nome_app>, broker=<broker_kafka>)
```

  - Un'applicazione Faust è un *programma*$_G$ *Python*$_G$ che elabora flussi di dati in tempo reale da *Kafka*$_{GG}$;
  - **nome_app:** Identifica l'applicazione, coincide con il *ConsumerGroup* di *Kafka*$_G$;
  - **broker_kafka:** Indirizzo del *broker*$_G$ *Kafka*$_{GG}$ (hostname: porta);
  - Importante configurare l'applicazione con l'indirizzo dello Schema Registry per garantire la validazione e deserializzazione dei messaggi.

- **Topic:**

```
1   app.topic(<nome_topic>, value_type=<tipo_dato>)
```

  - **nome_topic:** Nome del topic *Kafka*$_{GG}$ a cui iscrivere l'app Faust;
  - **tipo_dato:** Classe che rappresenta il tipo di dato del topic (es. faust_measurement);
  - Nel caso si vogliano aggiungere altri topic da cui consumare dati basterà aggiungerli prima del parametro value_type separati da virgole.

- **Tipo di dato atteso:**

```
1   class faust_measurement(faust.Record, serializer='json')
```

  - È una classe che eredita da **faust.Record**;
  - **faust.Record** è una classe fornita dalla libreria Faust che semplifica la definizione di record per la rappresentazione dei dati in streaming;

  - Rappresenta una singola misurazione proveniente da un *sensore*$_G$. Viene usata nella applicazione Faust per definire il tipo di dati atteso nei topic *Kafka*$_{GG}$.

- **Modello per il calcolo del punteggio di salute:**
  - **Processore di misurazioni:** Tramite il *pattern*$_G$ *Object Adapter* e l'interfaccia *processor* l'app Faust invia le misurazioni ottenute dai topic al modello per il calcolo del punteggio di salute che verrà adattato come *processor*.

- **Agente di elaborazione:**

```
1   @app.agent(<topic>)
```

  - Permette di definire una funzione asincrona che elabora i dati dai topic;
  - La funzione riceve un iteratore di oggetti del tipo specificato per il topic;
  - La funzione esegue l'elaborazione desiderata su ogni misurazione.

- **Interfaccia *processor*:**
  - Viene utilizzata per definire un contratto di logiche di elaborazione;
  - Viene utilizzata dagli agenti di elaborazione per inviare le misurazioni al modello per il calcolo del punteggio di salute.

- **Task aggiuntivo** (opzionale):

```
1   @app.task()
```

  - Definisce una funzione eseguita una sola volta all'avvio dell'applicazione;
  - Nel nostro progetto, viene chiamato il metodo *start()* del thread adibito all'ottenimento e scrittura periodico dei punteggi di salute.

3.6.3 Processing layer data-flow

Di seguito esposto il flusso dei dati per il calcolo del punteggio di salute.

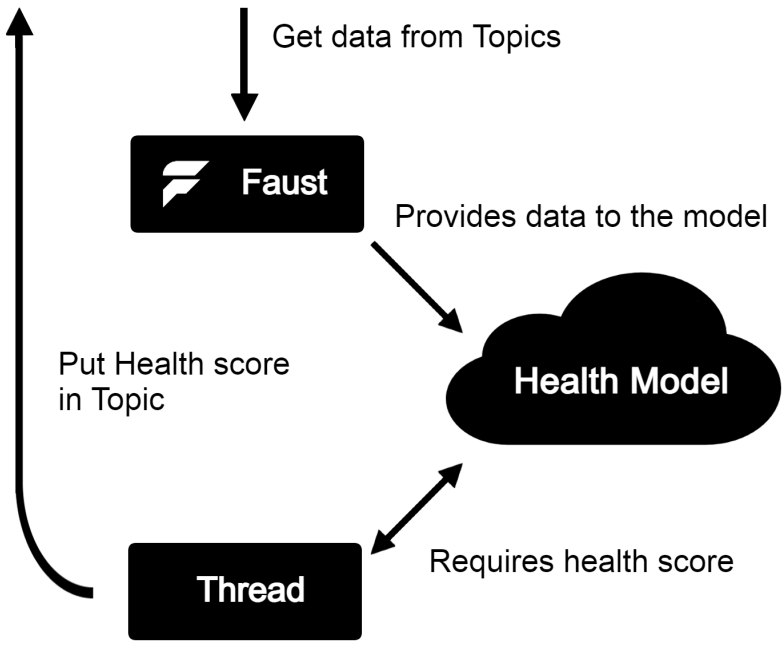

Figure 8: Faust data-flow

In breve:

1. Faust ottiene le misurazioni dai topic;
2. Tramite una porta di accesso le fornisce al modello per il calcolo del punteggio di salute;
3. Quando l'applicazione Faust viene avviata questa avvia a sua volta un Thread che periodicamente ottiene i punteggi di salute calcolati sulla base delle misurazioni ottenute da Faust e tramite il modulo Writer li invia al topic $Kafka_G$ dedicato.

3.6.4 Modello per il calcolo del punteggio di salute

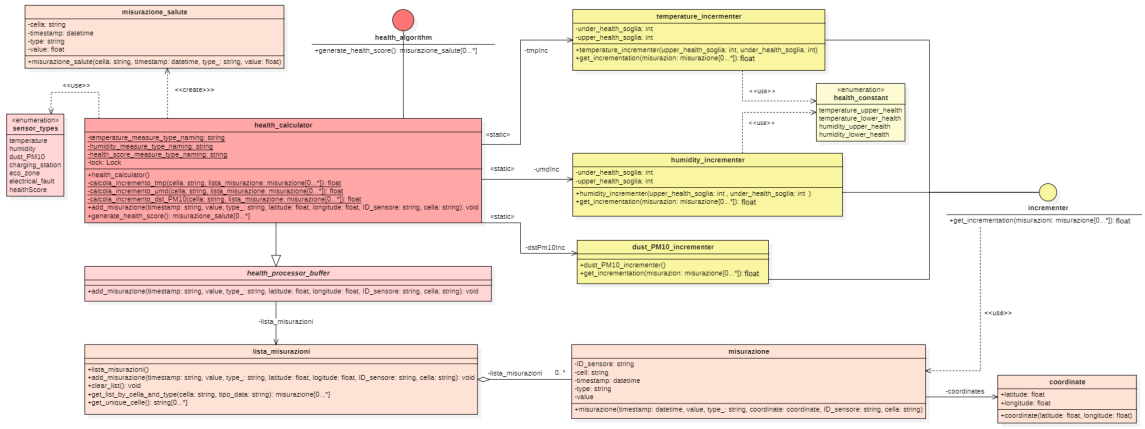

Figure 9: Modello per il calcolo del punteggio di salute - InnovaCity

Il modello per il calcolo del punteggio di salute riceve le letture dei sensori attraverso gli agenti di elaborazione dell'applicazione Faust, i quali sono in ascolto sui topic *Kafka*$_{GG}$ relativi alle misurazioni di temperatura, umidità e polveri sottili PM10.
Ad intervalli regolari, il *sistema*$_{G}$ calcola il punteggio di salute della città basandosi su tali misurazioni. Una volta effettuato il calcolo, il risultato è reso disponibile in un topic *Kafka*$_{GG}$ dedicato. Il modello che racchiude la logica per il calcolo, richiamato a intervalli regolari, è quello attualmente preso in esame.
In sintesi, il modello:

- Riceve le misurazioni di temperatura, umidità e polveri sottili dall'agente dell'app Faust;
- Riceve da un thread la richiesta ad intervalli regolari di calcolare i punteggi di salute per le celle della città con le misurazioni ottenute in tempo reale.

In modo simile a quanto accade in un'*architettura*$_{G}$ esagonale, la logica del modello è completamente disaccoppiata dai suoi utilizzatori, i quali interagiscono con esso tramite specifiche classi adapter. Questo approccio promuove la separazione delle preoccupazioni e favorisce la modularità del *sistema*$_{G}$. Gli adapter fungono da ponte tra il modello e gli utilizzatori, consentendo una comunicazione fluida e senza dipendenze dirette. Così, eventuali cambiamenti nella logica del modello possono essere implementati senza influenzare gli utilizzatori, garantendo una maggiore flessibilità e manutenibilità del *sistema*$_{G}$ nel suo complesso.
Il presente modulo è concepito per fornire la logica relativa al puro calcolo del punteggio di salute della città. Tale calcolo si basa su un modello che tiene conto delle misurazioni sopra

citate ed è stato progettato al fine di determinare un punteggio di salute per ciascuna cella della città in cui sono presenti misurazioni delle suddette tipologie.

3.6.4.1 Design pattern Strategy

Il modello per il calcolo del punteggio di salute è stato ideato mediante l'utilizzo del design *pattern*$_G$ Strategy. Tale *pattern*$_G$ consente di definire una famiglia di algoritmi, di incapsularli e renderli intercambiabili. Ciò permette di variare l'algoritmo impiegato per il calcolo del punteggio di salute senza incidere sui suoi utilizzatori. In particolare, l'interfaccia *health_algorithm* stabilisce il contratto che deve essere rispettato da tutti gli algoritmi per il calcolo del punteggio di salute.
Inoltre, un'implementazione del *pattern*$_G$ *Strategy* è presente anche negli "Incrementers". Questi, a partire dalle misurazioni fornite, restituiscono un valore da sommare al punteggio di salute della città. Tale incremento è determinato in base a delle soglie predefinite di temperatura, umidità e polveri sottili PM10, le quali sono definite di default in *health_constants* ma possono essere impostate al momento della costruzione.
In particolare, l'interfaccia *incrementer* specifica il contratto che deve essere rispettato da tutti gli Incrementers. Vengono implementati tre Incrementers, uno per la temperatura, uno per l'umidità e uno per le polveri sottili PM10, come strategie del *pattern*$_G$ Strategy.

3.6.4.2 Classi, interfacce, metodi e attributi

- **Interfaccia: *health_algorithm***
  - **Metodi:**
    - **generate_new_health_score(): List[misurazione_salute] [abstractmethod]** - Un metodo astratto che, nelle sue implementezioni concrete genera un punteggio di salute.
  - **Note:**
    - L'interfaccia definisce il contratto per un algoritmo di calcolo per un punteggio di salute. Le sottoclassi devono implementare il metodo *generate_new_health_score()*;
    - Rappresenta la componente "Strategy" del *pattern*$_G$ omonimo;
    - Per rispettare il Single Responsibility Principle *(SRP)*, *health_algorithm* è stata divisa dalla logica di buffering delle misurazioni presente nella classe astratta *health_processor_buffer* poiché l'utilizzatore *health_calculator_thread* non utilizza i metodi per il buffering.
- **Classe astratta: *health_processor_buffer***
  - **Attributi:**

        - **lista_misurazioni: lista_misurazioni [private]** - Una lista di oggetti *misurazione*;
        - **lock: threading.Lock [private]** - Un oggetto lock per gestire l'accesso concorrente alla lista di misurazioni.
    - **Metodi:**
        - **add_misurazione(timestamp, value, type_, latitude, longitude, ID_sensore, cella): None [public]** - Aggiunge una nuova misurazione alla lista di misurazioni;
        - **clear_list(): None [public]** - Svuota la lista di misurazioni.
    - **Note:**
        - La classe astratta definisce un buffer di misurazioni per effettuare processing su di esse;
        - La logica di buffering e quella dell'algoritmo per il calcolo del punteggio di salute vengono separate in due astrazioni per rispettare il Single Responsibility Principle *(SRP)*. Gli utilizzatori di questa classe, ovvero i Processor, sono interessati esclusivamente al metodo per l'invio del dato al buffer;
        - La classe astratta definisce un'interfaccia per la comunicazione con gli utilizzatori esterni al modello.

- **Classe: *health_calculator***
    - **Attributi:**
        - **tmpInc: temperature_incrementer [private]** - Utilizzato per il calcolo dell'incremento di temperatura;
        - **umdInc: humidity_incrementer [private]**; - Utilizzato per il calcolo dell'incremento di umidità;
        - **dstPm10Inc: dust_PM10_incrementer [private]** - Utilizzato per il calcolo dell'incremento di PM10;
        - **temperature_measure_type_naming: string [private]** - Nomenclatura dei tipi di misurazione di temperatura;
        - **humidity_measure_type_naming: string [private]** - Nomenclatura dei tipi di misurazione di umidità;
        - **dtsPm10_measure_type_naming: string [private]** - Nomenclatura dei tipi di misurazione di PM10;
        - **health_score_measure_type_naming: string [private]** - Nomenclatura dei tipi di misurazione di punteggio di salute;
        - **lock [private]** - Un oggetto lock per gestire l'accesso concorrente.
    - **Metodi:**

        - **generate_new_health_score(): List[misurazione_salute] [public]** - Genera e restituisce una nuova lista di punteggi di salute, uno per ogni cella della città di cui sono state fornite misurazioni;
        - **calcola_incremento_tmp(cella: str, lista_misurazioni): int [private]** - Calcola e restituisce l'incremento della temperatura;
        - **calcola_incremento_umd(cella: str, lista_misurazioni): int [private]** - Calcola e restituisce l'incremento dell'umidità;
        - **calcola_incremento_dstPm10(cella: str, lista_misurazioni): int [private]** - Calcola e restituisce l'incremento della polvere PM10.
    - **Note:**
        - La classe implementa l'interfaccia *health_algorithm* e la classe astratta *health_processor_buffer* per calcolare il punteggio di salute tramite la strategia concreta definita in *generate_new_health_score()* che genera una nuova lista di punteggi di salute;
        - Questa classe rappresenta il cervello del calcolo del punteggio di salute in quanto utilizzatore di tutti gli incrementatori e delle misurazioni bufferizzate per creare una strategia di calcolo.

- **Classe: *misurazione***
    - **Attributi:**
        - **timestamp: datetime [private]** - Timestamp della misurazione;
        - **value [private]** - Valore della misurazione;
        - **type: str [private]** - Tipo della misurazione;
        - **coordinates: coordinate [private]** - Coordinate della misurazione;
        - **ID_sensore: str [private]** - ID_G del $sensore_G$ che ha effettuato la misurazione;
        - **cella: str [private]** - Cella in cui è stata effettuata la misurazione.
    - **Metodi:**
        - **__eq__(other: misurazione): bool [public]** - Ridefinizione dell'operatore di uguaglianza per confrontare due oggetti *misurazione*.

- **Classe: *coordinate***
    - **Attributi:**
        - **latitude: float [private]** - Latitudine della coordinata;
        - **longitude: float [private]** - Longitudine della coordinata.
    - **Metodi:**

        - **__eq__(other: coordinate): bool [public]** - Ridefinizione dell'operatore di uguaglianza per confrontare due oggetti coordinate.

- **Classe: *misurazione_salute***
  - **Attributi:**
    - **timestamp: datetime [private]** - Il timestamp della misurazione di salute;
    - **value: float [private]** - Il valore della misurazione di salute;
    - **type: string [private]** - Il tipo della misurazione;
    - **cella: string [private]** - La cella della misurazione di salute.
  - **Note:**
    - La classe rappresenta una misurazione di salute. Contiene informazioni sul timestamp, il valore (ovvero il punteggio di salute calcolato), il tipo della misurazione e la cella relativa alla misurazione.

- **Classe: *lista_misurazioni***
  - **Attributi:**
    - **list:List[misurazione] [private]** - Una lista di oggetti *misurazione*.
  - **Metodi:**
    - **add_misurazione(timestamp, value, type_, latitude, longitude, ID_sensore, cella): None [public]** - Aggiunge una nuova misurazione alla lista;
    - **clear_list(): None [public]** - Svuota la lista di misurazioni;
    - **get_list_by_cella_and_type(cella: *str*, tipo_dato: *str*): List[misurazione] [public]** - Restituisce una lista di misurazioni che corrispondono alla cella e al tipo di misurazione specificati (temperatura, umidità, ecc..);
    - **get_unique_celle(): List[str] [public]** - Restituisce la lista di celle presenti nelle misurazioni senza ripetzioni.
  - **Note:**
    - La classe rappresenta una lista di misurazioni. Fornisce metodi per aggiungere misurazioni, svuotare la lista, ottenere misurazioni filtrate per cella e tipo, e ottenere le celle di cui si hanno misurazioni.

- **Enumerazione: *sensor_types***
  - **Costanti:**
    - **TEMPERATURE: str [public]** - Rappresenta la nomenclatura dei $sensore_G$ di temperatura;

        - **HUMIDITY: str [public]** - Rappresenta la nomenclatura dei *sensore*$_G$ di umidità;
        - **DUST_PM10: str [public]** - Rappresenta la nomenclatura dei *sensore*$_G$ di quantità di particelle PM10;
        - **CHARGING_STATION: str [public]** - Rappresenta la nomenclatura dei *sensore*$_G$ di stato delle colonnine di ricarica;
        - **ECOLOGICAL_ISLAND: str [public]** - Rappresenta la nomenclatura dei *sensore*$_G$ di stato riempimento isole ecologica;
        - **WATER_PRESENCE: str [public]** - Rappresenta la nomenclatura dei *sensore*$_G$ di presenza d'acqua;
        - **ELECTRICAL_FAULT: str [public]** - Rappresenta la nomenclatura dei *sensore*$_G$ di guasti elettrici.
    - **Note:**
        - L'enumerazione viene utilizzata per centralizzare la gestione della nomenclatura dei tipi di sensori che verrà salvata nelle misurazioni.

- **Interfaccia: *incrementer***
    - **Metodi:**
        - **get_incrementation(misurazioni: List[misurazione]): int [abstractmethod]** - Questo metodo astratto, nelle sue implementazioni sulle sottoclassi, calcola e restituire un incremento basato sulla lista di misurazioni fornita.
    - **Note:**
        - L'interfaccia definisce il contratto per un incrementatore. Le sottoclassi devono implementare il metodo *get_incrementation()*;
        - Rappresenta la componente "Strategy" del *pattern*$_G$ omonimo.

- **Classe: *temperature_incrementer***
    - **Attributi:**
        - **upper_health_soglia: int [private]** - La soglia superiore di benessere per la temperatura;
        - **under_health_soglia: int [private]** - La soglia inferiore di benessere per la temperatura.
    - **Metodi:**
        - **get_incrementation(misurazioni: List[misurazione]): float [public]** - Calcola e restituisce un incremento basato sulle sole misurazioni di temperatura della lista fornita.

  - **Note:**
    - La classe implementa l'interfaccia *incrementer*;
    - I valori di default per le soglie vengono presi dall'enumerazione *health_constant* altrimenti sono impostabili alla costruzione;
    - Rappresenta una strategia concreta del *pattern$_G$ Strategy* per il calcolo dell'incremento di temperatura.

- **Classe: *humidity_incrementer***
  - **Attributi:**
    - **upper_health_soglia: int [private]** - La soglia superiore di benessere per l'umidità;
    - **under_health_soglia: int [private]** - La soglia inferiore di benessere per l'umidità.
  - **Metodi:**
    - **get_incrementation(misurazioni: List[misurazione]): float [public]** - Calcola e restituisce un incremento basato sulle sole misurazioni di umidità della lista fornita.
  - **Note:**
    - La classe implementa l'interfaccia *incrementer*;
    - I valori di default per le soglie vengono presi dall'enumerazione *health_constant* altrimenti sono impostabili alla costruzione;
    - Rappresenta una strategia concreta del *pattern$_G$ Strategy* per il calcolo dell'incremento di umidità.

- **Classe: *dust_PM10_incrementer***
  - **Metodi:**
    - **get_incrementation(misurazioni: List[misurazione]): float [public]** - Calcola e restituisce un incremento basato sulle sole misurazioni di polveri sottili della lista fornita.
  - **Note:**
    - La classe implementa l'interfaccia *incrementer*;
    - Rappresenta una strategia concreta del *pattern$_G$ Strategy* per il calcolo dell'incremento di polveri sottili PM10;
    - A differenza degli altri *Incrementer*, *dust_PM10_incrementer* non definisce soglie di benessere in quanto è scontato che il valore ottimale di inquinamento è zero.

3.6.5 Modulo Writer

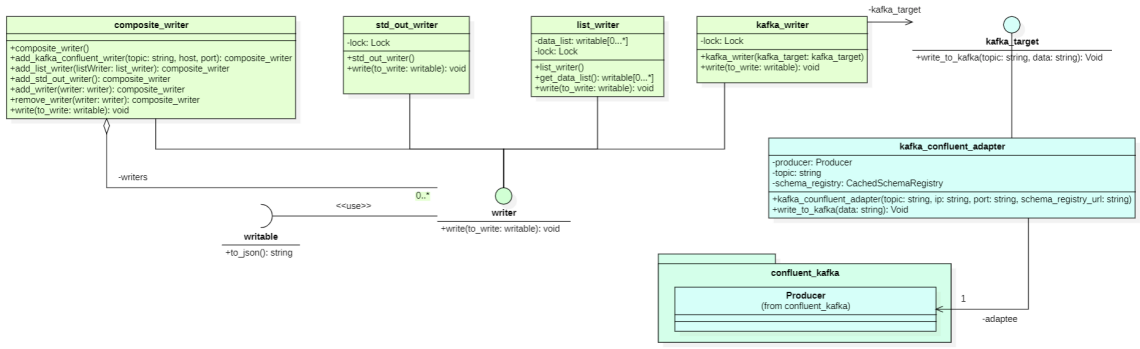

Figure 10: Modulo Writer - InnovaCity

Il modulo Writer è il medesimo descritto in 3.3.2 e viene nella sua totalità riutilizzato per la scrittura dei punteggi di salute calcolati. Non viene riportata la strategia di scrittura su di una lista poichè non ne è stato ritenuto necessario l'utilizzo.

Classi, interfacce metodi e attributi

Tutte le informazioni sono già state esposte in: 3.3.2.

3.6.6 Modulo Threading/Scheduling

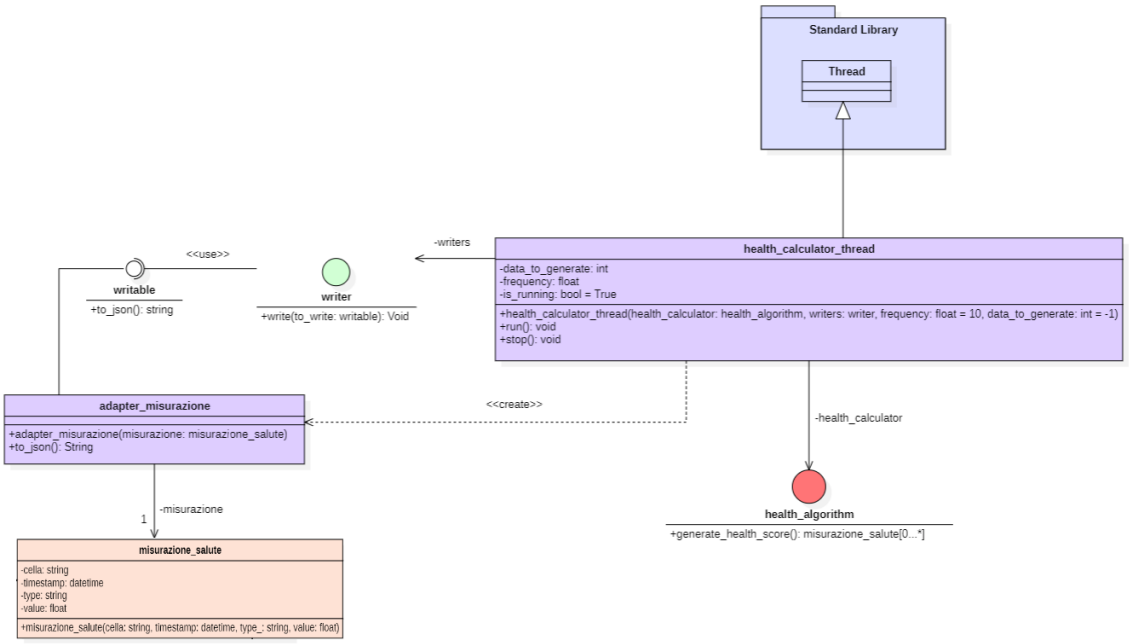

Figure 11: Modulo Threading/Scheduling health Model - InnovaCity

Questo modulo si occupa di integrare la logica di Scheduling e Threading per il calcolo periodico del punteggio di salute della città e quella di scrittura/invio di *writable*. In particolare, fornisce un'implementazione di un thread che, a intervalli regolari, richiama il calcolo del punteggio di salute della città. Successivamente, utilizzando il modulo *Writer*, adatta le misurazioni di salute ottenute all'interfaccia *writable* per inviare i dati al topic $Kafka_G$ e/o stamparli su terminale. Pertanto, questo modulo è utilizzatore del modello per il calcolo del punteggio di salute e del modulo di *Writer*. Il modulo è stato progettato per rispettare il Dependecy Inversion Principle *(DIP)*, di conseguenza sia i moduli di alto livello che quelli di basso livello dipendono da astrazioni (interfacce o classi astratte).
In sintesi, il modulo:

1. Richiama l'algoritmo per calcolo del punteggio di salute della città a intervalli regolari;
2. Scrive il risultato ottenuto sul topic $Kafka_{GG}$ dedicato.

Classi, interfacce, metodi e attributi

- **Classe: *health_calculator_thread***
  - **Attributi:**

        - **health_calculator: health_algorithm [private]** - Un implementatazione dell'interfaccia *health_algorithm*, ovvero una strategia per il calcolo del punteggio di salute;
        - **frequency: float [private]** - La frequenza con cui il thread genera nuovi punteggi di salute;
        - **is_running: bool [private]** - Un flag che indica se il thread è in esecuzione;
        - **data_to_generate: int [private]** - Il numero di misurazioni di salute da generare;
        - **writers: writer [private]** - Un oggetto che implementa la classe *writer*. (Singolo scrittore o albero, Composite *pattern*$_G$)
    - **Metodi:**
        - **run(): None [public]** - Esegue il thread, generando nuovi punteggi di salute ad una certa frequenza;
        - **stop(): None [public]** - Ferma l'esecuzione del thread.
    - **Note:**
        - La classe estende la classe *threading.Thread*;
        - Se *data_to_generate* è minore di zero, genera misurazioni di salute finchè il thread non viene interroto dall'esterno;
        - Grazie al *pattern*$_G$ *Strategy* è possibile cambiare agevolmente l'algoritmo volto al calcolo del punteggio di salute della città.

- **Classe: *adapter_misurazione***
    - **Metodi:**
        - **to_json(): String [public]** - Ritorna una stringa JSON compatibile con lo schema richiesto nello Schema Registry per le misurazione di salute.
    - **Note:**
        - Rappresenta il componente adapter del *pattern*$_G$ *Object adapter*;
        - L'adapter adatta le misurazioni di salute ottenute dall'algoritmo di calcolo del punteggio di salute all'interfaccia *writable* per l'invio al topic *Kafka*$_{GG}$.

3.6.7 Modulo Processing

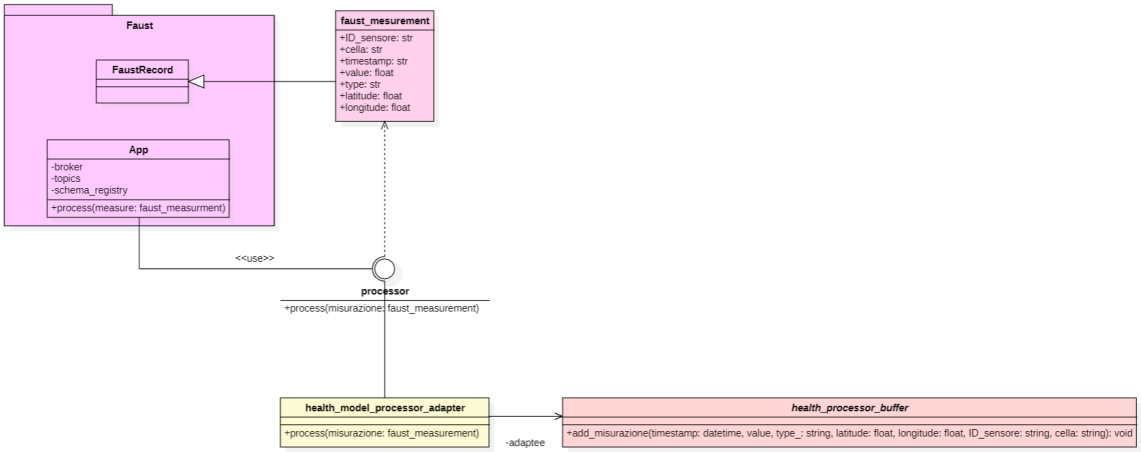

Figure 12: Modulo Processing - InnovaCity

Al fine garantire un'interfaccia uniforme delle operazioni di elaborazione dei dati provenienti da *Kafka*$_{GG}$ tramite Faust e per stabilire un canale di comunicazione con il modello per il calcolo del punteggio di salute, viene sviluppato il modulo di Processing. Il modulo offre l'interfaccia target denominata *processor*, e un suo adapter denominato *health_model_processor_adapter* per l'invio delle misurazioni al modello per il calcolo del punteggio di salute.

Design Pattern Object Adapter

Nel contesto dell'applicazione Faust, all'interno del ruolo svolto dagli agenti, ogni volta che una misurazione viene ricevuta, viene invocato il metodo *process()* dell'implementazione dell'interfaccia *processor* denominata *health_model_processor_adapter*.
In particolare, *health_model_processor_adapter* adatta la classe astratta *health_processor_buffer*, che rappresenta un buffer di misurazioni utilizzato per eseguire il calcolo periodico del punteggio di salute della città, all'interfaccia *processor*.
Questo *pattern*$_{G}$ consente di definire dei contratti per le logiche di elaborazione e di rendere il modello indipendente dall'implementazione specifica dell'applicazione Faust. Allo stesso tempo, facilita la sostituzione dell'operazione di elaborazione eseguita su ogni misurazione dagli agenti grazie al contratto definito nell'interfaccia *processor*.

Classi, interfacce, metodi e attributi

- **Interfaccia: *processor***
  - **Metodi:**

        - **process(misurazione: faust_measurement): None [public, abstract]** - Metodo astratto che deve essere implementato nelle sottoclassi per effettuare elaborazioni di una misurazione.
    - **Note:**
        - Le sottoclassi devono implementare il metodo astratto *process()* definendo la propria operazione da effettuare su ogni misurazione ricevuta dai topic di iscrizione;
        - Rappresenta la componente "Target" del *pattern$_G$ Object Adapter*;
        - L'interfaccia è stata progettata per garantire e rappresentare un contratto uniforme per i metodi di elaborazione dei dati provenienti da *Kafka$_{GG}$* tramite Faust;
        - Gli agenti in ascolto sul topic utilizzeranno un implementatazione di *processor* per effettuare l'elaborazione delle misurazioni ottenute.

- **Classe: *faust_measurement***
    - **Attributi:**
        - **timestamp: str** - Il timestamp della misurazione;
        - **value: float** - Il valore della misurazione;
        - **type: str** - Il tipo della misurazione;
        - **latitude: float** - La latitudine della misurazione;
        - **longitude: float** - La longitudine della misurazione;
        - **ID_sensore: str** - L'*ID$_G$* del *sensore$_G$* che ha effettuato la misurazione;
        - **cella: str** - La cella in cui è stata effettuata la misurazione.
    - **Note:**
        - La classe *faust_measurement* definita eridatando da *faust.Record* rappresenta un singolo record di misurazione proveniente da un *sensore$_G$* consumata da un'applicazione Faust;
        - Faust si occupa automaticamente della conversione dei dati in formato JSON sulla base degli attributi definiti, facilitando la ricezione e la deserializzazione dei dati nei topic *Kafka$_{GG}$*;
        - **In sintesi:** Questa classe viene utilizzata in un'applicazione Faust per definire il tipo dei dati attesi nei topic *Kafka$_{GG}$* di interesse. I dati provenienti dai sensori, contenenti timestamp, valore, tipo, coordinate geografiche, identificativo del *sensore$_G$* e eventuale cella di appartenenza, verranno convertiti in oggetti di tipo *faust_measurement* prima di essere elaborati dall'applicazione.

- **Classe: *health_model_processor_adapter***

- **Attributi:**
  - **health_calculator: health_processor_buffer** - Un implementazione della classe astratta *health_processor_buffer*.
- **Metodi:**
  - **process(misurazione: faust_measurement): None [public, async]** - Aggiunge la misurazione ad un implementazione della classe astratta *health_processor_buffer* adattando un oggetto di tipo *faust_measurement* alla porta di accesso fornita da *health_processor_buffer* per l'elaborazione volta al calcolo del punteggio di salute.
- **Note:**
  - La classe implementa l'interfaccia *processor* ed implementa il metodo astratto *process()* per aggiungere/adattare la misurazione del tipo *faust_measurement* ad un implementatazione di *health_processor_buffer*;
  - Rappresenta la componente "Adapter" del *pattern*$_G$ *Object Adapter*;
  - Il fine è adattare la classe astratta *health_processor_buffer* o più in generale il modello per il calcolo del punteggio di salute, all'interfaccia *processor* per l'elaborazione delle misurazioni provenienti dai topic *Kafka*$_{GG}$ e consumate dagli agenti Faust.

3.7 Configurazione Database

Si è optato per l'utilizzo di *ClickHouse*$_G$ per il salvataggio dei dati, le motivazioni sono descritte nella sezione 2.3.4. In particolare, per ogni *sensore*$_G$ di cui si desidera memorizzare le misurazioni, viene creata una tabella che le acquisisce dal relativo topic *Kafka*$_G$ e una tabella della famiglia MergeTree che permette la loro persistenza. Le tipologie di sensori le quali misurazioni si vogliono trattare nel progetto sono:

- Sensori di temperatura;
- Sensori di umidità;
- Sensori di rilevamento polveri sottili;
- Sensori di stato di riempimento delle isole ecologiche;
- Sensori di occupazione delle colonnine di ricarica;
- Sensori di guasti elettrici;
- Sensori di presenza dell'acqua.

Oltre alla misurazioni dei sensori, si è deciso di memorizzare in una tabella apposita anche i punteggi di salute calcolati per ogni cella sulla base delle relative misurazioni.
La progettazione del *database*$_G$ *ClickHouse*$_G$ è cruciale, poiché un'adeguata ottimizzazione consente di garantire prestazioni ottimali per un *sistema*$_G$ orientato al tempo reale e che richiede analisi su grandi volumi di dati.

3.7.1 Funzionalità Clickhouse utilizzate

3.7.1.1 Materialized Views

Documentazione:
https://clickhouse.com/docs/en/guides/developer/cascading-materialized-views
(Consultato: 25/03/2024).
Le Materialized Views in *ClickHouse*$_G$ sono un meccanismo volto a migliorare le prestazioni delle *query*$_G$ e semplificare l'accesso ai dati.
Funzionano mantenendo una copia fisica dei risultati di una *query*$_G$ di selezione, che viene quindi memorizzata su disco. Questa copia è aggiornata periodicamente in base ai dati sottostanti.

3.7.1.2 Utilizzi principali delle Materialized Views

- **Calcolo aggregazioni e popolamento tabelle:** Le materialized Views vengono utilizzate per calcolare aggregazioni su dati e quindi popolare altre tabelle con i risultati aggregati;
- **Ottimizzazione delle prestazioni:** Memorizzando i risultati di una *query*$_G$ complessa, le Materialized Views consentono di eseguire rapidamente le *query*$_G$ successive senza dover ricalcolare i dati ogni volta. Ciò è particolarmente utile in applicazioni che richiedono interrogazioni frequenti su grandi volumi di dati;
- **Decomposizione delle query complesse:** Le Materialized Views consentono di decomporre *query*$_G$ complesse in passaggi più semplici e riutilizzabili, migliorando la leggibilità del codice e semplificando lo sviluppo e la manutenzione delle *query*$_G$.

Nel progetto le materialized view sono fondamentali per spostare automaticamente i dati dai topic *Kafka*$_G$ alle tabelle di destinazione.

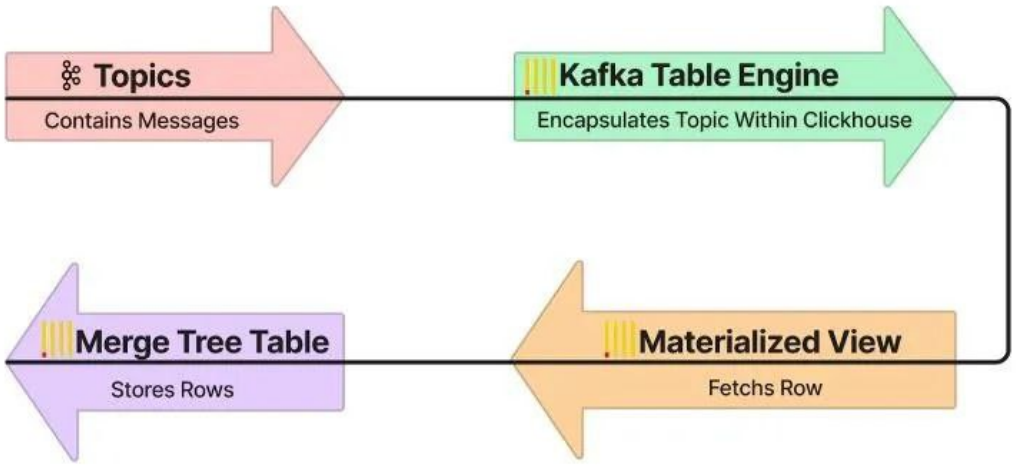

Figure 13: Clickhouse data pipeline

3.7.1.3 MergeTree

Documentazione:
https://clickhouse.com/docs/en/engines/table-engines/mergetree-family/mergetree#mergetree (Consultato: 25/03/2024).
MergeTree è uno dei motori di archiviazione di base più utilizzati in *ClickHouse*$_G$. È progettato per gestire dati ordinati in *append-only* e offre un'ottima combinazione di prestazioni di lettura, scrittura, scalabilità e funzionalità per diversi casi d'uso.
Caratteristiche principali:

- **Ordinato:** I dati sono archiviati in ordine crescente in base alla chiave di ordinamento specificata;
- **Append-only:** I nuovi dati vengono sempre aggiunti alla fine della tabella;
- **Scalabile:** Altamente scalabile orizzontalmente;
- **Partizionamento:** Supporto per partizionare la tabella in base a una colonna specifica;
- **TTL (Time to Live):** Consente di definire un TTL per i dati, eliminandoli automaticamente dopo un periodo di tempo specificato;
- **Compressione:** Supporta diverse tecniche di compressione per ridurre lo spazio di archiviazione utilizzato.

Casi d'uso comuni:

- Analisi di *log*$_G$;
- Dati finanziari;
- Dati di monitoraggio;
- Time series data.

Vantaggi:

- Prestazioni di lettura e scrittura elevate;
- Scalabilità orizzontale;
- Funzionalità avanzate;
- Adatto a diversi casi d'uso.

3.7.1.4 Time to Live in ClickHouse

Documentazione:
https://clickhouse.com/docs/en/guides/developer/ttl#implementing-a-rollup
(Consultato: 25/03/2024).
TTL *(Time To Live)* si riferisce alla capacità di spostare, eliminare o eseguire il rollup di righe o colonne dopo che è trascorso un determinato intervallo di tempo. Sebbene l'espressione "time to live" sembri applicarsi solo all'eliminazione di vecchi dati, TTL ha diversi casi d'uso:

- **Eliminazione dei vecchi dati:** Rimuovere i dati dopo un certo periodo di tempo;
- **Spostamento dei dati tra dischi:** Spostare i dati tra volumi di archiviazione dopo un certo periodo di tempo;
- **Rollup dei dati:** Effettuare operazioni di aggregazione sui dati dopo un intervallo di tempo prestabilito.

Nel progetto è stato fatto utilizzo dove opportuno dell'operazione di Rollup di misurazioni dei sensori. Nello specifico dopo un certo intervallo di tempo non vengono piu conservate tutte le misurazioni di ogni *sensore*$_G$, ma solo la media delle loro misurazioni nel periodo di tempo specificato. Questo permette di ridurre il volume di dati da analizzare e di mantenere comunque un'informazione utile per l'analisi.

3.7.1.5 Partition

Documentazione:
https://clickhouse.com/docs/en/engines/table-engines/mergetree-family/mergetree#partition-by (Consultato: 25/03/2024).
Le partizioni sono una funzionalità fondamentale di *ClickHouse*$_G$ che consente di organizzare in modo efficiente la gestione di grandi volumi di dati. Questa funzionalità permette di suddividere i dati in gruppi logici in base a criteri specifici, come il valore di una colonna o un intervallo di tempo.
Grazie a questa organizzazione ottimizzata, le *query*$_G$ che richiedono l'accesso a dati specifici all'interno di una partizione possono essere eseguite rapidamente, garantendo prestazioni elevate anche su dataset di grandi dimensioni.
Le partizioni rappresentano uno strumento particolarmente utile quando si lavora con dati di serie temporali. La decisione sull'utilizzo del partizionamento dovrebbe derivare da una serie di considerazioni chiave:

- **Interrogazione singola:** È previsto che le *query*$_G$ siano principalmente focalizzate su una singola partizione? Ad esempio, se le interrogazioni tendono a riguardare risultati entro un periodo specifico come un giorno o un mese, sarebbe vantaggioso partizionare i dati in base a tali periodi temporali;
- **Scadenza dei dati (TTL):** Se si desidera applicare una politica di TTL ai dati, in modo che una volta che una partizione raggiunge una certa età, venga applicata un'azione specifica su di essa, allora può risultare opportuno l'utilizzo del partitioning in base a tali periodi temporali.

Spesso si consiglia di mantenere il numero di partizioni al di sotto di circa 100. Anche se è tecnicamente possibile utilizzare fino a 1000 partizioni, tale approccio potrebbe non essere ottimale e potrebbe influenzare le prestazioni del *sistema*$_G$, inclusi tempi di avvio, tempi di inserimento, *query*$_G$-time e l'utilizzo di memoria.
Questo è dovuto all'impatto che un elevato numero di partizioni può avere sul file system e sulle dimensioni dell'indice, con conseguente aumento della complessità gestionale e dei carichi di lavoro.
Nel nostro progetto viene fatto utilizzo del partizionamento temporale nelle tabelle in cui è ritenuto oppurtuno sulla base delle considerazioni sopra descritte. L'utilizzo viene giustificato infatti dall'impiego di TTL, oltre al fatto che le interrogazioni riguardino principalmente una singola partizione.

3.7.1.6 Projection

Documentazione:

https://clickhouse.com/docs/en/sql-reference/statements/alter/projection (Consultato: 25/03/2024).
Approfondimento:
https://presentations.clickhouse.com/percona2021/projections.pdf (Consultato: 25/03/2024).
Le projection in *ClickHouse*$_G$ sono una funzionalità di ottimizzazione per migliorare le prestazioni delle *query*$_G$ su tabelle MergeTree. Esse creano delle viste pre-aggregate o pre-ordinate dei dati originali, consentendo a *ClickHouse*$_G$ di accedere e analizzare i dati in modo più efficiente.
Questa funzionalità è utile per:

- Eseguire *query*$_G$ basate su di una colonna che non fa parte della chiave primaria;
- Pre-aggregare colonne, riducendo sia i calcoli che l'I/O.

Come funzionano le projection:

- **Definizione:** Si definisce una projection su una tabella MergeTree esistente. La projection specifica una sottocategoria di colonne e, opzionalmente, un nuovo ordine di ordinamento per queste colonne;
- **Creazione interna:** *ClickHouse*$_G$ crea internamente una nuova tabella nascosta che contiene i dati della projection;
- **Aggregazione o ordinamento:** I dati selezionati vengono aggregati (utilizzando funzioni come media, somma, ecc.) o ordinati in base alle colonne specificate nella definizione della projection;
- **Selezione automatica:** Durante l'esecuzione di una *query*$_G$, *ClickHouse*$_G$ analizza la projection e la tabella originale. Se la projection rientra nei criteri di selezione della *query*$_G$ (colonne e ordine), *ClickHouse*$_G$ può utilizzare la projection per rispondere alla *query*$_G$, evitando di accedere alla tabella originale completa.

Vantaggi delle projection:

- **Miglioramento delle prestazioni:** Accedendo ai dati pre-aggregati o pre-ordinati, le *query*$_G$ possono essere eseguite più velocemente;
- **Riduzione dell'utilizzo della CPU:** Le aggregazioni e gli ordinamenti vengono eseguiti durante la creazione della projection, riducendo il carico di lavoro della CPU durante le *query*$_G$;
- **Più flessibilità di query:** Le projection possono supportare ordini diversi rispetto alla tabella originale, aumentando la flessibilità delle *query*$_G$.

Nel contesto del progetto, le projection sono state impiegate strategicamente al fine di ottimizzare le prestazioni delle interrogazioni basate sulla colonna "*cella*". Questa colonna, pur non facendo parte della chiave primaria, è oggetto di interrogazioni molto frequenti su vasti volumi di dati.
L'utilizzo di tale strategia si rivela particolarmente vantaggiosa vista la previsione di interrogazioni ricorrenti e su dataset di grandi dimensioni basate su tale colonna, contribuendo così a garantire tempi di risposta ottimizzati e una migliore esperienza utente.
In generale l'introduzione delle projections produce risultati di notevole importanza, come illustrato di seguito.
Consideriamo una *query*$_G$, che calcola la media globale di **170'000** misurazioni di temperatura tramite *Grafana*$_G$:

```
1   SELECT avgMerge(value) count(*)
2   FROM innovacity.temperatures
3   WHERE cella = 'Arcella'
4
5   --Query id: 59811218-6f57-4753-b24a-a81c2a8380df
```

Listing 1: Esempio query - Grafana

Senza l'utilizzo delle projections, il risultato ottenuto è il seguente:

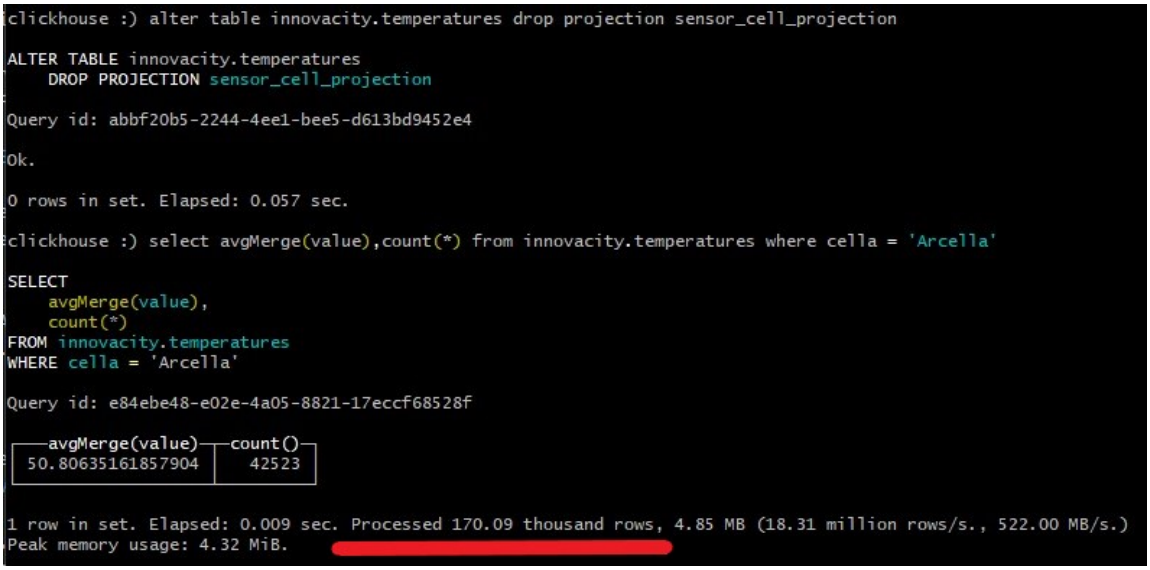

Figure 14: Query esempio senza projection - ClickHouse

Il totale di righe processate per ottenere il risultato è **170'090**, ovvero la totalità delle righe

presenti nella tabella, con **9**ms di tempo utilizzati e un picco di memoria RAM impiegata di **4.32**MiB.
Con l'impiego delle projections invece il risultato ottenuto è il seguente:

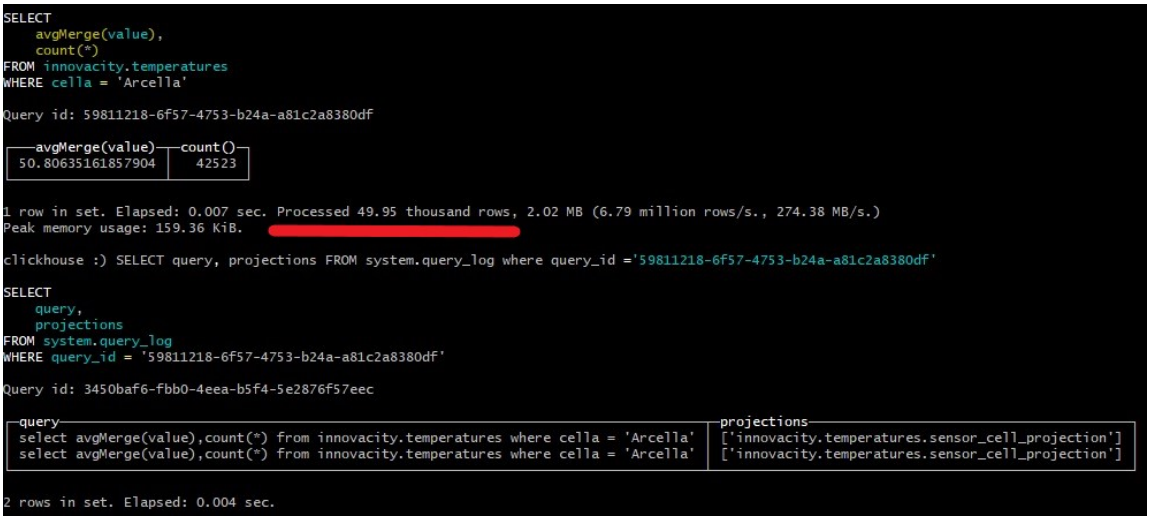

Figure 15: Query esempio projection - ClickHouse

Il totale di righe processate per ottenere il risultato è di **49'950** con **7**ms di tempo utilizzati e un picco di memoria RAM impiegata di **159.36**KiB.

3.7.2 Integrazione Kafka tramite Kafka Engine in ClickHouse

ClickHouse$_G$ supporta l'*integrazione*$_G$ con *Kafka*$_G$ tramite *Kafka*$_G$ Engine, permettendo la lettura dei dati da un topic *Kafka*$_G$ e il loro salvataggio in una tabella *ClickHouse*$_G$ adatta a grandi dataset. Tale funzionalità riveste un'importanza notevole per applicazioni che richiedono l'elaborazione in tempo reale di dati provenienti da fonti esterne, una necessità frequente nel contesto del monitoraggio urbano. L'*integrazione*$_G$ con *Kafka*$_G$ consente l'acquisizione e la memorizzazione efficiente di dati, anche in grosse quantità, garantendo comunque prestazioni elevate.
Kafka$_G$ Engine è progettato per il recupero di dati una sola volta. Ciò significa che una volta che i dati vengono interrogati da una tabella *Kafka*$_G$, vengono considerati consumati dalla coda. Pertanto, non si dovrebbero mai selezionare dati direttamente da una tabella di *Kafka*$_G$ Engine, ma utilizzare invece una Materialized View.
Una Materialized View viene attivata una volta che i dati sono disponibili in una tabella di *Kafka*$_G$ Engine. Automaticamente i dati vengono spostati da una tabella *Kafka*$_G$ a una tabella di tipo MergeTree o Distributed.

Sono quindi necessarie almeno 3 tabelle:

- La tabella di origine del *Kafka*$_G$ Engine;
- La tabella di destinazione (MergeTree o Distributed);
- Materialized View per spostare i dati.

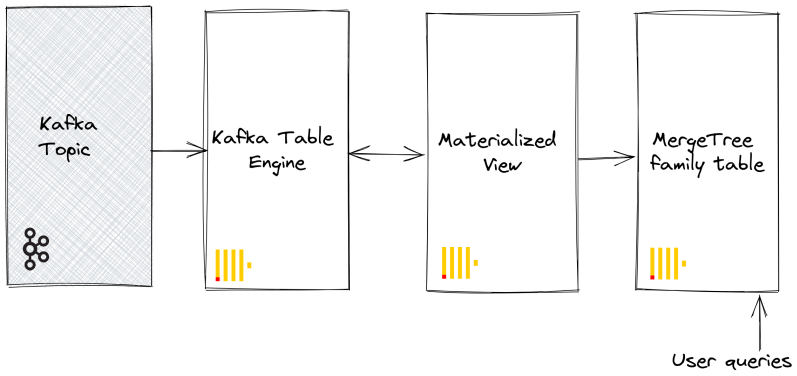

Figure 16: Architettura di Kafka Engine in ClickHouse

3.7.3 Trasferimento dati tramite Materialized View

Una materialized view funge da ponte tra la fonte dei dati (*Kafka*$_G$ Engine) e la destinazione dei dati (MergeTree). Quando nuovi dati vengono scritti nella tabella *Kafka*$_G$ Engine, la materialized view viene attivata automaticamente.
La materialized view esegue una *query*$_G$ sulla tabella *Kafka*$_G$ Engine per selezionare i dati più recenti. Una volta selezionati, questi dati vengono inseriti nella tabella di destinazione (ad esempio, una tabella MergeTree). Questo processo avviene in modo automatico e immediato, senza bisogno di intervento manuale.
In pratica, la materialized view si assicura che la tabella di destinazione sia sempre aggiornata con i dati più recenti presenti nella tabella *Kafka*$_G$ Engine.
Questo offre numerosi vantaggi:

- **Automatizzazione del processo:** Non è necessario eseguire manualmente operazioni di trasferimento dati da una tabella all'altra. La materialized view si occupa di tutto in modo automatico;
- **Efficienza:** Il trasferimento dei dati avviene in tempo reale, garantendo che la tabella di destinazione sia sempre allineata con la fonte dei dati senza ritardi;

- **Ottimizzazione delle risorse:** Il processo di trasferimento dei dati è gestito in modo efficiente, utilizzando al meglio le risorse disponibili e garantendo prestazioni elevate.

Nel contesto specifico, le materialized view sono responsabili di eseguire controlli sui dati, come ad esempio la verifica della loro correttezza ed affidabilità nel contesto di utilizzo, prima di inserirli nella tabella di destinazione.
Questo processo assicura che i dati siano sempre affidabili e pronti per l'analisi, senza la necessità di ulteriori operazioni di pulizia o preparazione.
Per esempio, nel caso dei dati di umidità raccolti da sensori in un'area urbana, la materialized view potrebbe eseguire controlli per assicurarsi che i valori rientrino all'interno di un intervallo plausibile e che non ci siano discrepanze improbabili. Ciò garantirebbe che i dati di umidità inseriti nella tabella di destinazione siano accurati e affidabili per l'analisi meteorologica o ambientale.

3.7.4 Tabella Kafka Engine per un sensore generico

Le tabelle del *database*$_G$ impiegate per ottenere le misurazioni di ciascuna tipologia di *sensore*$_G$ dai topic *kafka*$_G$ presentano una configurazione sostanzialmente simile, differenziandosi principalmente per il tipo di dato della colonna relativa alla misurazione e per il *topic* di riferimento utilizzato per ottenere le misurazioni.
Nello specifico per ogni *sensore*$_G$ si avrà la seguente tabella *Clickhouse*$_G$:

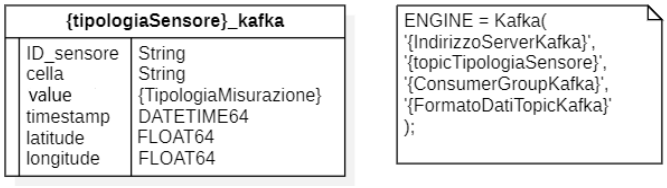

Figure 17: Tabella sensore generico per il reperimento da Kafka - ClickHouse

La tabella è configurata con il motore di *storage*$_G$ *Kafka*$_G$, il che significa che i dati verranno letti direttamente da un topic *Kafka*$_G$.
I campi sono:

- **ID_sensore:** Un campo di tipo *String* che identifica univocamente il *sensore*$_G$ che ha effettuato la misurazione;
- **cella:** Un campo di tipo *String* che rappresenta la cella della città in cui è stata effettuata la misurazione;
- **value:** Un campo di tipo variabile a seconda del tipo di misurazione;

- **timestamp:** Un campo di tipo *DATETIME64* che rappresenta il timestamp della misurazione;
- **latitude:** Un campo di tipo *Float64* che rappresenta la latitudine del luogo dove è stata effettuata la misurazione;
- **longitude:** Un campo di tipo *Float64* che rappresenta la longitudine del luogo dove è stata effettuata la misurazione.

Mentre i parametri esposti racchiusi da parentesi graffe variano per ogni tipolgia di *sensore*$_G$ correlato alla misurazione e sono:

- **tipologiaSensore:** Viene sostituito con la tipologia del *sensore*$_G$ che effettua le misurazioni salvate nella tabella (ex. temperatures);
- **TipologiaMisurazione:** Viene sostituito con il tipo del dato che rappresenta la misurazione;
- **IndirizzoServerKafka:** Specifica l'indirizzo del server *Kafka*$_G$. Nel nostro caso il server è in esecuzione su un *container*$_G$ *Docker* raggiungibile tramite l'indirizzo: *'kafka:9092'*;
- **topicTipologiaSensore:** Specifica il nome del topic *Kafka*$_G$ da cui leggere i dati (ex.temperature). Accetta anche liste di topic *Kafka*$_G$ separati da virgole;
- **ConsumerGroupKafka:** Specifica il nome del consumer group *Kafka*$_G$ che verrà utilizzato per leggere i messaggi dal topic *Kafka*$_G$. Un consumer group in *Kafka*$_G$ è un gruppo di consumatori che lavorano insieme per consumare i messaggi da uno o più topic. Ogni messaggio inviato a un topic *Kafka*$_G$ può essere consumato da uno dei consumatori nel gruppo.

  I consumer all'interno di uno stesso gruppo condividono l'elaborazione dei messaggi all'interno dei topic: ogni messaggio viene elaborato da uno e un solo consumatore all'interno del gruppo. Nel nostro caso sarà sempre '*CG_Clickhouse_1*' per indicare il *servizio*$_G$ di salvataggio *Clickhouse*;
- **FormatoDatiTopicKafka:** specifica il formato dei dati nel topic *Kafka*$_G$. Nel nostro caso, i dati sono nel formato JSONEachRow, che è un formato di serializzazione JSON di *ClickHouse* che consente di scrivere o leggere record JSON separati da una riga;
- **KafkaSkipBrokenMessages:**
  - è un'opzione di configurazione utilizzata nell'Engine *Kafka*$_G$ di *ClickHouse*$_G$. Determina il comportamento dell'Engine quando incontra messaggi *Kafka*$_G$ considerati "corrotti" o non processabili. Un messaggio *Kafka*$_G$ può essere considerato corrotto per diversi motivi, tra cui:

    - **Formato non valido:** Il messaggio potrebbe avere un formato JSON o Avro non valido, impedendo a *ClickHouse*$_G$ di decodificarlo correttamente;
    - **Dati mancanti:** Il messaggio potrebbe contenere dati mancanti o incompleti, violando lo schema previsto;
    - **Errori di codifica:** Il messaggio potrebbe avere errori di codifica che impediscono la lettura dei dati.
  - Per impostazione predefinita, *kafka_skip_broken_messages* è impostato su 0. Ciò significa che *ClickHouse*$_G$ interrompe l'elaborazione del flusso di dati da *Kafka*$_G$ e registra un errore quando incontra un messaggio corrotto;
  - Puoi configurare *kafka_skip_broken_messages* su un valore diverso da zero per modificare il comportamento. Il valore rappresenta il numero massimo di messaggi corrotti consecutivi per blocco, considerato nel contesto di *kafka_max_block_size*, che *ClickHouse*$_G$ ignorerà prima di interrompere l'elaborazione;
  - Bisogna anche ricordare che per come è stato progettato il *sistema*$_G$ i messaggi corrotti vengono scartati "alla fonte" dallo Schema Registry di *Kafka*$_G$;
  - Nel nostro caso vogliamo che ogni messaggio malformato nel blocco venga ignorato.

- **input_format_skip_unknown_fields**: è un'impostazione utilizzata con alcuni formati di input di *ClickHouse*$_G$, compreso quello da noi utilizzato *JSONEachRow* per specificare come gestire i dati in entrata che contengono colonne sconosciute alla tabella di destinazione. Impostando *input_format_skip_unknown_fields* su 1, *ClickHouse*$_G$ ignorerà le colonne sconosciute nei dati in entrata e importerà solo le colonne corrispondenti alla tabella di destinazione. Questo è utile quando si desidera importare solo una parte dei dati in entrata, ignorando le colonne non necessarie o non rilevanti. Nel nostro caso l'impostazione di default è quella richiesta.

3.7.5 Misurazioni temperatura

Di seguito viene presentata una configurazione dettagliata per l'archiviazione delle misurazioni di temperatura. Tale configurazione, progettata per acquisire dati da un topic *Kafka*$_G$ permette la persistenza delle misurazioni di temperatura che include l'*ID*$_G$ del *sensore*$_G$ (String), la cella della città da cui proviene la misurazione (String), la temperatura misurata (Float32), il timestamp della misurazione (DATETIME64), la latitudine (Float64) e la longitudine (Float64) del *sensore*$_G$.
La tabella di destinazione è denominata temperatures e utilizza il motore di *storage*$_G$ MergeTree, che è ottimizzato per l'archiviazione e l'analisi di dati cronologicamente ordinati in append-only. Questa scelta è giustificata dal fatto che le misurazioni di temperatura sono tipicamente ordinate cronologicamente.

Da osservare che l'attributo presente nel messaggio JSON : *type* viene ignorato, poichè non necessario in una tabella dedicata alle sole misurazioni di temperatura. Ciò accade grazie al valore di deault dell'impostazione *input_format_skip_unknown_fields* della tabella con Kakfa Engine. Vengono definite come *PRIMARY KEY* le colonne *ID_sensore* e *timestamp* per garantire l'unicità delle misurazioni e la possibilità di effettuare ricerche efficienti.

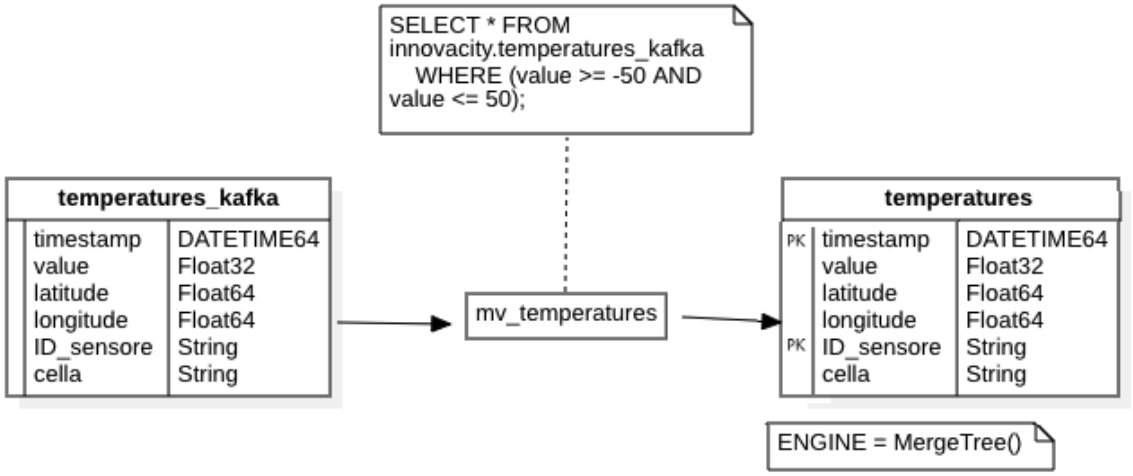

Figure 18: Tabella temperatures_kafka e temperatures

La tabella temperatures_kafka come spiegato in precedenza funge da tramite tra il topic *Kafka*$_G$ relativo alle misurazioni di temperatura e il *sistema*$_G$ di gestione dei dati *ClickHouse*$_G$. Questa tabella agisce come un'interfaccia trasformando i flussi di dati provenienti dal topic *Kafka*$_G$ in un formato comprensibile per *ClickHouse*$_G$. Successivamente, una Materialized View, in questo caso mv_temperatures, opera su questa tabella per trasferire i dati ottenuti verso la tabella di destinazione temperatures della famiglia MergeTree come spiegato in 3.7.3.

3.7.5.1 Projections per misurazioni di temperatura

Durante la fase di progettazione, è stata dedicata particolare attenzione all'utilizzo delle tabelle precedentemente descritte e alle richieste che verranno formulate su di esse. È emerso che, considerando il requisito di suddividere la città in una serie di celle e specificare la cella di origine della misurazione, la filtrazione delle misurazioni per celle diventerà una richiesta frequente al *database*$_G$. Di conseguenza, si è optato per l'utilizzo delle projections per ottimizzare il filtering su tale campo, le quali sono dettagliatamente descritte nella sezione 3.7.1.6.

```
1 --Projection per tabella temperatures
```

```
2   ALTER TABLE innovacity.temperatures ADD PROJECTION
        tmp_sensor_cell_projection (SELECT * ORDER BY cella);
3   ALTER TABLE innovacity.temperatures MATERIALIZE PROJECTION
        tmp_sensor_cell_projection;
```

Listing 2: implementazione projection tabella temperatures

La proiezione ci consentirà di effettuare rapidamente filtraggi basati sulle celle, anche se tale attributo non è definito come *PRIMARY_KEY* nella tabella originale.

3.7.5.2 TTL per misurazioni di temperatura

L'implementazione del Time To Live *(TTL)* di rollup per le misurazioni di temperatura deve permettere di salvare un solo dato aggregato per ora e per *sensore*$_G$ dopo che è trascorso più di un mese dal timestamp della misurazione.

```
1   TTL toDateTime(timestamp) + INTERVAL 1 MONTH
2   GROUP BY ID_sensore, toStartOfHour(timestamp)
3   SET value = avg(value);
```

Listing 3: implementazione TTL tabella temperatures

3.7.5.3 Partition per misurazioni di temperatura

La tabella è partizionata per anno e mese in base al valore della colonna timestamp. Ciò significa che i dati vengono archiviati in parti separate per ogni combinazione di anno e mese. Questo approccio consente di organizzare i dati in modo efficiente e di eseguire *query*$_G$ su intervalli temporali specifici in modo rapido ed efficiente. Inoltre, il partizionamento per anno e mese consente di applicare il TTL in modo selettivo. La scelta sull'utilizzo delle Partition e sulla relativa configurazione è stata dettata dalle considerazioni esposte in 3.7.1.5.

3.7.6 Misurazioni umidità

Le considerazioni relative al salvataggio delle misurazioni di umidità coincidono con quelle espresse nella sezione 3.7.5 riguardo alle misurazioni di temperatura. In questa situazione, dove le misure riguardano l'umidità, la tabella di destinazione *ClickHouse*$_G$ è nominata 'humidity': Il tipo della colonna *value* è *Float32* poichè il valore dell'umidità è in percentuale, compreso tra 0 e 100. Le altre colonne sono identiche a quelle della tabella 'temperatures' e sono definite con lo stesso tipo di dato per garantire la precisione e l'integrità dei dati.

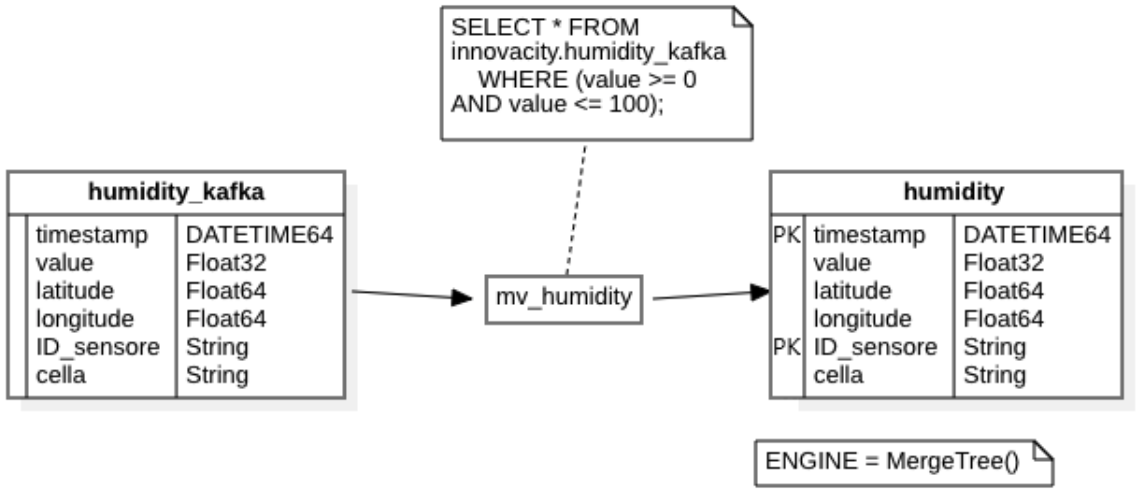

Figure 19: Tabella humidity_kafka e humidity

3.7.6.1 Projections per misurazioni di umidità

Dopo aver considerato le stesse argomentazioni presentate nella sezione 3.7.5.1 riguardanti le misurazioni di temperatura, abbiamo deciso di estendere l'utilizzo delle projection anche alle misurazioni di umidità. I vantaggi ottenuti risultano essere simili a quelli evidenziati per le misurazioni di temperatura, come descritto nella stessa sezione. A seguire, vengono illustrate le configurazioni delle projection relative alle tabelle delle misurazioni di umidità:

```
1  --Projection per tabella humidity
2  ALTER TABLE innovacity.humidity ADD PROJECTION
       umd_sensor_cell_projection (SELECT * ORDER BY cella);
3  ALTER TABLE innovacity.humidity MATERIALIZE PROJECTION
       umd_sensor_cell_projection;
```

3.7.6.2 Partition & TTL per misurazioni di umidità

La configurazione riguardante il partizionamento e il TTL per le misurazioni di umidità corrisponde a quella descritta nella sezione 3.7.5.2 e 3.7.5.3 in merito alle misurazioni di temperatura.

3.7.7 Misurazioni di polveri sottili

Le considerazioni concernenti l'archiviazione delle misurazioni di polveri sottili corrispondono a quelle espresse nella sezione 3.7.5 in merito alle misurazioni di temperatura. Il tipo della

colonna *value* è *Float32* poichè il valore delle polveri sottili è espresso in microgrammi per metro cubo (µg/m^3), un valore compreso tra 0 e 1000. La tabella di destinazione *ClickHouse*$_G$ è nominata 'dust_PM10'.
Le altre colonne sono identiche a quelle della tabella 'temperatures' e sono definite con lo stesso tipo di dato per garantire la precisione e l'integrità dei dati.

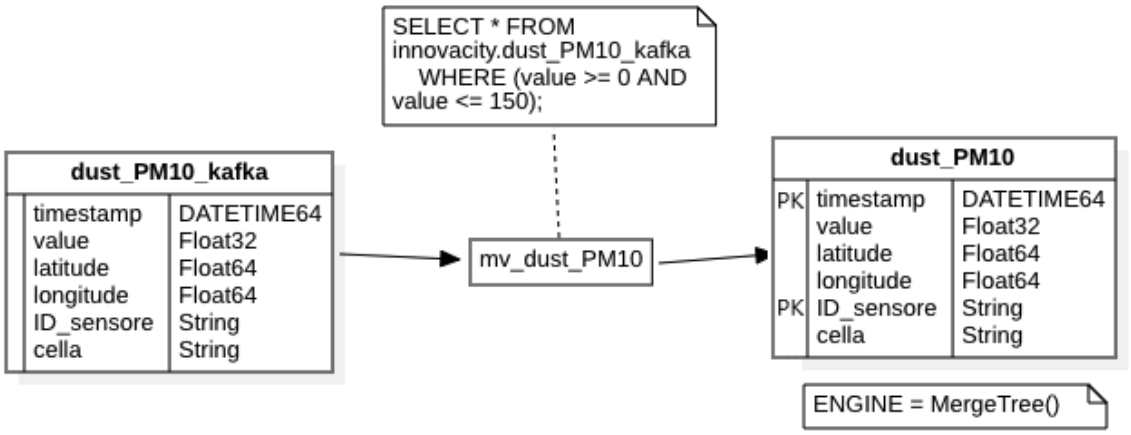

Figure 20: Tabella dust_PM10_kafka e dust_PM10

3.7.7.1 Projections per misurazioni di polveri sottili

Dopo aver considerato le stesse argomentazioni presentate nella sezione 3.7.5.1 riguardanti le misurazioni di temperatura, abbiamo deciso di estendere l'utilizzo delle projection anche alle misurazioni di polveri sottili. I vantaggi ottenuti risultano essere simili a quelli evidenziati per le misurazioni di temperatura, come descritto nella stessa sezione. A seguire, vengono illustrate le configurazioni delle projection relative alle tabelle delle misurazioni di polveri sottili:

```
1 --Projection per tabella dust_PM10
2 ALTER TABLE innovacity.dust_PM10 ADD PROJECTION
      dust_sensor_cell_projection (SELECT * ORDER BY cella);
3 ALTER TABLE innovacity.dust_PM10 MATERIALIZE PROJECTION
      dust_sensor_cell_projection;
```

3.7.7.2 Partition & TTL per misurazioni di polveri sottili

La configurazione riguardante il partizionamento e il TTL per le misurazioni di dust_PM10 corrisponde a quella descritta nella sezione 3.7.5.2 e 3.7.5.3 in merito alle misurazioni di temperatura.

3.7.8 Misurazioni isole ecologiche

Le considerazioni concernenti l'archiviazione delle misurazioni delle isole ecologiche corrispondono a quelle espresse nella sezione 3.7.5 in merito alle misurazioni di temperatura. Il tipo della colonna *value* è *Float32* poichè il valore della misurazione riguarda la percentuale di riempimento, compreso tra 0 e 100. Le altre colonne sono identiche a quelle della tabella 'temperatures' e sono definite con lo stesso tipo di dato per garantire la precisione e l'integrità dei dati.

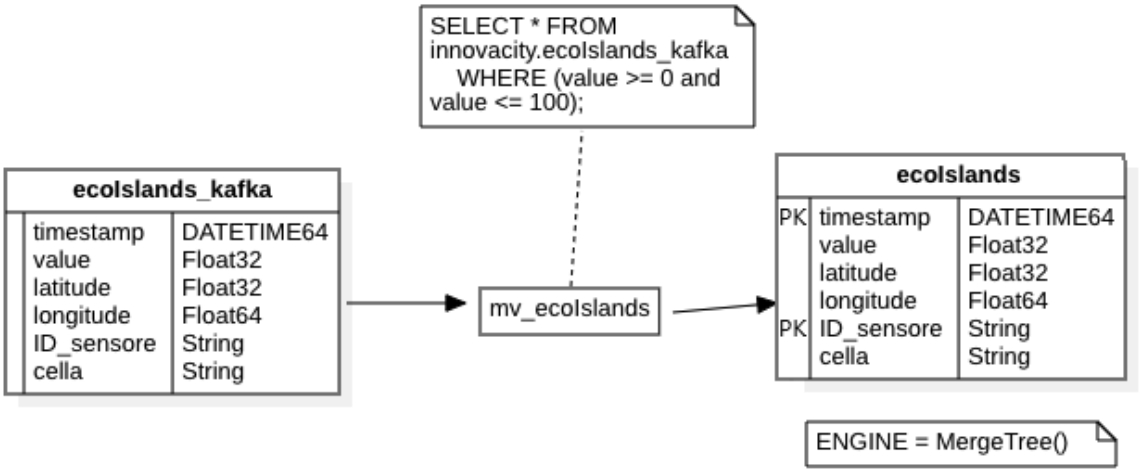

Figure 21: Tabella ecoIslands_kafka e ecoIslands

3.7.8.1 Projections per misurazioni delle isole ecologiche

Dopo aver considerato le stesse argomentazioni presentate nella sezione 3.7.5.1 riguardanti le misurazioni di temperatura, abbiamo deciso di estendere l'utilizzo delle projection anche alle misurazioni delle isole ecologiche. I vantaggi ottenuti risultano essere simili a quelli evidenziati per le misurazioni di temperatura, come descritto nella stessa sezione. A seguire, vengono illustrate le configurazioni delle projection relative alle tabelle delle misurazioni delle isole ecologiche:

```
1 --Projection per tabella ecoIslands
2   ALTER TABLE innovacity.ecoIslands ADD PROJECTION
        umd_sensor_cell_projection (SELECT * ORDER BY cella);
3   ALTER TABLE innovacity.ecoIslands MATERIALIZE PROJECTION
        umd_sensor_cell_projection;
```

3.7.8.2 Partition & TTL per misurazioni di umidità

La configurazione riguardante il partizionamento e il TTL per le misurazioni di riempimento delle isole ecologiche corrisponde a quella descritta nella sezione 3.7.5.2 e 3.7.5.3 in merito alle misurazioni di temperatura.

3.7.9 Misurazioni guasti elettrici

Le considerazioni concernenti l'archiviazione delle misurazioni dei guasti elettrici corrispondono a quelle espresse nella sezione 3.7.5 in merito alle misurazioni di temperatura. Il tipo della colonna *value* è *UInt8* poichè il valore delle misurazioni dei guasti elettrici è binario, 0 (guasto rilevato) oppure 1 (guasto non rilevato).
La tabella di destinazione *ClickHouse*$_G$ è nominata 'electricalFault': Le altre colonne sono identiche a quelle della tabella 'temperatures' e sono definite con lo stesso tipo di dato per garantire la precisione e l'integrità dei dati.

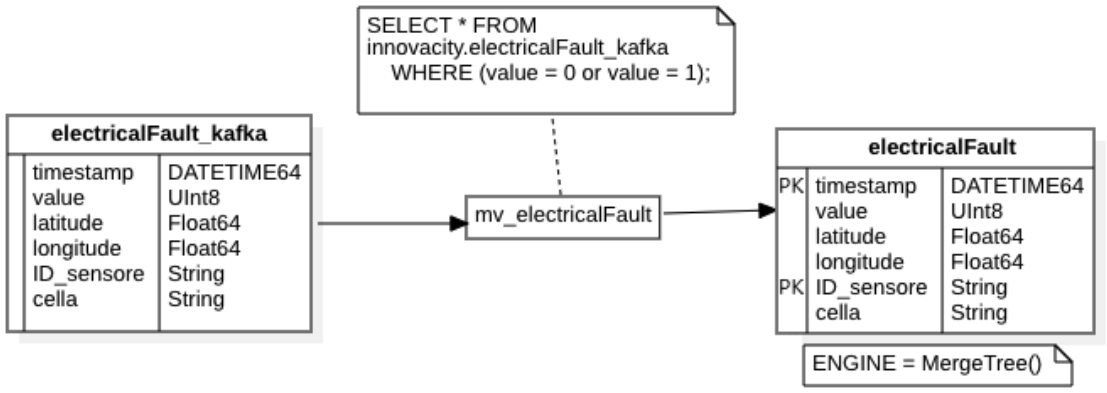

Figure 22: Tabella electricalFault_kafka e electricalFault

3.7.9.1 Considerazioni su projection, partition e TTL per misurazioni di guasti elettrici

Le proiezioni per questo tipo di misurazione non offrono la stessa utilità rispetto a quelle relative a sensori con misurazioni non binarie, dove spesso si eseguono analisi temporali su ampi periodi e grandi volumi di dati filtrati per cella.
In questo contesto di misurazione, le operazioni di selezione sulla tabella si concentrano sul recupero dell'ultimo valore registrato.
La creazione di una proiezione per questa tabella comporterebbe solo un sovraccarico di calcolo e spazio su disco utilizzato, senza un reale vantaggio in termini di prestazioni o funzionalità.

Inoltre non si verificano le condizioni che giustifichino l'utilizzo del partizionamento e del TTL, in quanto non avrebbe senso aggregare misurazioni di questo genere dopo un certo periodo di tempo che porterebbero a misurazioni prive di valore informativo o senza significato.

3.7.10 Misurazioni stazioni di ricarica

Le considerazioni concernenti l'archiviazione delle misurazioni delle stazioni di ricarica corrispondono a quelle espresse nella sezione 3.7.9 in merito alle misurazioni guasti elettrici. Il tipo della colonna *value* è *UInt8* poichè il valore delle misurazioni delle stazioni di ricarica è binario, 0 (stazione di ricarica non occupata) oppure 1 (stazione di ricarica occupata).
La tabella di destinazione *ClickHouse*$_G$ è nominata 'chargingStation':

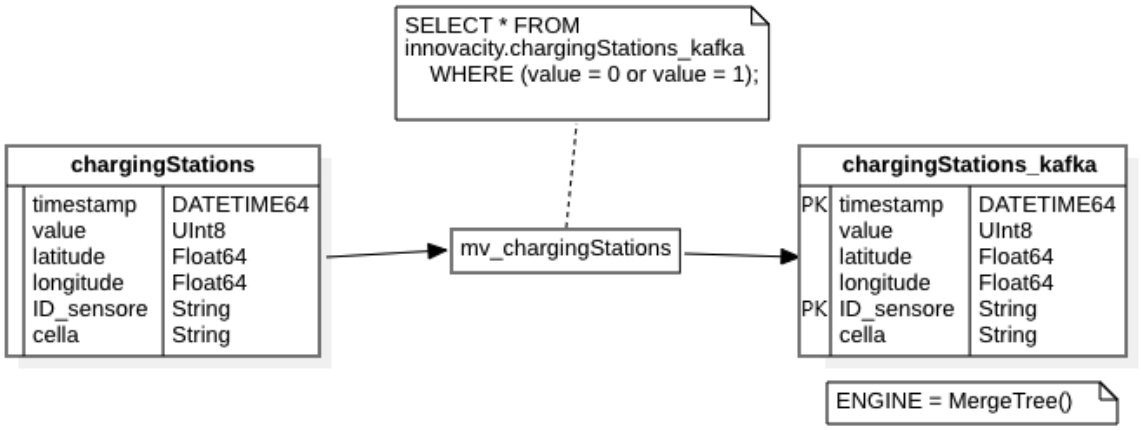

Figure 23: Tabella chargingStation_kafka e chargingStation

3.7.11 Misurazioni sensori di rilevameno dell'acqua

Le considerazioni concernenti l'archiviazione delle misurazioni dei sensori di livello dell'acqua corrispondono a quelle espresse nella sezione 3.7.9 in merito alle misurazioni guasti elettrici. Il tipo della colonna *value* è *UInt8* poichè il valore delle misurazioni dei sensori di livello dell'acqua è binario, 0 (acqua non rilevata) oppure 1 (acqua rilevata).
La tabella di destinazione *ClickHouse*$_G$ è nominata 'waterPresence':

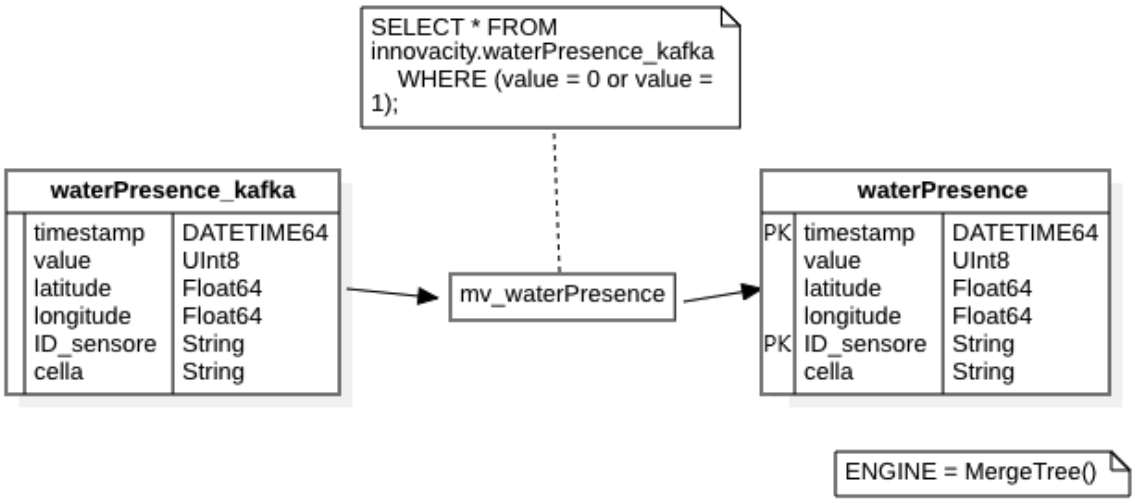

Figure 24: Tabella waterPresence_kafka e waterPresence

3.7.12 Punteggi di salute

Viene creata anche una tabella dedicata alla persistenza dei punteggi di salute nel tempo calcolati per ogni cella della città. La tabella di destinazione *ClickHouse*$_G$ è nominata 'healthScore', i campi sono: cella (String), value (Float32), timestamp(DATETIME64).

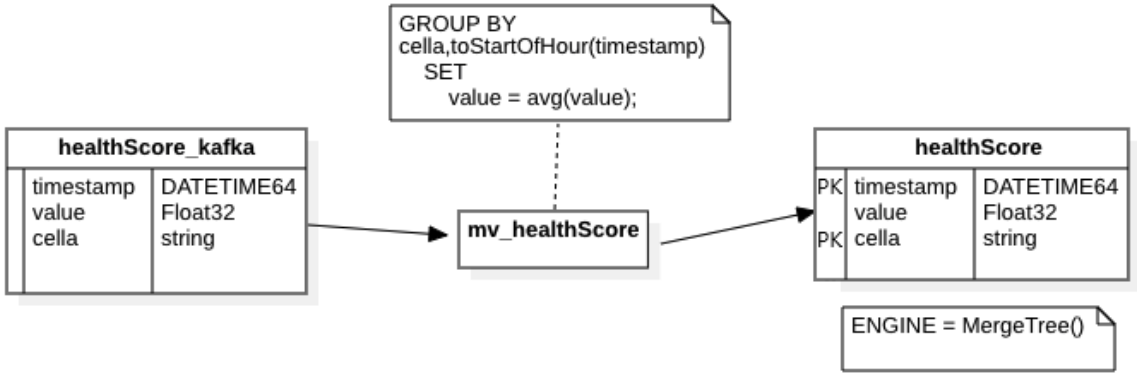

Figure 25: Tabella healthScore_kafka e healthScore

3.7.12.1 Time to Live punteggio di salute

La tabella dedicata alle misurazioni di salute include la configurazione di un TTL che consente l'aggregazione dei dati dopo un mese dal timestamp della misurazione in una singola misurazione per cella e per ora.
Questo approccio consente di ridurre l'utilizzo dello spazio di archiviazione e accelerare il tempo di interrogazione, mantenendo comunque un livello di dettaglio adeguato per le analisi.

3.8 Grafana

Grafana$_G$ è un *software*$_G$ open source per la visualizzazione e l'analisi dei dati progettato per interagire con vari data-source, tra cui *Clickhouse*$_G$. *Grafana*$_G$ offre un'interfaccia utente intuitiva e flessibile che consente di creare e condividere *dashboard*$_G$ personalizzate per monitorare i dati di diversa natura in tempo reale.

3.8.1 Utenti

L'accesso a *Grafana*$_G$ è vincolato a due utenze e non permette uteriori registrazioni per l'accesso alla *piattaforma*$_G$ di monitoraggio.

- **Amministratore:**
  - Accesso riservato all'amministratore di *sistema*$_G$ per permettere manutenzione e modifiche alle impostazioni sensibili della *piattaforma*$_G$;
  - Non accessibile in produzione;
  - Credenziali:
    - **Username:** admin
    - **Password:** admin
- **User:**
  - Accesso riservato alle autorità locali per la visualizzazione e il monitoraggio dei dati;
  - Credenziali:
    - **Username:** user
    - **Password:** user

3.8.2 Dashboards

Per soddisfare tutti i requisiti definiti in *Analisi dei requisiti v2.0.0 - Sez. Req. Funzionali* sono state create due *dashboard*$_G$:

- **Dashboard Principale:** Questa *dashboard*$_G$ fornisce una visualizzazione chiara e intuitiva delle misurazioni provenienti da tutti i sensori, di tutte le tipologie, distribuiti nell'area urbana. La *dashboard*$_G$ include una mappa interattiva della città che mostra la posizione geografica di ciascun *sensore*$_G$ e la relativa ultima misurazione. Inoltre, viene presentato il punteggio di salute della città o di celle specifiche;
- **Dashboard dedicata:** Mostra le misurazioni di una specifica tipologia di *sensore*$_G$ selezionata dall'utente in modo più dettagliato e permette di effettuare le *attività*$_G$ di filtraggio e aggregazione definite in *Analisi dei requisiti v2.0.0 - Sez. Req. Funzionali*.

Dashboard Principale - Progettazione in dettaglio

La *dashboard*$_G$ principale è suddivisa in righe comprimibili. Di seguito, vengono elencate le informazioni da visualizzare per ciascuna riga, nell'ordine dato dall'enumerazione, **l'intervallo di tempo scelto dall'utente tramite interfaccia di default *Grafana*$_G$ verrà chiamato *UserInterval*.**

1. Titolo riga: **City manager**
   (a) Pannello per la scelta delle celle della città di cui si intende visualizzare le misurazioni;
   (b) Mappa interattiva della città, presenta i sensori interni alle celle selezionate;
   (c) Pannello per la visualizzazione del punteggio di salute relativo alla celle selezionate;
   (d) Pannello per la visualizzazione dello stato degli alert relativi alle celle selezionate.
2. Titolo riga: **Temperatura**
   (a) Pannello per la scelta dei sensori di temperatura da analizzare (vengono proposti per la scelta solo i sensori interni alle celle selezionate per l'analisi).
   Verrà chiamato *tmps* l'insieme dei sensori di temperatura scelti nel pannello appena esposto e presenti nelle celle selezionate nel pannello apposito;
   (b) Pannello con vista time-series delle misurazioni di temperatura. Vengono presentate le misurazioni dei sensori *tmps* in *UserInterval*;
   (c) Pannello con vista della media delle misurazioni dei sensori *tmps* in *UserInterval*.
3. Titolo riga: **Umidità**

(a) Pannello per la scelta dei sensori di umidità da analizzare (vengono proposti per la scelta solo i sensori interni alle celle selezionate per l'analisi).
Verrà chiamato *umds* l'insieme dei sensori di umidità scelti nel pannello appena esposto e presenti nelle celle selezionate nel pannello apposito;

(b) Pannello con vista time-series delle misurazioni di umidità. Vengono presentate le misurazioni dei sensori *umds* in *UserInterval*;

(c) Pannello con vista della media delle misurazioni dei sensori *umds* in *UserInterval*.

4. Titolo riga: **Polveri sottili**

   (a) Pannello per la scelta dei sensori di polveri sottili da analizzare (vengono proposti per la scelta solo i sensori interni alle celle selezionate per l'analisi).
   Verrà chiamato *pm_sensors* l'insieme dei sensori di polveri sottili scelti nel pannello appena esposto e presenti nelle celle selezionate nel pannello apposito;

   (b) Pannello con vista time-series delle misurazioni di polveri sottili. Vengono presentate le misurazioni dei sensori *pm_sensors* in *UserInterval*;

   (c) Pannello con vista della media delle misurazioni dei sensori *pm_sensors* in *UserInterval*.

5. Titolo riga: **Isole ecologiche**

   (a) Pannello per la scelta delle isole ecologiche da analizzare (vengono proposte per la scelta solo le isole ecologiche interne alle celle selezionate per l'analisi).
   Verrà chiamato *islands* l'insieme delle isole ecologiche scelte nel pannello appena esposto;

   (b) Pannello con vista time-series delle misurazioni delle isole ecologiche. Vengono presentate le misurazioni delle isole ecologiche *islands* in *UserInterval*;

   (c) Pannello con vista della media delle misurazioni delle isole ecologiche *islands* in *UserInterval*.

6. Titolo riga: **Colonnine di ricarica**

   (a) Pannello per la scelta delle colonnine di ricarica da analizzare (vengono proposte per la scelta solo le colonnine di ricarica interne alle celle selezionate per l'analisi).
   Verrà chiamato *chsSt* l'insieme delle colonnine di ricarica scelte nel pannello appena esposto;

   (b) Pannello con vista sul numero di colonnine di ricarica libere in *chsSt* considerando l'ultima misurazione in *UserInterval*;

(c) Pannello con vista sul numero di colonnine di ricarica occupate in *chsSt* considerando l'ultima misurazione in *UserInterval*;

(d) Pannello con vista tabellare delle ultime misurazioni in *UserInterval* delle colonnine di ricarica in *chsSt*.

7. Titolo riga: **Guasti elettrici**

   (a) Pannello per la scelta dei sensori di guasti elettrici da analizzare (vengono proposti per la scelta solo i sensori di guasti elettrici interni alle celle selezionate per l'analisi). **Verrà chiamato *GstEl* l'insieme dei sensori di guasti elettrici scelti nel pannello appena esposto**;

   (b) Pannello con vista sul numero di sensori che hanno rilevato anomalie in *GstEl* considerando l'ultima misurazione in *UserInterval*;

   (c) Pannello con vista sul numero di sensori che non hanno rilevato anomalie in *GstEl* considerando l'ultima misurazione in *UserInterval*;

   (d) Pannello con vista tabellare delle ultime misurazioni in *UserInterval* dei sensori *GstEl*.

8. Titolo riga: **Sensori di presenza dell'acqua**

   (a) Pannello per la selezione dei sensori di presenza dell'acqua da analizzare (vengono proposti solo i sensori di presenza dell'acqua interni alle celle selezionate per l'analisi).
   Verrà chiamato *PresAcq* l'insieme dei sensori di presenza dell'acqua scelti nel pannello appena esposto;

   (b) Pannello con vista sul numero di sensori che hanno rilevato acqua in *PresAcq* considerando l'ultima misurazione nell'intervallo temporale definito dall'utente (*UserInterval*);

   (c) Pannello con vista sul numero di sensori che non hanno rilevato acqua in *PresAcq* considerando l'ultima misurazione nell'intervallo temporale definito dall'utente (*UserInterval*);

   (d) Pannello con vista tabellare delle ultime misurazioni nell'intervallo (*UserInterval*) dei sensori *PresAcq*.

Dashboard dedicata - Progettazione in dettaglio

La $dashboard_G$ dedicata è suddivisa in due sezioni: la prima, posta nella parte superiore della $dashboard_G$, è dedicata ai pannelli per le variabili di input, mentre la seconda, posta nella parte inferiore della $dashboard_G$, è dedicata alla visualizzazione delle misurazioni dei sensori secondo le impostazioni selezionate.

Di seguito vengono esposti i dettagli relativi alla progettazione della selezione delle variabili di input:

1. Pannello "**Selezione cella**": Permette di selezionare le celle della città da analizzare;
2. Pannello "**Tipologia misurazioni**": Permette di selezionare la tipologia di sensori da analizzare;
3. Pannello "**Selezione sensori**": Permette di selezionare i sensori da analizzare relativi alla tipologia e alla cella selezionata;
4. Pannello "**Aggregazione temporale**": Permette di selezionare l'intervallo temporale di aggregazione delle misurazioni:Automatico, Secondo, Minuto, Ora, Giorno, Mese, Nessuno. Maggiori dettagli in 3.8.4;
5. Pannello "**Misurazine minima**": Permette di selezionare il valore minimo delle misurazioni da visualizzare;
6. Pannello "**Misurazione massima**": Permette di selezionare il valore massimo delle misurazioni da visualizzare.

Di seguito vengono esposti i dettagli relativi alla progettazione della visualizzazione delle misurazioni secondo le variabili selezionate:

1. Pannello "**Grafico a linee**": Visualizzazione delle misurazioni attraverso un grafico a linee time-series. Le misurazioni sono mostrate in base ai parametri specificati dall'utente. In particolare le misurazioni esposte rispettano i parametri scelti dall'utente nella sezione di selezione delle variabili di input sopra elencate;
2. Pannello "**Tabella misurazioni**": Visualizzazione delle misurazioni in forma tabellare. Come nel pannello precedente, le misurazioni sono mostrate in base ai parametri specificati dall'utente.

3.8.3 ClickHouse data source plugin

3.8.3.1 Documentazione:

https://grafana.com/grafana/plugins/grafana-clickhouse-datasource/
Questo plugin di grafana consente di connettersi a un'istanza di *ClickHouse*$_G$ e di visualizzarne i dati in tempo reale. È possibile eseguire *query*$_G$ SQL personalizzate e visualizzare i risultati in forma di grafici, tabelle e pannelli personalizzati. Il plugin offre anche funzionalità di aggregazione e di calcolo dei dati, consentendo di analizzare e visualizzare i dati in modo flessibile e personalizzato.

3.8.3.2 Data sources configuration

La configurazione del data source avviene tramite file *yaml* che deve essere presente in *"/provisioning/datasources"*. Il protocollo di trasporto utilizzato è *TLS* e, se necessario, può essere modificato nel file appena citato grazie al parametro di configurazione *protocol*.

3.8.3.3 Macro utilizzate

Per semplificare la sintassi e consentire operazioni dinamiche, come filtri per intervalli temporali, le *query*$_G$ al *database*$_G$ *Clickhouse*$_G$ possono contenere macro. Le macro utilizzate sono:

- **$__timeFilter(columnName)**: Permette di effettuare il filtro temporale alla *query*$_G$ per ottenere le sole misurazioni all'interno dell'intervallo di tempo selezionato dall'utente;
- **$__timeInterval(columnName)**: Permette di modificare il raggruppamento temporale delle misurazioni in automatico sulla base dell'ampiezza dell'intervallo temporale selezionato dall'utente. In questo modo è possibile avere una visione ottimizzate delle misurazioni.

3.8.4 Variabili Grafana

Documentazione: https://grafana.com/docs/grafana/latest/dashboards/variables/ (Consultato: 25/03/2024)
Le variabili in *Grafana*$_G$ sono un potente strumento per rendere le *dashboard*$_G$ dinamiche e interattive. Permettono di filtrare i dati visualizzati in base a valori scelti dall'utente, rendendo la *dashboard*$_G$ più versatile e adattabile a differenti esigenze.

Variabili nella dashboard principale:

Nella *dashboard*$_G$ principale, le variabili sono:

- **variabile ($cella)**: Per mostrare solo le misurazioni provenienti da determinate celle della città;
- **variabili ($<TipoSensore>_sensors_id)**: Per mostrare le misurazioni di determinati sensori di un certo tipo.

Queste variabili, all'interno delle *query*$_G$ al *database*$_G$, permettono il filtraggio delle misurazioni sulla base di quanto selezionato dall'utente. Un esempio di *query*$_G$ per la visualizzazione delle misurazioni time-series di temperatura è:

```
1  SELECT    ID_sensore, avg(value) as value,
2            $__timeInterval(timestamp) as timestamp
3  FROM    innovacity.temperatures
```

```
4   WHERE    $__timeFilter(timestamp) AND cella IN ($Cella) AND ID_sensore in (
        ${tmp_sensors_id})
5   GROUP BY ID_sensore, timestamp;
```

La *query*$_G$ mostra anche l'utilizzo delle macro esposte in: 3.8.3.3

Variabili nella dashboard dedicata:

Nella *dashboard*$_G$ dedicata alla visualizzazione specifica delle misurazioni di una sola tipologia sono presenti le seguenti variabili:

- **variabile ($cella)**: Per mostrare solo le misurazioni provenienti da determinate celle della città;
- **variabili ($<TipoSensore>_sensors_id)**: Per mostrare le misurazioni di determinati sensori di un certo tipo;
- **variabili ($tabella)**: Per selezionare la tipoligia di *sensore*$_G$ di cui si vuole visualizzare la *dashboard*$_G$ dedicata e quindi la tabella del *database*$_G$ da cui ricavare i dati;
- **($aggregazione)**: Per selezionare l'intervallo temporale di aggregazione delle misurazioni (Automatico, Secondo, Minuto, Ora, Giorno, Mese, Nessuno). Nel caso della selezione della modalità "Automatico" si utilizza l'intervallo temporale di aggregazione più opportuno sulla base dell'ampiezza dell'intervallo temporale selezionato dall'utente;
- **($Max_value)**: Variabile ad input numerico per filtrare le misurazioni con valore al di sotto di quello indicato;
- **($Min_value)**: Variabile ad input numerico per filtrare le misurazioni con valore al di sopra di quello indicato.

3.8.5 Grafana alerts

Documentazione: `https://grafana.com/docs/grafana/latest/alerting/` (Consultato: 25/03/2024)
Grafana$_G$ offre un *sistema*$_G$ di alerting completo per monitorare i dati e inviare notifiche quando si verificano determinate condizioni. Le notifiche possono essere inviate tramite diversi canali, tra cui email, Slack, Telegram e *Discord*$_G$.

3.8.5.1 Alert Rule

Per poter configurare un alert è necessario creare una regola di alert. Tale regola viene impostata tramite *query*$_G$ al data source e fa scattare l'alert quando la *query*$_G$ restituisce un risultato che soddisfa le condizioni impostate. Gli alert sono configurati per i seguenti eventi:

- Quando un *sensore*$_G$ di temperatura registra una temperatura superiore ai 40°C o inferiore ai -10°C;
- Quando un *sensore*$_G$ di polveri sottili supera i 50 microgrammi al metro cubo;
- Quando un *sensore*$_G$ di guasti elettrici rileva un guasto.

Gli alert attraversano 3 stati:

- **Pending:** La condizione per l'attivazione dell'avviso è stata soddisfatta, ma il periodo di valutazione dell'avviso non è ancora trascorso;
- **Firing:** Indica che un alert è stato attivato e la sua valutazione ha confermato che la condizione di alert è soddisfatta per il periodo impostato nella regola e quindi viene inviata la notifica ai canali impostati;
- **OK:** Indica che un alert è stato disattivato e la sua valutazione ha confermato che la condizione di alert non è più soddisfatta.

  Le regole di allerta sono configurabili tramite l'interfaccia grafica di *Grafana*$_G$ e vengono esportate in formato *yaml* ed inserite in *"/provisioning/alerting"*.

3.8.5.2 Configurazione canale di notifica

Per configurare i canali di notifica è necessario andare in *Alerting* e selezionare *Notification channels* dall'interfaccia grafica di *Grafana*$_G$.
Per il progetto è stato scelto *Discord*$_G$ come unico canale di notifica.
Per configurare il canale di notifica è necessario seguire i seguenti passaggi:

- Seleziona *Discord*$_G$ come canale di notifica;
- Configurazione Server *Discord*$_G$ "InnovaCity":
  - Il server *Discord*$_G$ è stato creato ed è raggiungibile tramite l'indirizzo: `https://discord.gg/cCp9qxK7`;
  - Per ottenere il webhook URL del canale *Discord*$_G$ andare in: *Impostazioni server/Integrazioni* e seleziona *visualizza webhook*.
- Inserisci il webhook URL del tuo canale *Discord*$_G$;
- Personalizza il messaggio di notifica.

Anche le impostazioni di configurazione del canale di notifica sono esportabili in formato *yaml* e vengono inserite in */provisioning/alerting*.

3.8.5.3 Notification policies

Le norme di notifica negli alert di *Grafana*$_G$ sono un modo potente per gestire l'invio degli alert a diversi canali di notifica.
Per una spiegazione dettagliata della configurazione si rimanda alla documentazione ufficiale di *Grafana*$_G$: `https://grafana.com/docs/grafana/latest/alerting/alerting-rules/create-notification-policy/` (Consultato: 25/03/2024).
Anche le impostazioni delle notification policies sono esportabili in formato *yaml* e vengono inserite in */provisioning/alerting*.

3.8.6 Altri plugin utilizzati

3.8.6.1 Orchestra Cities Map plugin

Documentazione:
`https://grafana.com/grafana/plugins/orchestracities-map-panel/` (Consultato: 25/03/2024)
Il plugin Orchestra Cities Map per *Grafana*$_G$ estende il pannello Geomap con diverse funzionalità avanzate per la visualizzazione di dati geolocalizzati su mappe. Viene utilizzato al fine di consentire una rappresentazione differenziata delle icone corrispondenti ai vari tipi di sensori distribuiti in città, nonché la visualizzazione dell'ultima misurazione effettuata, ovvero lo stato attuale del *sensore*$_G$ stesso.
Funzionalità principali:

- **Supporto per GeoJSON**: Permette di visualizzare dati GeoJSON su mappe, come shapefile di città, regioni o stati;
- **Icone personalizzate**: Permette di utilizzare icone personalizzate per rappresentare diversi tipi di dati sui punti mappa;
- **Popup informativi**: Permette di visualizzare popup con informazioni dettagliate quando si clicca su un punto mappa;
- **Strati multipli**: Permette di creare più strati sovrapposti per visualizzare diversi set di dati sulla stessa mappa;
- **Filtraggio e ricerca**: Permette di filtrare i punti mappa in base a diversi criteri, come proprietà dei dati o valori delle metriche;
- **Colorazione dei punti**: Permette di colorare i punti mappa in base a valori di metriche o ad altri criteri;
- **Legende personalizzate**: Permette di creare legende personalizzate per spiegare il significato dei colori e delle icone utilizzati nella mappa.

3.8.6.2 Variable Panel plugin

Documentazione: https://volkovlabs.io/plugins/volkovlabs-variable-panel/
(Consultato: 25/03/2024)
Il plugin permette di creare dei pannelli *Grafana*$_G$ che possono essere posizionati ovunque nella *dashboard*$_G$ e che consentono di selezionare i valori delle variabili. In aggiunta, il *sistema*$_G$ consente la rappresentazione a forma di albero delle variabili, la quale risulta vantaggiosa nel nostro contesto in cui i sensori sono localizzati all'interno delle celle della città.

4 Architettura di deployment

Al fine di implementare ed eseguire l'intero *stack tecnologico*$_G$ ed i layer del modello architetturale, è stato configurato un ambiente *Docker*$_G$ che simula la suddivisione e la distribuzione dei servizi. Informazioni aggiuntive sulle immagini utilizzate e sulle configurazioni dell'ambiente sono disponibili nel file *docker-compose.yml* presente nel *repository*$_G$ *MVP* del progetto oltre che nella sezione: 2.1.
In particolare,per l'ambiente di produzione, sono stati creati i seguenti *container*$_G$:

- **Data feed**:
  - Container **Simulators**:
    - Esegue i **simulatori dei sensori** per la raccolta dei dati;
    - Produce dati nel formato JSON definito nello Schema Registry e li invia al *broker*$_G$ *Kafka*$_G$;
- **Streaming layer**:
  - Container **Kafka**:
    - Esegue **Apache Kafka** per la gestione del flusso di dati in tempo reale;
    - Accessibile agli altri *container*$_G$ tramite l'indirizzo ***kafka:9092***.
  - Componenti di supporto:
    - Container **Zookeper**:
      - Esegue il *servizio*$_G$ di coordinamento per *Kafka*$_G$, oltre alla memorizzazione e al controllo degli schemi;
      - Accessibile dagli altri *container*$_G$ attraverso l'indirizzo ***zookeeper:2181***.
    - Container **Schema Registry**:
      - Esegue il *servizio*$_G$ di registrazione degli schemi per *Kafka*$_G$;
      - Accessibile dagli altri *container*$_G$ attraverso l'indirizzo ***schema_registry:8081***;

        - Per la registrazione degli schemi vengono eseguiti *container*$_G$ che tramite *API*$_G$ REST (fornita dallo schema registry) permettono la registrazione degli schemi JSON.

- **Processing layer**:
  - Container **Faust**:
    - Esegue l'app **Faust** per il processing e il calcolo del punteggio di salute;
- **Storage layer**:
  - Container **Clickhouse**:
    - Esegue **Clickhouse** per lo *storage*$_G$ delle misurazioni;
    - La banca dati è accessibile agli altri *container*$_G$ tramite l'indirizzo ***clickhouse:8123***.
- **Data Visualization Layer**:
  - Container **Grafana**:
    - Esegue **Grafana** come interfaccia utente per la visualizzazione dei dati;
    - Espone la porta ***3000*** all'esterno per permettere l'accesso al *servizio*$_G$ di dashboarding.

Questa struttura permette una distribuzione modulare e scalabile del *sistema*$_G$, semplificando la gestione e la manutenzione dei componenti e consentendo una rapida scalabilità in risposta alle esigenze emergenti. Grazie all'uso di *Docker*$_G$, si garantisce coerenza e riproducibilità dell'ambiente di esecuzione, semplificando il deployment e garantendo maggiore affidabilità nell'ambiente di produzione nonchè la possibilità di attribuire le risorse necessarie ad ogni *servizio*$_G$ in modo mirato.

5 Stato dei requisiti funzionali

Si riporta ciascun requisito mediante il corrispondente codice, rispetto alla stessa tabella presente nel documento *Analisi dei Requisiti v2.0.0 - Sez. Req. Funzionali*, qui è presente una colonna indicante la soddisfazione di tale requisito.

Codice	Importanza	Descrizione	Stato

RF0	Obbligatorio	L'accesso al prodotto è vincolato da un *sistema*$_G$ di *login*$_G$, tuttavia, non è necessario che gli utenti si registrino autonomamente. Le credenziali di accesso sono fornite da terze parti o dall'amministratore del *sistema*$_G$.	Soddisfatto
RF1	Obbligatorio	Il prodotto non deve avere una gestione di amministrazione.	Soddisfatto
RF2	Obbligatorio	Il *sistema*$_G$ deve integrare simulatori di diverso tipo al fine di generare dati di misurazioni che siano coerenti con l'ambito del *sensore*$_G$ simulato.	Soddisfatto
RF3	Obbligatorio	Ogni misurazione trasmessa dal simulatore del *sensore*$_G$ deve essere composta dall'id del *sensore*$_G$, il timestamp e la misurazione.	Soddisfatto
RF4	Obbligatorio	Il *sistema*$_G$ deve essere in grado di simulare almeno un *sensore*$_G$ che rilevi la temperatura espressa in gradi Celsius.	Soddisfatto
RF5	Obbligatorio	Il *sistema*$_G$ deve essere in grado di simulare almeno un *sensore*$_G$ che misuri l'umidità, espressa in percentuale di umidità nell'aria.	Soddisfatto
RF6	Obbligatorio	Il *sistema*$_G$ deve essere in grado di simulare almeno un *sensore*$_G$ per la rilevazione delle particelle di polveri sottili presenti nell'aria, espresse in microgrammi per metro cubo.	Soddisfatto

RF7	Obbligatorio	Il *sistema*$_G$ deve includere la simulazione di almeno un *sensore*$_G$ per individuare guasti elettrici. Questi sensori segnalano interruzioni nella fornitura di energia elettrica tramite un *bit*$_G$ binario, con il valore 0 che indica l'assenza di energia elettrica.	Soddisfatto
RF8	Obbligatorio	Il *sistema*$_G$ deve essere in grado di simulare almeno un *sensore*$_G$ per monitorare lo stato di riempimento dei diversi conferitori nelle isole ecologiche. L'indicazione fornita sarà una percentuale di riempimento dell'isola ecologica.	Soddisfatto
RF9	Obbligatorio	Il *sistema*$_G$ deve includere la simulazione di almeno un *sensore*$_G$ per le colonnine di ricarica. Questi sensori indicheranno tramite un *bit*$_G$ binario se la colonnina è occupata (*bit*$_G$ 1) o libera (*bit*$_G$ 0).	Soddisfatto
RF10	Obbligatorio	Il *sistema*$_G$ deve includere la simulazione di almeno un *sensore*$_G$ per il livello dell'acqua. Questi sensori indicheranno il livello dell'acqua.	Soddisfatto
RF11	Obbligatorio	Ogni dato generato dai simulatori dei sensori deve essere strettamente correlato al dato successivo, garantendo così una transizione realistica e plausibile tra le misurazioni.	Soddisfatto

RF12	Obbligatorio	Il *sistema*$_G$ deve essere in grado di memorizzare in modo sicuro e efficiente i dati generati dai sensori. Ciò include la registrazione accurata di ogni misurazione, assicurando l'integrità e la coerenza dei dati.	Soddisfatto
RF13	Obbligatorio	La *piattaforma*$_G$ deve supportare la visualizzazione di dati provenienti da diversi tipi di sensori.	Soddisfatto
RF14	Obbligatorio	L'utente deve poter visualizzare una *dashboard*$_G$ con una panoramica completa dello stato della città tramite l'utilizzo di *widget*$_G$ adibiti alla rappresentazione delle misurazioni dei sensori.	Soddisfatto
RF15	Obbligatorio	L'utente deve avere la possibilità di visualizzare le misurazioni all'interno dei *widget*$_G$ adibiti alla rappresentazione delle rilevazioni dei sensori in formato grafico time series.	Soddisfatto
RF16	Obbligatorio	L'utente deve avere la possibilità di visualizzare le misurazioni all'interno dei *widget*$_G$ adibiti alla rappresentazione delle rilevazioni dei sensori in formato testuale time series.	Soddisfatto

RF17	Obbligatorio	La visualizzazione delle misurazioni in formato testuale time series deve presentare le informazioni nel formato: `IDSensore ,TIMESTAMP, Dato`.	Soddisfatto
RF18	Obbligatorio	L'utente deve essere in grado di visualizzare le ultime misurazioni all'interno dei *widget*$_G$ dedicati alla presentazione dei rilevamenti dei sensori che trasmettono dati binari (ex. Occupato/Libero) attraverso una mappa interattiva. La mappa, tramite etichette adeguate, deve rappresentare chiaramente il valore corrispondente all'ultima misurazione effettuata da ciascun *sensore*$_G$.	Soddisfatto
RF19	Obbligatorio	La *dashboard*$_G$ richiede un aggiornamento quasi istantaneo per garantire che i dati provenienti dai sensori siano riflessi nel minor tempo possibile, entro un massimo di 10 secondi.	Soddisfatto
RF20	Obbligatorio	La *dashboard*$_G$ deve mostrare un *widget*$_G$ distinto per ciascun tipo di *sensore*$_G$ attivo che trasmette dati al *sistema*$_G$, contenente le misurazioni in formato grafico, testuale o mappa interattiva.	Soddisfatto

RF21	Obbligatorio	Ogni *widget*$_G$ che visualizza le misurazioni deve includere, insieme ai dati stessi, informazioni sull'identificativo dei sensori che hanno contribuito a quelle misurazioni.	Soddisfatto
RF22	Obbligatorio	La *dashboard*$_G$ deve includere un *widget*$_G$ dedicato alle misurazioni dei sensori di temperatura.	Soddisfatto
RF23	Obbligatorio	Il *widget*$_G$ destinato alla rappresentazione delle misurazioni effettuate dai sensori di temperatura deve offrire all'utente di default la visualizzazione di tali dati in un formato grafico a linee, con una linea corrispondente a ciascun *sensore*$_G$ coinvolto.	Soddisfatto
RF24	Obbligatorio	La *dashboard*$_G$ deve includere un *widget*$_G$ dedicato alle misurazioni dei sensori di umidità.	Soddisfatto
RF25	Obbligatorio	Il *widget*$_G$ destinato alla rappresentazione delle misurazioni effettuate dai sensori di umidità deve offrire all'utente di default la visualizzazione di tali dati in un formato grafico a linee, con una linea corrispondente a ciascun *sensore*$_G$ coinvolto.	Soddisfatto
RF26	Obbligatorio	La *dashboard*$_G$ deve includere un *widget*$_G$ dedicato alle misurazioni dei sensori delle polveri sottili.	Soddisfatto

RF27	Obbligatorio	Il *widget*$_G$ destinato alla rappresentazione temporale delle misurazioni effettuate dai sensori di polveri sottili deve offrire all'utente la possibilità di visualizzare tali dati in un formato grafico a linee, con una linea corrispondente a ciascun *sensore*$_G$ coinvolto.	Soddisfatto
RF28	Obbligatorio	La *dashboard*$_G$ deve includere un *widget*$_G$ dedicato alle misurazioni dei sensori dei guasti elettrici.	Soddisfatto
RF29	Obbligatorio	Il *widget*$_G$ destinato alla rappresentazione delle misurazioni effettuate dai sensori dei guasti elettrici deve offrire all'utente di default la visualizzazione di tali dati con una mappa interattiva delle ultime misurazioni.	Soddisfatto
RF30	Obbligatorio	La *dashboard*$_G$ deve includere un *widget*$_G$ dedicato alle misurazioni dei sensori di soglia delle isole ecologiche.	Soddisfatto
RF31	Obbligatorio	Il *widget*$_G$ destinato alla rappresentazione delle misurazioni effettuate dai sensori di soglia delle isole ecologiche deve offrire all'utente la visualizzazione di tali dati con una mappa interattiva delle ultime misurazioni.	Soddisfatto

RF32	Obbligatorio	La *dashboard*$_G$ deve includere un *widget*$_G$ dedicato alle misurazioni dei sensori delle colonnine di ricarica.	Soddisfatto
RF33	Obbligatorio	Il *widget*$_G$ destinato alla rappresentazione delle misurazioni effettuate dai sensori delle colonnine di ricarica deve offrire all'utente la visualizzazione di tali dati con una mappa interattiva delle ultime misurazioni.	Soddisfatto
RF34	Obbligatorio	La *dashboard*$_G$ deve includere un *widget*$_G$ dedicato alle misurazioni dei sensori del livello dell'acqua.	Soddisfatto
RF35	Obbligatorio	Il *widget*$_G$ destinato alla rappresentazione delle misurazioni effettuate dai sensori del livello dell'acqua deve offrire all'utente la visualizzazione di tali dati con una mappa interattiva delle ultime misurazioni.	Soddisfatto
RF36	Obbligatorio	La *dashboard*$_G$ della città deve includere una mappa interattiva che mostra la posizione dei diversi sensori nella città.	Soddisfatto
RF37	Obbligatorio	I sensori nella mappa devono essere etichettati in modo da consentirne il riconoscimento della tipologia.	Soddisfatto
RF38	Obbligatorio	I sensori posizionati sulla mappa devono visualizzare l'ultimo valore registrato quando il puntatore del mouse è posizionato sopra di essi.	Soddisfatto

RF39	Desiderabile	La *dashboard*$_G$ deve fornire un *widget*$_G$ con il punteggio di salute relativo alla città basato sui dati aggregati provenienti dai sensori.	Soddisfatto
RF40	Obbligatorio	L'utente deve avere la possibilità di selezionare una cella, ovvero un'area specifica della città, al fine di visualizzare una *dashboard*$_G$ dedicata contenente esclusivamente sensori, misurazioni e punteggio di salute correlati a essa.	Soddisfatto
RF41	Obbligatorio	L'utente deve poter filtrare la visualizzazione delle misurazioni di una specifica tipologia di sensori inserendo uno specifico intervallo temporale.	Soddisfatto
RF42	Obbligatorio	Il *sistema*$_G$ deve verificare la validità dell'intervallo temporale inserito dall'utente.	Soddisfatto
RF43	Obbligatorio	In caso di intervallo temporale non valido, il *sistema*$_G$ deve generare una notifica di errore.	Soddisfatto
RF44	Obbligatorio	La notifica di errore relativa all'inserimento di un intervallo temporale non valido deve richiedere all'utente di reinserire date valide.	Soddisfatto

RF45	Obbligatorio	La notifica di errore relativa all'inserimento di un intervallo temporale non valido deve essere chiara e informativa, indicando il motivo specifico dell'invalidità dell'intervallo temporale (data fine precedente a data inizio, arco temporale precedente o antecedente all'inizio della trasmissione dati).	Soddisfatto
RF46	Obbligatorio	L'utente ha la possibilità di selezionare l'intervallo temporale desiderato (secondo, minuto, ora, giorno, mese, anno) per aggregare le misurazioni in base al relativo periodo di registrazione corrispondente.	Soddisfatto
RF47	Obbligatorio	Il *sistema*$_G$ deve essere in grado di adattare dinamicamente la rappresentazione delle misurazioni secondo un intervallo temporale di aggregazione selezionato dall'utente.	Soddisfatto
RF48	Obbligatorio	L'utente deve avere la possibilità di definire due valori (un minimo e un massimo) per filtrare le misurazioni dei sensori di una specifica tipologia, utilizzando questi limiti come criterio per visualizzare solo i dati compresi in quei range.	Soddisfatto
RF49	Obbligatorio	Il *sistema*$_G$ deve verificare la validità dell'intervallo di rilevamento inserito dall'utente.	Soddisfatto

RF50	Obbligatorio	In caso di intervallo di rilevamento non valido, il *sistema*$_G$ deve generare una notifica di errore.	Soddisfatto
RF51	Obbligatorio	La notifica di errore relativa all'inserimento di un intervallo di rilevamento non valido deve richiedere all'utente di reinserire valori validi.	Soddisfatto
RF52	Obbligatorio	La notifica di errore relativa all'inserimento di un intervallo di rilevamento non valido deve essere chiara e informativa, indicando il motivo specifico dell'invalidità dell'intervallo di rilevamento (data fine precedente a data inizio, arco temporale precedente o antecedente all'inizio della trasmissione dati).	Soddisfatto
RF53	Obbligatorio	L'utente deve avere la possibilità di filtrare le misurazioni selezionando uno o più sensori di una specifica categoria in modo tale da visualizzare esclusivamente le misurazioni corrispondenti ai sensori selezionati.	Soddisfatto
RF54	Obbligatorio	L'utente deve poter filtrare la visualizzazione delle misurazioni di una specifica tipologia di sensori selezionando una o più specifiche celle come criterio di filtro.	Soddisfatto

RF55	Obbligatorio	L'utente deve poter applicare più filtri simultaneamente per la visualizzazione delle misurazioni di una specifica tipologia di sensori.	Soddisfatto
RF56	Obbligatorio	L'utente deve poter rimuovere i filtri applicati e ripristinare la visualizzazione senza tali filtri.	Soddisfatto
RF57	Opzionale	L'utente deve poter salvare in una lista di misurazioni rilevanti una misurazione trasmessa da un *sensore*$_G$.	Non soddisfatto
RF58	Opzionale	Il *sistema*$_G$ deve effettuare una verifica prima di salvare la misurazione tra le misurazioni rilevanti, assicurandosi che il dato non sia già presente in tale lista.	Non soddisfatto
RF59	Opzionale	L'utente deve poter visualizzare la lista delle misurazioni rilevanti.	Non soddisfatto
RF60	Opzionale	Ogni misurazione nella lista dei rilevanti deve fornire l'identificativo del *sensore*$_G$ che ha effettuato la misurazione.	Non soddisfatto
RF61	Opzionale	Ogni misurazione nella lista dei rilevanti deve fornire la tipologia del *sensore*$_G$ che ha effettuato la misurazione.	Non soddisfatto
RF62	Opzionale	Ogni misurazione nella lista dei rilevanti deve fornire l'orario e la data di misurazione.	Non soddisfatto

RF63	Opzionale	Ogni misurazione nella lista dei rilevanti deve fornire il valore misurato e la relativa unità di misura.	Non soddisfatto
RF64	Opzionale	L'utente deve poter rimuovere una misurazione dalla lista delle misurazioni rilevanti.	Non soddisfatto
RF65	Obbligatorio	L'utente deve essere in grado di ricevere notifiche nel caso in cui i sensori superino determinate soglie di sicurezza.	Soddisfatto
RF66	Obbligatorio	L'utente deve essere in grado di visualizzare le informazioni dei sensori.	Soddisfatto
RF67	Obbligatorio	L'utente deve essere in grado di visualizzare l'ID_G dei sensori.	Soddisfatto
RF68	Obbligatorio	L'utente deve essere in grado di visualizzare il tipo dei sensori.	Soddisfatto
RF69	Obbligatorio	L'utente deve essere in grado di visualizzare la posizione dei sensori in coordinate.	Soddisfatto
RF70	Obbligatorio	L'utente deve essere in grado di visualizzare la cella in cui è installato il $sensore_G$.	Soddisfatto
RF71	Obbligatorio	L'utente deve essere in grado di visualizzare la data di installazione dei sensori.	Soddisfatto
RF72	Obbligatorio	L'utente deve essere in grado di visualizzare l'unità di misura associata al $sensore_G$.	Soddisfatto
RF73	Obbligatorio	La $piattaforma_G$ deve poter ricevere più rilevazioni in parallelo.	Soddisfatto

5.1 Grafici riassuntivi requisiti soddisfatti

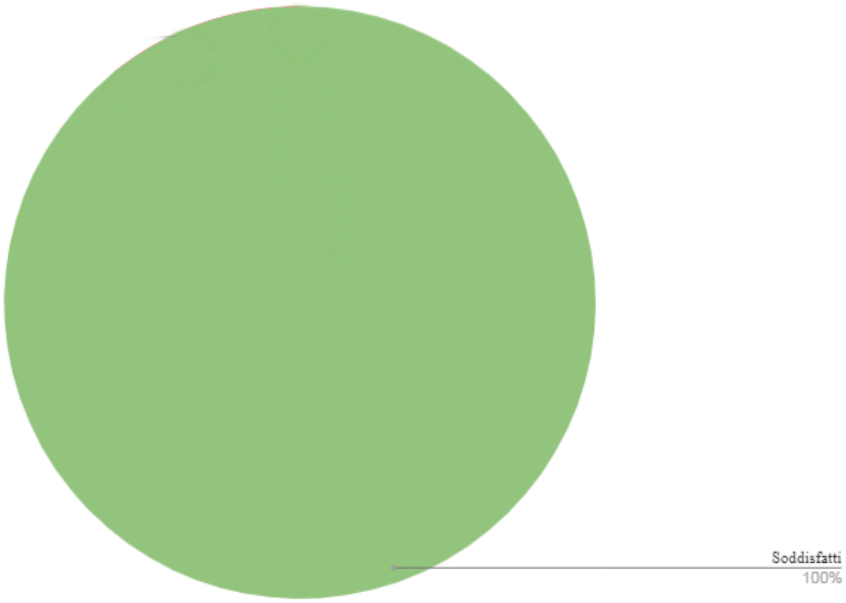

Figure 26: Stato dei requisiti funzionali obbligatori

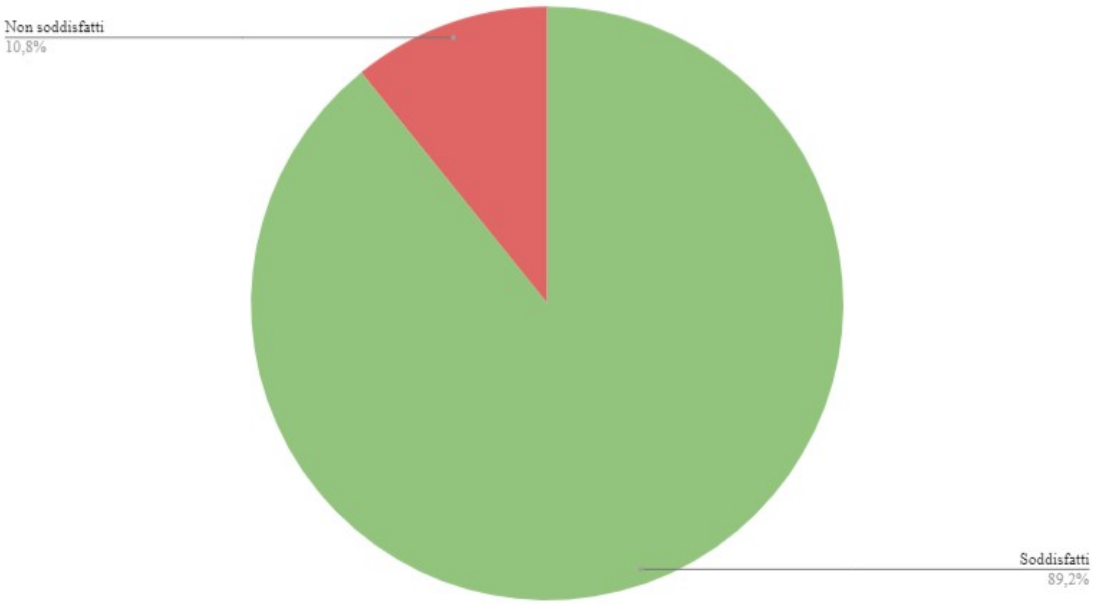

Figure 27: Stato dei requisiti funzionali totali