Specifica Tecnica

NearYou
Smart custom advertising platform

sevenbits.swe.unipd@gmail.com

Registro modifiche

Versione	Data	Autore	Verificatore	Descrizione
1.0.0	2025-04-02	Leonardo Trolese	Manuel Gusella	Approvazione documento per PB_G
0.2.4	2025-04-01	Riccardo Piva	Giovanni Cristellon	Cambio immagine schemaDB e piccole correzioni generali
0.2.3	2025-03-31	Federico Pivetta	Alfredo Rubino	Miglioramento alla descrizione dei design-pattern$_G$
0.2.2	2025-03-31	Alfredo Rubino	Federico Pivetta	Aggiornamento sezione pattern Adapter
0.2.1	2025-03-29	Federico Pivetta	Riccardo Piva	Correzioni generali
0.2.0	2025-03-28	Uncas Peruzzi	Federico Pivetta	Refactoring Struttura, Aggiunti SimulationService e PositionToMessageService
0.1.10	2025-03-28	Riccardo Piva	Federico Pivetta	Refactor sezione Clickhouse$_G$ e Grafana$_G$ architettura di sistema$_G$
0.1.9	2025-03-28	Uncas Peruzzi	Federico Pivetta	Aggiunte immagini servizi e diagrammi UML$_G$
0.1.8	2025-03-22	Alfredo Rubino	Uncas Peruzzi	Aggiunto pattern Adapter
0.1.7	2025-03-22	Federico Pivetta	Uncas Peruzzi	Ampliati i paragrafi per i pattern Strategy e Factory
0.1.6	2025-03-21	Riccardo Piva	Uncas Peruzzi	Sezione Integrazione Architettura logica e Architettura di sistema$_G$
0.1.5	2025-03-17	Riccardo Piva	Alfredo Rubino	Fix generali e miglioramenti
0.1.4	2025-03-16	Riccardo Piva	Alfredo Rubino	Redazione generale macro sezioni documento
0.1.3	2025-03-05	Alfredo Rubino	Manuel Gusella	Redazione sottosezione Strumenti e Servizi della sezione Tecnologie
0.1.2	2025-03-05	Leonardo Trolese	Manuel Gusella	Conclusione redazione sottosezione Panoramica dei Linguaggi della sezione Tecnologie
0.1.1	2025-03-02	Leonardo Trolese	Manuel Gusella	Redazione sottosezione Panoramica dei Linguaggi della sezione Tecnologie
0.1.0	2025-02-26	Leonardo Trolese	Manuel Gusella	Inizio redazione del documento

Indice

1 Introduzione **5**
1.1 Scopo del documento . . . 5
1.2 Glossario . . . 5
1.3 Riferimenti . . . 5
1.3.1 Riferimenti normativi . . . 5
1.3.2 Riferimenti informativi . . . 5

2 Tecnologie **6**
2.1 Panoramica tecnologica . . . 6
2.2 Linguaggi di programmazione . . . 6
2.2.1 Python . . . 6
2.2.1.1 Specifiche . . . 6
2.2.1.2 Ruolo nel progetto . . . 6
2.2.1.3 Dipendenze . . . 7
2.2.2 SQL . . . 8
2.2.2.1 Specifiche . . . 8
2.2.2.2 Ruolo nel progetto . . . 8
2.2.3 Formati di interscambio dati . . . 8
2.2.3.1 YAML . . . 8
2.2.3.2 Specifiche . . . 8
2.2.3.3 Ruolo nel progetto . . . 8
2.2.3.4 JSON . . . 8
2.2.3.5 Specifiche . . . 9
2.2.3.6 Ruolo nel progetto . . . 9
2.3 Infrastruttura e servizi . . . 9
2.3.1 Apache ZooKeeper . . . 9
2.3.1.1 Specifiche . . . 9
2.3.1.2 Ruolo nel progetto . . . 9
2.3.2 Apache Kafka . . . 9
2.3.2.1 Specifiche . . . 9
2.3.2.2 Ruolo nel progetto . . . 9
2.3.3 Apache Flink . . . 10
2.3.3.1 Specifiche . . . 10
2.3.3.2 Ruolo nel progetto . . . 10
2.3.4 ClickHouse . . . 10
2.3.4.1 Specifiche . . . 10
2.3.4.2 Ruolo nel progetto . . . 10
2.3.5 Grafana . . . 10
2.3.5.1 Specifiche . . . 11
2.3.5.2 Ruolo nel progetto . . . 11
2.3.6 Docker . . . 11
2.3.6.1 Specifiche . . . 11
2.3.6.2 Ruolo nel progetto . . . 11

3 Architettura logica **11**
3.1 Pattern di architettura-esagonale . . . 11
3.2 Servizi principali e loro componenti . . . 12

4 Architettura del Sistema **14**
4.1 Panoramica architetturale . . . 14
4.2 K-Architecture: Event Streaming Platform . . . 14
4.2.1 Motivazioni della scelta architetturale . . . 14
4.2.2 Componenti principali . . . 14
4.3 Integrazione Architettura logica e Architettura di sistema . . . 15
4.3.1 Descrizione . . . 15
4.3.2 Mappatura dei componenti . . . 16
4.4 Dataflow . . . 16

4.5 Implementazione tecnica dei componenti principali . . . 18
4.5.1 DataSource - Simulation Service . . . 18
4.5.1.1 Diagramma della classi . . . 18
4.5.1.2 Design Pattern - Strategy Pattern . . . 19
4.5.1.3 Design Pattern - Factory Pattern . . . 21
4.5.1.4 Design Pattern - Adapter Pattern . . . 22
4.5.2 Classi, interfacce, metodi e attributi . . . 24
4.5.2.1 SensorSimulationAdministrator . . . 24
4.5.2.2 SensorSubject . . . 25
4.5.2.3 GpsSensor . . . 25
4.5.2.4 GeoPosition . . . 25
4.5.2.5 IPositionSimulationStrategy . . . 26
4.5.2.6 BycicleSimulationStrategy . . . 26
4.5.2.7 GraphWrapper . . . 27
4.5.2.8 SensorFactory . . . 27
4.5.2.9 UserSensorService . . . 27
4.5.2.10 IUserRepository . . . 28
4.5.2.11 UserRepository . . . 28
4.5.2.12 UserDTO . . . 28
4.5.2.13 ISensorRepository . . . 29
4.5.2.14 SensorRepository . . . 29
4.5.2.15 SensorDTO . . . 30
4.5.2.16 DatabaseConnection . . . 30
4.5.2.17 DatabaseConfigParameters . . . 30
4.5.2.18 IJsonSerializable . . . 31
4.5.2.19 PositionJsonAdapter . . . 31
4.5.2.20 PositionSender . . . 31
4.5.2.21 KafkaConfluentAdapter . . . 32
4.5.2.22 KafkaConfigParameters . . . 32
4.5.3 Streaming Layer - Apache Kafka . . . 33
4.5.3.1 Topic e partitioning . . . 33
4.5.3.2 Producer e Consumer . . . 33
4.5.3.3 Integrazione con Flink keyed stream . . . 33
4.5.3.4 Schema topic simulator position . . . 33
4.5.3.5 Schema message elaborated . . . 33
4.5.3.6 Kafka poisoning . . . 33
4.5.4 Processing Layer - PositionToMessageProcessor . . . 35
4.5.4.1 Apache Flink . . . 35
4.5.4.2 Diagrammi delle classi . . . 36
4.5.4.3 Design Pattern - Adapter Pattern . . . 38
4.5.4.4 Design Pattern - Strategy Pattern . . . 39
4.5.4.5 Classi, interfacce, metodi e attributi: . . . 40
4.5.4.6 FlinkJobManager . . . 40
4.5.4.7 IMessageWriter . . . 40
4.5.4.8 KafkaMessageWriter . . . 41
4.5.4.9 JsonRowSerializationAdapter . . . 41
4.5.4.10 KafkaWriterConfiguration . . . 42
4.5.4.11 IPositionReceiver . . . 42
4.5.4.12 KafkaPositionReceiver . . . 42
4.5.4.13 JsonRowDeserializationAdapter . . . 43
4.5.4.14 KafkaSourceConfiguration . . . 43
4.5.4.15 FilterMessageValidator . . . 43
4.5.4.16 PositionToMessageProcessor . . . 44
4.5.4.17 LLMService . . . 45
4.5.4.18 CustomPrompt . . . 45
4.5.4.19 StructuredResponseMessage . . . 45
4.5.4.20 GroqLLMService . . . 46
4.5.4.21 IActivityRepository . . . 46

4.5.4.22 ClickhouseActivityRepository . . . 46
4.5.4.23 ActivityDTO . . . 47
4.5.4.24 IUserRepository . . . 47
4.5.4.25 ClickhouseUserRepository . . . 48
4.5.4.26 UserDTO . . . 48
4.5.4.27 IMessageRepository . . . 48
4.5.4.28 ClickhouseMessageRepository . . . 49
4.5.4.29 MessageDTO . . . 49
4.5.4.30 DatabaseConnection . . . 50
4.5.4.31 DatabaseConfigParameters . . . 50
4.5.4.32 IFlinkSerializable . . . 51
4.5.4.33 MessageSerializer . . . 51
4.5.4.34 FilterMessageAlreadyDisplayed . . . 51
4.5.5 ClickHouse . . . 52
4.5.5.1 Architettura MergeTree . . . 52
4.5.5.2 Schema del database . . . 53
4.5.6 Grafana . . . 58
4.5.6.1 Utenti . . . 58
4.5.6.2 Dashboards . . . 59
4.5.6.3 Dashboard generale . . . 59
4.5.6.4 Querying Clickhouse . . . 60
4.5.6.5 Variabili dashboard . . . 62
4.5.6.6 Trasformazioni e array interessi . . . 63
4.5.6.7 Connettore Clickhouse . . . 64
4.5.6.8 Provisioning automatico . . . 64
4.5.7 Best practices architetturali . . . 65
4.5.7.1 PEP8 - Stile di codifica Python . . . 65
4.5.7.2 Principi SOLID . . . 65
4.5.7.3 Dependency Injection . . . 67
4.6 Implementazione nel FlinkProcessor . . . 67

5 Architettura di deployment **70**
5.1 Panoramica dell'infrastruttura . . . 70
5.1.1 Ambiente Docker dei Componenti Principali . . . 70
5.1.1.1 Zookeeper Service . . . 70
5.1.1.2 Kafka Service . . . 70
5.1.1.3 Kafdrop Service . . . 71
5.1.1.4 Grafana Service . . . 72
5.1.1.5 ClickHouse Service . . . 73
5.1.1.6 Position Simulator Service . . . 74
5.1.1.7 Flink Service . . . 74
5.1.1.8 Test Service . . . 75
5.1.2 Dipendenze tra componenti . . . 76
5.2 Continuous Integration . . . 77
5.3 Vantaggi dell'architettura containerizzata . . . 78
5.4 Comunicazione tra container . . . 79
5.5 Orchestrazione e gestione . . . 79
5.6 Evoluzione futura . . . 79

6 Stato dei requisiti funzionali **80**
6.1 Riepilogo dei requisiti . . . 80
6.2 Tabella dei requisiti funzionali . . . 80
6.3 Stato di implementazione . . . 83
6.4 Riepilogo e Conclusioni . . . 84

Elenco delle figure

1 Architettura esagonale del SimulationService 12
2 Architettura esagonale del PositionToMessageService 13
3 Diagramma dell'architettura di Sistema$_G$ 15
4 Flusso dei dati nell'architettura 16
5 SimulationService Core 18
6 Factory di Sensori 19
7 PositionToMessageProcessorService InBound/OutBound Ports con il Broker$_G$ 37
8 PositionToMessageProcessorService OutBound Ports 38
9 Schema del database$_G$ 53
10 Stato dei requisiti funzionali obbligatori 83
11 Stato dei requisiti funzionali totali 84

Elenco delle tabelle

2 Mappatura dei componenti tra Kappa-architecture$_G$ e Architettura-esagonale$_G$ 16
4 Stato di implementazione dei requisiti funzionali 83

1 Introduzione

1.1 Scopo del documento

Il presente documento si propone come una risorsa completa per la comprensione degli aspetti tecnici e progettuali della piattaforma "NearYou", dedicata alla creazione di soluzioni di advertising personalizzato tramite intelligenza artificiale. L'obiettivo principale è fornire una descrizione dettagliata dell'architettura implementativa e di deployment, illustrando le tecnologie adottate e le motivazioni alla base delle scelte progettuali.
Nel contesto dell'architettura implementativa, il documento analizza nel dettaglio i moduli principali del sistema$_G$, i design-pattern$_G$ utilizzati. Saranno inclusi diagrammi delle classi, e una spiegazione dettagliata dei design-pattern$_G$ utilizzati e delle motivazioni di queste scelte.
Gli obiettivi di questo documento sono: motivare le decisioni architetturali, fungere da guida per lo sviluppo della piattaforma, e garantire la piena tracciabilità e copertura dei requisiti definiti nel documento di *Analisi dei Requisiti_v2.0.0*.
In sintesi, il documento intende essere un punto di riferimento essenziale per tutti gli attori coinvolti nel ciclo-di-vita$_G$ del progetto$_G$, offrendo una visione chiara e strutturata delle fondamenta tecniche che sorreggono NearYou e delle logiche che ne determinano il funzionamento.

1.2 Glossario

Con l'intento di evitare ambiguità interpretative del linguaggio utilizzato, viene fornito un Glossario che si occupa di esplicitare il significato dei termini che riguardano il contesto del Progetto$_G$. I termini presenti nel glossario sono contrassegnati con una G a pedice : Termine$_G$.
I termini composti, oltre alla $_G$ a pedice, saranno uniti da un "-" come segue: termine-composto$_G$.
Le definizioni sono presenti nell'apposito documento *Glossario_v2.0.0.pdf*.

1.3 Riferimenti

1.3.1 Riferimenti normativi

- Regolamento del Progetto$_G$ didattico
  https://www.math.unipd.it/~tullio/IS-1/2024/Dispense/PD1.pdf
  (Consultato: 2025-02-10).
- Capitolato$_G$ C4 - NearYou - Smart custom advertising platform
  https://www.math.unipd.it/~tullio/IS-1/2024/Progetto/C4p.pdf
  (Consultato: 2025-02-10).
- *Norme_di_Progetto_v2.0.0*

1.3.2 Riferimenti informativi

- *Glossario_v2.0.0*
- *Analisi_dei_Requisiti_v2.0.0*
- Analisi-dei-Requisiti$_G$ - SWE 2024-25
  https://www.math.unipd.it/~tullio/IS-1/2024/Dispense/T05.pdf
  (Consultato: 2025-02-10).
- Dependency Injection - SWE 2024-25
  https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Architetturali%20-%20Dependency%20Injection.pdf
  (Consultato: 2025-02-26).
- Design-pattern$_G$ Creazionali - SWE 2024-25
  https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Creazionali.pdf
  (Consultato: 2025-02-26).

- Design-pattern$_G$ Strutturali - SWE 2024-25
https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Strutturali.pdf
(Consultato: 2025-02-26).
- Software Architecture Patterns - SWE 2024-25
https://www.math.unipd.it/~rcardin/swea/2022/Software%20Architecture%20Patterns.pdf
(Consultato: 2025-02-26).
- Verbali Interni
- Verbali Esterni

2 Tecnologie

Questa sezione descrive le tecnologie utilizzate per lo sviluppo del sistema$_G$ NearYou, presentando una panoramica degli strumenti, dei linguaggi e dei servizi impiegati, con le motivazioni alla base delle scelte effettuate.

2.1 Panoramica tecnologica

Il sistema$_G$ NearYou si basa su un'architettura a microservizi event-driven che utilizza diverse tecnologie integrate:

- **Python**: Linguaggio principale per lo sviluppo dei componenti del sistema$_G$;
- **Apache Kafka$_G$**: Sistema$_G$ di messaggistica distribuito per la comunicazione tra componenti;
- **Apache Flink$_G$**: Framework$_G$ di elaborazione dati in tempo reale;
- **ClickHouse**: Database$_G$ colonnare ad alte prestazioni;
- **Grafana**: Piattaforma di visualizzazione dei dati in tempo reale;
- **Docker**: Sistema$_G$ di containerizzazione per il deployment.

2.2 Linguaggi di programmazione

2.2.1 Python

Linguaggio di programmazione ad alto livello, interpretato e orientato agli oggetti, scelto per la sua leggibilità, la vasta libreria standard e il ricco ecosistema di framework$_G$ disponibili, particolarmente adatto allo sviluppo rapido di applicazioni.

2.2.1.1 Specifiche

- **Versione**: 3.12.2;
- **Documentazione**: https://docs.python.org/ (Consultato: 2025-03-02).

2.2.1.2 Ruolo nel progetto

Nel contesto di NearYou, Python$_G$ viene impiegato per:

- Sviluppo di un simulatore per gli spostamenti di più utenti;
- Implementazione della logica di elaborazione dati;
- Interazione con i servizi esterni e le API$_G$;
- Gestione della persistenza dei dati;
- Applicazione degli algoritmi di selezione dei PoI$_G$ rilevanti.

2.2.1.3 Dipendenze

- **ClickHouse Connect**:
  - **Descrizione**: Libreria client per l'interazione con il database$_G$ Clickhouse$_G$, permettendo operazioni di query$_G$ e gestione dei dati;
  - **Versione**: 0.6.8;
  - **Documentazione**: https://clickhouse.com/docs/integrations/python (Consultato: 2025-03-02).
- **PyFlink**:
  - **Descrizione**: API$_G$ Python$_G$ di Apache Flink$_G$ per l'elaborazione di flussi di dati distribuiti, sia in modalità batch che streaming;
  - **Versione**: 1.18.1;
  - **Documentazione**: https://pyflink.readthedocs.io/en/main/getting_started/index.html (Consultato: 2025-03-02).
- **LangChain**:
  - **Descrizione**: Framework$_G$ per lo sviluppo di applicazioni basate su modelli linguistici, consentendo di orchestrare prompt e integrare fonti di dati esterne;
  - **Versione**: 0.1.12;
  - **Documentazione**: https://python.langchain.com/docs/introduction/ (Consultato: 2025-03-02).
- **Groq**:
  - **Descrizione**: Client Python$_G$ per l'API Groq$_G$, utilizzato per la generazione di contenuti tramite LLM$_G$;
  - **Versione**: 0.4.2;
  - **Documentazione**: https://console.groq.com/docs/libraries (Consultato: 2025-03-02).
- **Confluent Kafka$_G$**:
  - **Descrizione**: Libreria per l'interazione con Apache Kafka$_G$, utilizzata per la pubblicazione e sottoscrizione di messaggi;
  - **Versione**: 2.8.0;
  - **Documentazione**: https://docs.confluent.io/kafka/overview.html (Consultato: 2025-03-02).
- **GeoPy**:
  - **Descrizione**: Libreria per operazioni geospaziali e calcolo delle distanze;
  - **Versione**: 2.4.1;
  - **Documentazione**: https://geopy.readthedocs.io/en/stable/index.html (Consultato: 2025-03-02).
- **OSMnx**:
  - **Descrizione**: Libreria per scaricare e analizzare reti stradali da OpenStreetMap, utilizzata per simulare percorsi realistici;
  - **Versione**: 1.9.1;
  - **Documentazione**: https://osmnx.readthedocs.io/en/stable/ (Consultato: 2025-03-02).
- **Faker**:
  - **Descrizione**: Libreria per la generazione di dati realistici per test$_G$;

  - **Versione**: 24.1.0;
  - **Documentazione**: https://faker.readthedocs.io/en/master/ (Consultato: 2025-03-02).

- **Pylint**:
  - **Descrizione**: Strumento di analisi statica del codice Python$_G$;
  - **Versione**: 3.0.3;
  - **Documentazione**: https://pylint.pycqa.org/en/latest/index.html (Consultato: 2025-03-03).

- **Pytest**:
  - **Descrizione**: Framework$_G$ per test$_G$ automatizzati;
  - **Versione**: 7.4.3;
  - **Documentazione**: https://docs.pytest.org/en/stable/ (Consultato: 2025-03-03).

2.2.2 SQL

Linguaggio standard per l'interrogazione e la manipolazione di database$_G$ relazionali, utilizzato nel contesto di Clickhouse$_G$ per definire lo schema del database$_G$ e per interrogare i dati.

2.2.2.1 Specifiche

- **Dialetto**: Clickhouse$_G$ SQL$_G$;
- **Documentazione**: https://clickhouse.com/docs/sql-reference (Consultato: 2025-03-05).

2.2.2.2 Ruolo nel progetto

In NearYou, SQL$_G$ viene utilizzato per:

- Definizione dello schema del database$_G$;
- Interrogazione dei dati per la visualizzazione;
- Creazione di query$_G$ analitiche per l'identificazione delle relazioni spaziali.

2.2.3 Formati di interscambio dati

2.2.3.1 YAML

YAML è un formato di serializzazione dei dati human-readable basato sull'indentazione, utilizzato principalmente per file di configurazione.

2.2.3.2 Specifiche

- **Versione**: 1.2;
- **Documentazione**: https://yaml.org/spec/1.2.2/ (Consultato: 2025-03-05).

2.2.3.3 Ruolo nel progetto

- Configurazione dell'ambiente Docker$_G$;
- Workflow CI/CD;
- Configurazione dei servizi.

2.2.3.4 JSON

JSON$_G$ è un formato di interscambio dati leggero e indipendente dal linguaggio, basato su coppie chiave-valore.

2.2.3.5 Specifiche

- **Versione**: 2.0;
- **Documentazione**: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON (Consultato: 2025-03-02).

2.2.3.6 Ruolo nel progetto

- Serializzazione dei messaggi scambiati tra i componenti;
- Configurazione delle dashboard$_G$ di visualizzazione;
- Comunicazione con i servizi API$_G$ esterni.

2.3 Infrastruttura e servizi

2.3.1 Apache ZooKeeper

Servizio di coordinamento distribuito che fornisce primitive per la gestione della configurazione, la sincronizzazione e la denominazione dei nodi in sistemi distribuiti.

2.3.1.1 Specifiche

- **Versione**: 7.6.0;
- **Documentazione**: https://zookeeper.apache.org/documentation.html (Consultato: 2025-03-05).

2.3.1.2 Ruolo nel progetto

In NearYou, ZooKeeper è utilizzato per:

- Gestione dei broker$_G$ Kafka$_G$ e delle loro configurazioni;
- Monitoraggio dello stato dei nodi nel sistema$_G$ distribuito;
- Coordinamento delle operazioni distribuite tra i componenti;
- Gestione delle elezioni dei leader per le partizioni Kafka$_G$.

2.3.2 Apache Kafka

Sistema$_G$ di messaggistica distribuita in grado di gestire flussi di dati in tempo reale, caratterizzato da elevata scalabilità, affidabilità e tolleranza ai guasti.

2.3.2.1 Specifiche

- **Versione**: 7.6.0;
- **Documentazione**: https://kafka.apache.org/documentation/ (Consultato: 2025-03-05).

2.3.2.2 Ruolo nel progetto

In NearYou, Kafka$_G$ rappresenta il backbone della comunicazione tra componenti:

- Gestione del flusso di dati di posizione dagli utenti;
- Trasferimento dei messaggi pubblicitari generati;
- Garanzia di consegna delle informazioni anche in caso di guasti;
- Supporto al pattern event-driven dell'architettura.

2.3.3 Apache Flink

Framework$_G$ di elaborazione dati stream e batch distribuito, caratterizzato da bassa latenza, elevato throughput e gestione efficiente dello stato.

2.3.3.1 Specifiche

- **Versione**: 1.20.0;
- **Documentazione**: https://nightlies.apache.org/flink/flink-docs-stable/ (Consultato: 2025-03-05).

2.3.3.2 Ruolo nel progetto

In NearYou, Flink$_G$ è utilizzato per:

- Elaborazione in tempo reale dei dati di posizione;
- Calcolo della prossimità tra utenti e punti di interesse tramite clickhouse$_G$;
- Orchestrazione del processo$_G$ di generazione dei messaggi pubblicitari tramite LLM$_G$;
- Configurazione dei job per l'elaborazione dei dati.

2.3.4 ClickHouse

Database$_G$ colonnare progettato per l'analisi OLAP (OnLine Analytical Processing, tecnica che consente di interrogare ed esaminare rapidamente grandi volumi di dati da diverse prospettive) che permette agli utenti di generare report analitici utilizzando query$_G$ SQL$_G$ in tempo reale. La struttura è ottimizzata per aggregazioni e interrogazioni, consentendo operazioni complesse su dataset estesi in tempi brevissimi. Le caratteristiche principali di Clickhouse$_G$ sono:

- Architettura colonnare per interrogazioni analitiche efficienti;
- Supporto a funzioni geospaziali per calcoli di distanza;
- Supporto di dati time series;
- Integrazione nativa con Kafka$_G$ per l'ingestione di dati;
- Scalabilità orizzontale per gestire grandi volumi di dati.

2.3.4.1 Specifiche

- **Versione**: 24.10;
- **Documentazione**: https://clickhouse.com/docs/en/ (Consultato: 2025-03-05).

2.3.4.2 Ruolo nel progetto

In NearYou, Clickhouse$_G$ è utilizzato per:

- Archiviazione dei dati di posizione degli utenti;
- Rilevamento della prossimità tra utenti e punti di interesse;
- Memorizzazione delle informazioni sui punti di interesse;
- Storicizzazione dei messaggi pubblicitari generati;
- Supporto alle query$_G$ analitiche per la visualizzazione.

2.3.5 Grafana

Piattaforma open-source per la visualizzazione e il monitoraggio dei dati, con supporto per diverse fonti di dati e creazione di dashboard$_G$ interattive.

2.3.5.1 Specifiche

- **Versione**: 11.5.2;
- **Documentazione**: https://grafana.com/docs/ (Consultato: 2025-03-05).

2.3.5.2 Ruolo nel progetto

In NearYou, Grafana$_G$ è utilizzato per:

- Visualizzazione in tempo reale delle posizioni degli utenti;
- Rappresentazione dei punti di interesse sulla mappa;
- Monitoraggio dei messaggi pubblicitari generati;
- Creazione di dashboard$_G$ interattive per l'analisi dei dati.

2.3.6 Docker

Piattaforma di containerizzazione che consente di impacchettare applicazioni con le loro dipendenze in unità standardizzate chiamate container$_G$.

2.3.6.1 Specifiche

- **Versione**: 28.0.1;
- **Documentazione**: https://docs.docker.com/ (Consultato: 2025-03-05).

2.3.6.2 Ruolo nel progetto

In NearYou, Docker$_G$ è utilizzato per:

- Containerizzazione dei diversi componenti del sistema$_G$;
- Creazione di un ambiente di sviluppo e deployment coerente;
- Semplificazione della distribuzione dell'applicazione;
- Isolamento dei servizi e gestione delle dipendenze.

3 Architettura logica

La logica del progetto$_G$ adotta un approccio esagonale incentrato sugli eventi, con l'obiettivo di separare chiaramente la logica di dominio dai servizi esterni. Al centro si trova il core esagonale, che contiene le regole per la gestione del business legato alla generazione di messaggi pubblicitari personalizzati. Questo nucleo è isolato da sistemi Kafka$_G$, Clickhouse$_G$ e API$_G$ esterne, tramite porte (interface) e adattatori (infrastructure), favorendo una netta suddivisione delle responsabilità.

3.1 Pattern di architettura-esagonale

Il pattern esagonale è stato scelto per la sua capacità di disaccoppiare la logica di business dalle tecnologie specifiche. Questa separazione consente una maggiore manutenibilità e testabilità del sistema$_G$, oltre a facilitare l'evoluzione tecnologica senza impattare sul nucleo funzionale.
Nell'architettura esagonale implementata, possiamo distinguere:

- **Core domain**: Il nucleo centrale che rappresenta le entità e la logica di business del sistema$_G$. La logica di business, ha una o più porte;
- **Porte (Ports)**: Una porta definisce un set di operazioni ed il modo con il quale la logica si interfaccia con la parte esterna, solitamente sono implementate tramite un'*Interfaccia*. Si distinguono in:

. **Inbound ports**: Interfacce che espongono le funzionalità del dominio verso l'esterno;

. **Outbound ports**: Interfacce che definiscono come il dominio può utilizzare servizi esterni.

- **Adapters**: Implementazioni concrete delle porte che collegano il dominio alle tecnologie specifiche:

. **Inbound adapters**: Adattatori che convertono le richieste esterne nel formato atteso dal dominio;

. **Outbound adapters**: Adattatori che implementano le interfacce di uscita collegandole a tecnologie specifiche.

3.2 Servizi principali e loro componenti

NearYou implementa due servizi principali, entrambi progettati secondo il pattern esagonale:

- **Generatore di posizioni GPS**:

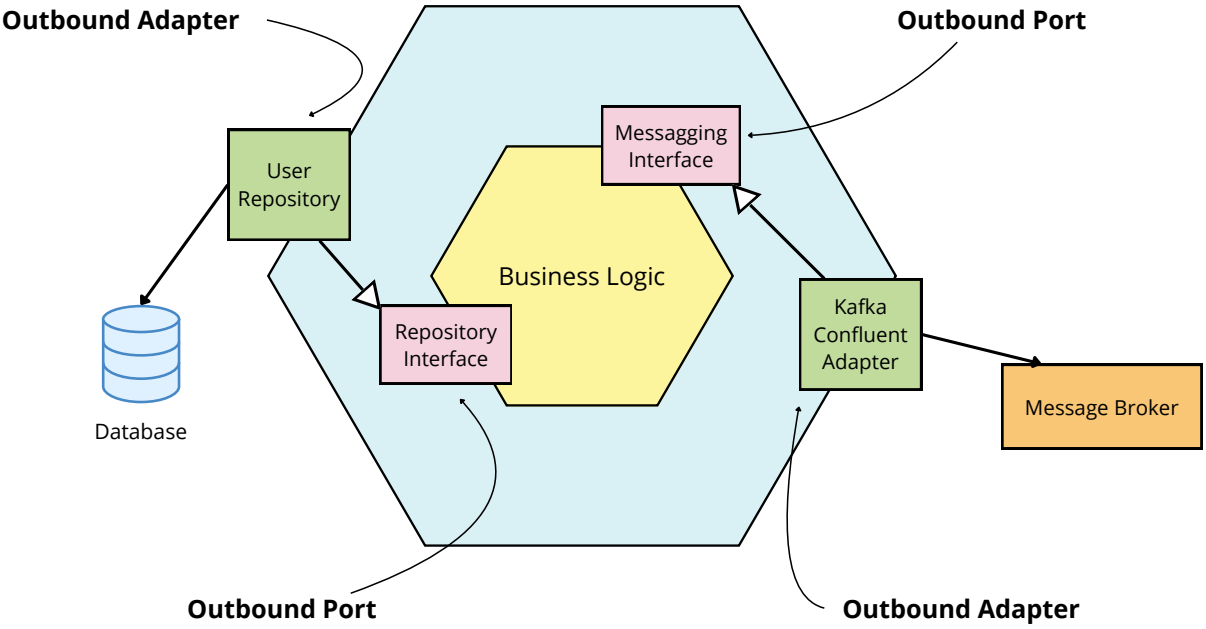

Figure 1: Architettura esagonale del SimulationService

. **Core domain**: Implementa la logica di simulazione del movimento di utenti nello spazio, sfruttando percorsi reali e dinamiche di movimento implementate da diverse strategie. Nella logica di business troviamo infatti il modo in cui avviene la creazione dei vari sensori nel sistema$_G$ in `SensorFactory` e il modo in cui vengono simulate le posizioni dei sensori in `IPositionSimulationStrategy`;

. **Inbound ports**: Non dispone di inbound ports in quanto il modulo si occupa della mera creazione di dati fittizi;

. **Outbound ports**: È necessario che il modulo si interfacci con l'esterno: con il database$_G$, per garantire una corretta associazione Sensore Simulato - Utente nel sistema$_G$, e con un'interfaccia di messaggistica, per pubblicare i dati prodotti dalla logica di business. Nel nostro sistema$_G$, le interfacce sono `IUserRepository`, `ISensorRepository` e la classe astratta `PositionSender`;

. **Adapters**: Implementa adattatori come `ClickhouseUserRepository` e `ClickhouseSensorRepository` che implementano le rispettive porte e permettono di eseguire le operazioni di accesso al database$_G$. Per quanto riguarda la scrittura dei messaggi, l'adapter associato è `KafkaConfluentAdapter` che sfrutta la tecnologia di message brokering scelta 2.3.2.

- **Elaboratore di posizioni / Generatore di messaggi**:

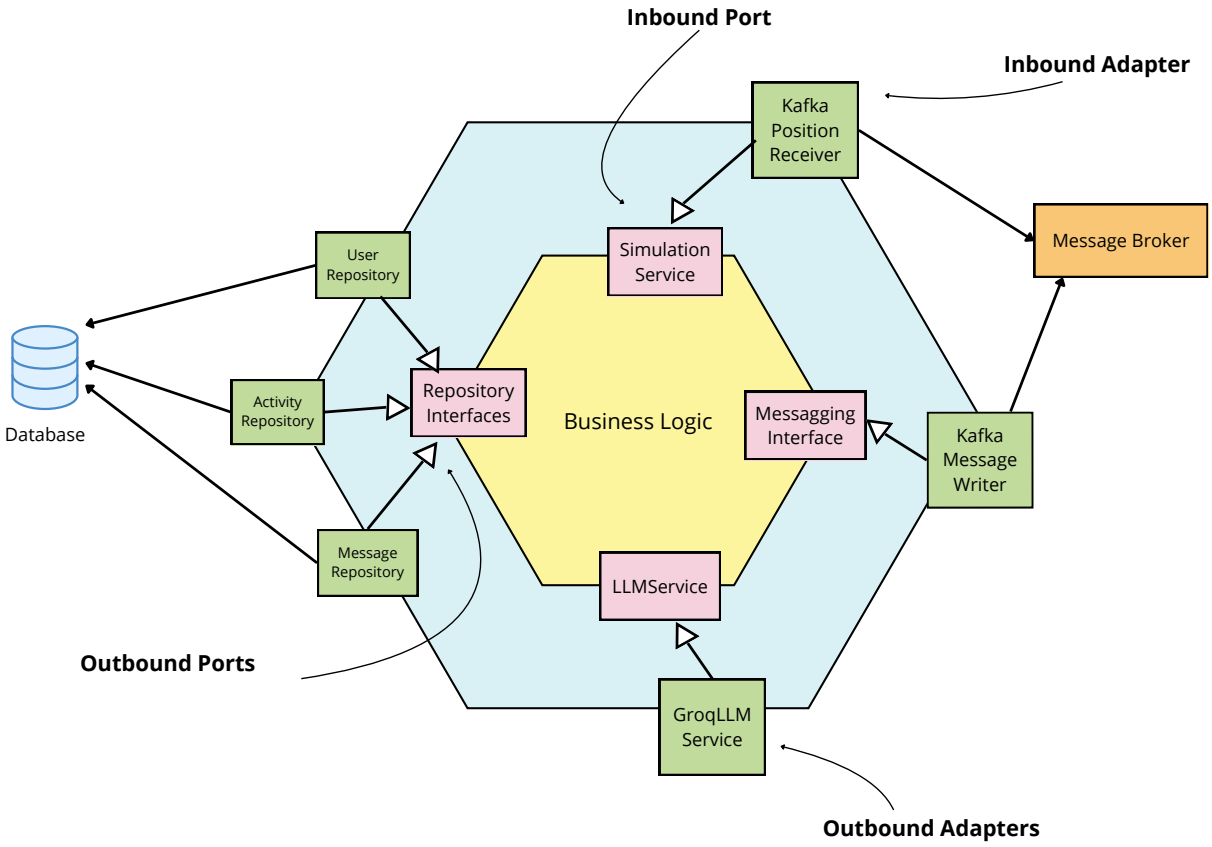

Figure 2: Architettura esagonale del PositionToMessageService

. **Core domain**: Contiene la logica di identificazione della prossimità e generazione di messaggi pubblicitari. Questa logica permette di elaborare i dati ricevuti tramite la inbound port e tramite il sistema$_G$ di stream-processing$_G$ 2.3.3 , consumandoli come datastream per PoI$_G$ effettuare operazioni di Filter e Mapping su di essi;

. **Inbound ports**: Permette di ricevere i dati che andranno PoI$_G$ elaborati, interfaccia chiamata `IPositionReceiver`;

. **Outbound ports**: Comprende interfacce verso i diversi repository$_G$ necessari alla logica (`IUserRepository`, `IActivityRepository`,`IMessageRepository`), servizi esterni necessari per l'elaborazione dei dati come (`LLMService`) e publisher di messaggi (`IMessageWriter`);

. **Adapters**: Implementa adattatori per tecnologie specifiche, come `KafkaPositionReceiver`, `ClickHouseActivityAdapter` e `GroqLLMAdapter`.

Questo approccio garantisce che ogni componente abbia responsabilità chiaramente definite e che il nucleo di business rimanga indipendente dalle tecnologie utilizzate per l'implementazione.

4 Architettura del Sistema

4.1 Panoramica architetturale

L'architettura del progetto$_G$ si basa su un insieme di microservizi event-driven che comunicano fra di loro mediante Kafka$_G$. I dati di posizione vengono raccolti in tempo reale, elaborati per verificare la prossimità dei punti d'interesse ed eventualmente sottoporli ad un servizio LLM$_G$ che genera messaggi pubblicitari personalizzati per l'utente in base al tipo della attività.

4.2 K-Architecture: Event Streaming Platform

La Kappa-architecture$_G$ è stata selezionata come architettura di riferimento per il progetto$_G$ in virtù della sua capacità di unificare l'elaborazione di dati in tempo reale e batch all'interno di un unico stack tecnologico, garantendo flessibilità, semplicità operativa e scalabilità. Il vantaggio principale è l'eliminazione della duplicazione di tecnologie e pipeline tipica della Lambda Architecture. In quest'ultima, due sistemi separati (uno per il batch e uno per lo streaming) richiedono codice, logiche e infrastrutture distinte, aumentando i costi di sviluppo,

4.2.1 Motivazioni della scelta architetturale

- **Vantaggi per l'elaborazione in tempo reale**: Tra i vantaggi per l'elaborazione in tempo reale c'è la capacità di gestire flussi di dati continui senza ritardi significativi;
- **Tecniche di riduzione della latenza**: La riduzione della latenza è garantita dalla gestione dei dati in tempo reale, senza la necessità di processi batch;
- **Benefici sul disaccoppiamento**: Tra i benefici del disaccoppiamento c'è la possibilità di sviluppare e scalare indipendentemente i componenti del sistema$_G$, garantendo una maggiore flessibilità;
- **Ottimizzazione del codebase**: La semplificazione della pipeline di dati permette di ridurre la complessità del codice e di semplificare la manutenzione.

4.2.2 Componenti principali

Il progetto$_G$ si suddivide in cinque componenti principali, ognuno dei quali svolge un ruolo specifico nell'architettura complessiva:

- Data Source;
- Straming Layer;
- Processing Layer;
- Storage Layer;
- Data Visualization.

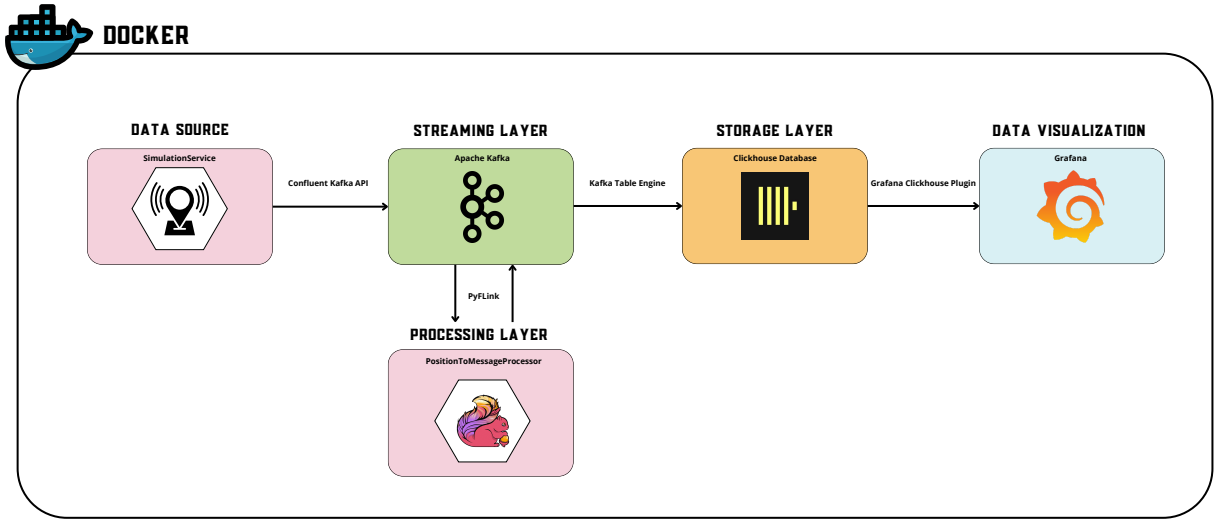

Figure 3: Diagramma dell'architettura di Sistema$_G$

Descrizione dei componenti dell'architettura:

- **Data Source**: Il ruolo di questo componente è coperto dal *SimulationService*, che si occupa della generazione delle posizioni appartenenti a percorsi realisti;
- **Streaming Layer**: Basato su Apache Kafka$_G$, gestisce la comunicazione asincrona tra i microservizi, garantendo la scalabilità e la resilienza del sistema$_G$ grazie alla gestione dei topic$_G$ e delle partizioni con le chiavi che corrispondono all'id del sensore per facilitare l'elaborazione in maniera parallela. Infine Kafka$_G$ è responsabile della storicizzazione dei log nelle apposite entry del database$_G$ Clickhouse$_G$;
- **Elaborazione dei dati**: Implementato con il *PositionToMessageService* che sfrutta le funzionalità Apache Flink$_G$, elabora i dati valutando la prossimità dei punti d'interesse e interagendo con l'LLM per generare annunci personalizzati;
- **Storage**: Supportato dal database$_G$ Clickhouse$_G$, memorizza i dati in tabelle colonnari ad alte prestazioni, consentendo query$_G$ analitiche rapide grazie all'ottimizzazione per letture intensive;
- **Visualizzazione**: Basato su Grafana$_G$, costituisce una soluzione di visualizzazione dei dati su una mappa e l'integrazione di tale interfaccia con i dati del datasource avviene tramite delle query$_G$. Sfrutta inoltre il connettore nativo di Clickhouse$_G$, permettendo l'integrazione e le query$_G$ in tempo reale delle informazioni.

4.3 Integrazione Architettura logica e Architettura di sistema

4.3.1 Descrizione

Le due architetture, la Kappa-architecture$_G$ e l'architettura esagonale, rappresentano due prospettive differenti dello stesso sistema$_G$. Mentre la Kappa-architecture$_G$ si riferisce all'implementazione concreta del codice e al flusso continuo di dati in tempo reale, l'architettura esagonale evidenzia la separazione logica tra il core di business e le interfacce esterne. Nonostante l'approccio e la terminologia differiscano, i componenti del sistema$_G$ sono gli stessi condivisi fra le due architetture e possono quindi essere mappati fra di loro. Ovviamente, il layer di Visualizzazione non rientra nell'architettura esagonale, poiché non è un componente realizzato dal gruppo, ma si interfaccia solamente con il database$_G$ Clickhouse$_G$ per la parte di interfaccia utente.

4.3.2 Mappatura dei componenti

Kappa Architecture	Architettura Esagonale	Ruolo nel Progetto
Log Immutabile	*SimulationService* Outbound Port	Il servizio che si occupa di creare le posizioni simulate, invia queste ultime tramite l'apposita outbound port, come uno stream di dati persistente, partizionato per utente e ordinato temporalmente, implementato con ApacheKafka, usando il topic$_G$ `SimulatorPosition`.
Stream-processing$_G$ Engine	*PositionToMessageService* Inbound Port/Core Logic	Il servizio riceve le posizioni tramite l'apposita Inbound Port, elabora lo stream in tempo reale, parallelamente per ogni singolo utente
Viste Materializzate	*PositionToMessageService* Outbound Port	Si occupa di prelevare i dati dello stream elaborato e storicizzarli nell'apposito database$_G$ Clickhouse$_G$ ottimizzato per rispondere alle query$_G$ dell'interfaccia grafica in maniera efficiente

Table 2: Mappatura dei componenti tra Kappa-architecture$_G$ e Architettura-esagonale$_G$

4.4 Dataflow

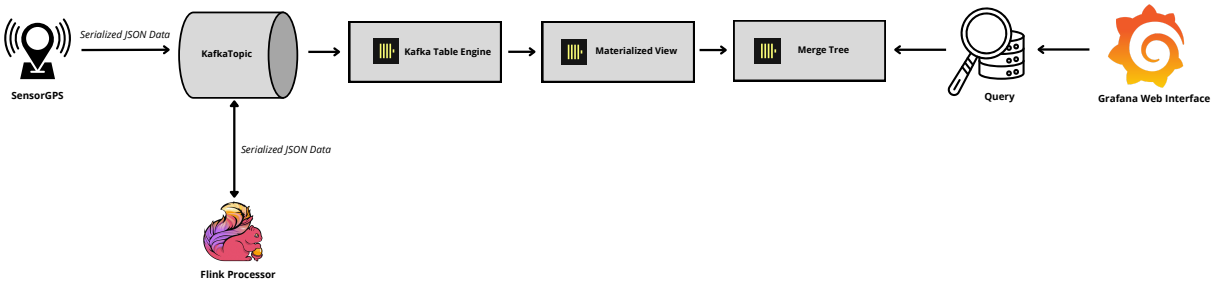

Figure 4: Flusso dei dati nell'architettura

Il flusso dei dati nell'architettura di sistema$_G$ segue un percorso$_G$ ben definito, garantendo la separazione tra la logica di business e le tecnologie di implementazione. Di seguito viene descritto il flusso di dati end-to-end:

1. **Generazione delle posizioni**:
   - Il core domain del *SimulationService* crea oggetti `GeoPosition` che rappresentano le coordinate degli utenti;
   - Questi oggetti vengono inviati all'esterno attraverso l'outbound port `PositionSender`;
   - L'adapter `KafkaConfluentAdapter` serializza i dati in formato JSON$_G$ e costruisce un istanza di un Producer che pubblicherà i dati sul topic$_G$ Kafka$_G$ `SimulatorPosition`.
2. **Consumo delle posizioni**:
   - L'inbound adapter `KafkaPositionReceiver` del servizio *PositionToMessageService* istanzia una *KafkaSource* collegata al topic$_G$ Kafka$_G$ `SimulatorPosition`;

- I payload ricevuti vengono deserializzati secondo uno schema ben definito in `JsonRowDeserializationSchema` e convertiti in oggetti di dominio `UserPosition`;
- L'istanza che implementa `IPositionReceiver` viene collegata al datastream nella logica di business del servizio di elaborazione che elaborerà le posizioni ricevute in input.

3. **Elaborazione e generazione di messaggi**:
   - Il core domain applica una funzione di *Filter*, con la classe `FilterMessageValidator`, al datastream in input per validare i dati ricevuti in input e limitare il *KafkaPoisoning* (scelta esplicata in 4.5.3.6)
   - Il core domain valuta la prossimità dell'utente rispetto ai punti di interesse, utilizzando l'outbound port `IUserRepository` per ottenere le informazioni specifiche dell'utente collegato alla posizione ricevuta in input, utilizza PoI_G `IActivityRepository` per recuperare le attività nelle vicinanze dell'utente con gli interessi condivisi;
   - In caso di rilevamento di un punto di interesse valido in prossimità, il core domain `PositionToMessageProcessor` crea un prompt per PoI_G richiedere un messaggio personalizzato tramite l'outbound port `LLMService`;
   - L'adapter `GroqLLMService` comunica con il servizio LLM_G esterno e restituisce il messaggio generato;
   - Viene applicata un'altra funzione di *Filter*, implementata in `FilterMessageAlreadyDisplayed`, per prevenire la duplicazione di messaggi generati per il singolo utente, necessario per rispettare i requisiti stabiliti;
   - Il messaggio personalizzato viene incapsulato in un oggetto `MessageDTO` del dominio per facilitarne la serializzazione.
4. **Pubblicazione del messaggio pubblicitario**:
   - L'oggetto `MessageDTO` viene passato all'outbound port `IMessageWriter`;
   - L'adapter `KafkaMessageWriter` serializza il messaggio secondo uno schema ben definito in `JsonRowSerializationSchema` e lo pubblica sul topic_G Kafka_G `MessageElaborated`.
5. **Persistenza e visualizzazione**:
   - Clickhouse_G, attraverso il connettore Kafka_G nativo, in particolare sfruttando le tecnologie *Kafka Table Engine* ⇒ *Materialized View* ⇒ *Merge Tree* consuma e archivia nella apposita tabella i messaggi dal topic_G `messageTable`;
   - Grafana_G interroga Clickhouse_G tramite l'apposito plugin e il sistema_G di query_G, per recuperare e visualizzare i dati in tempo reale attraverso dashboard_G interattive.

Il flusso dei dati è progettato per essere asincrono, garantendo la scalabilità e la resilienza del sistema_G. Ogni componente può funzionare indipendentemente, con Kafka_G che funge da buffer di messaggi affidabile tra i vari stadi del processo_G.
Gli adattatori si occupano d'interfacciarsi con l'esterno: il simulatore di posizioni produce eventi JSON_G su Kafka_G, successivamente elaborati da Flink_G per definire la prossimità ai punti di interesse e gestire i dati necessari alla logica di dominio. Qualora sia richiesta la generazione di contenuti, un LLM_G esterno crea i messaggi personalizzati che confluiscono nel dominio. La persistenza e la consultazione storica avvengono tramite Clickhouse_G, mentre Grafana_G rende immediatamente disponibili tali informazioni agli utenti.
Questo utilizzo di Kafka_G come backbone di comunicazione sostiene la natura asincrona ed event-driven del sistema_G, svincolando i componenti gli uni dagli altri. Grazie a questa separazione in porte e adattatori, l'architettura risulta flessibile, manutenibile e facile da testare: ogni modifica alla periferia può essere gestita senza impattare la logica di dominio, preservando nel contempo la coerenza e la semplicità di estensione all'intero sistema_G.

4.5 Implementazione tecnica dei componenti principali

4.5.1 DataSource - Simulation Service

Il SimulationModule è una componente architettonica progettata per simulare dati di posizionamento geografico in un ecosistema più ampio di gestione dati. Questo modulo rappresenta l'applicazione pratica dell'integrazione tra i principi della Kappa-architecture$_G$ e dell'Architettura Esagonale.
Il sistema$_G$ opera attraverso tre fasi fondamentali. Inizialmente, prepara l'ambiente di simulazione acquisendo le risorse necessarie e configurando il modello geografico. In questa parte vengono creati i sensori che sono associati uno ad uno con gli utenti gia registrati nel sistema$_G$. Successivamente, viene attivato il processo$_G$ di simulazione che genera flussi di dati rappresentanti movimenti virtuali attraverso percorsi realistici. Infine, questi dati vengono incanalati verso il sistema$_G$ di streaming centrale.
La simulazione crea un flusso continuo di eventi che rispecchia scenari di movimento reali. Questo approccio event-driven si allinea perfettamente con la filosofia Kappa, dove tutti i dati sono modellati come flussi di eventi, mentre la struttura interna rispetta i principi dell'Architettura Esagonale.

4.5.1.1 Diagramma della classi

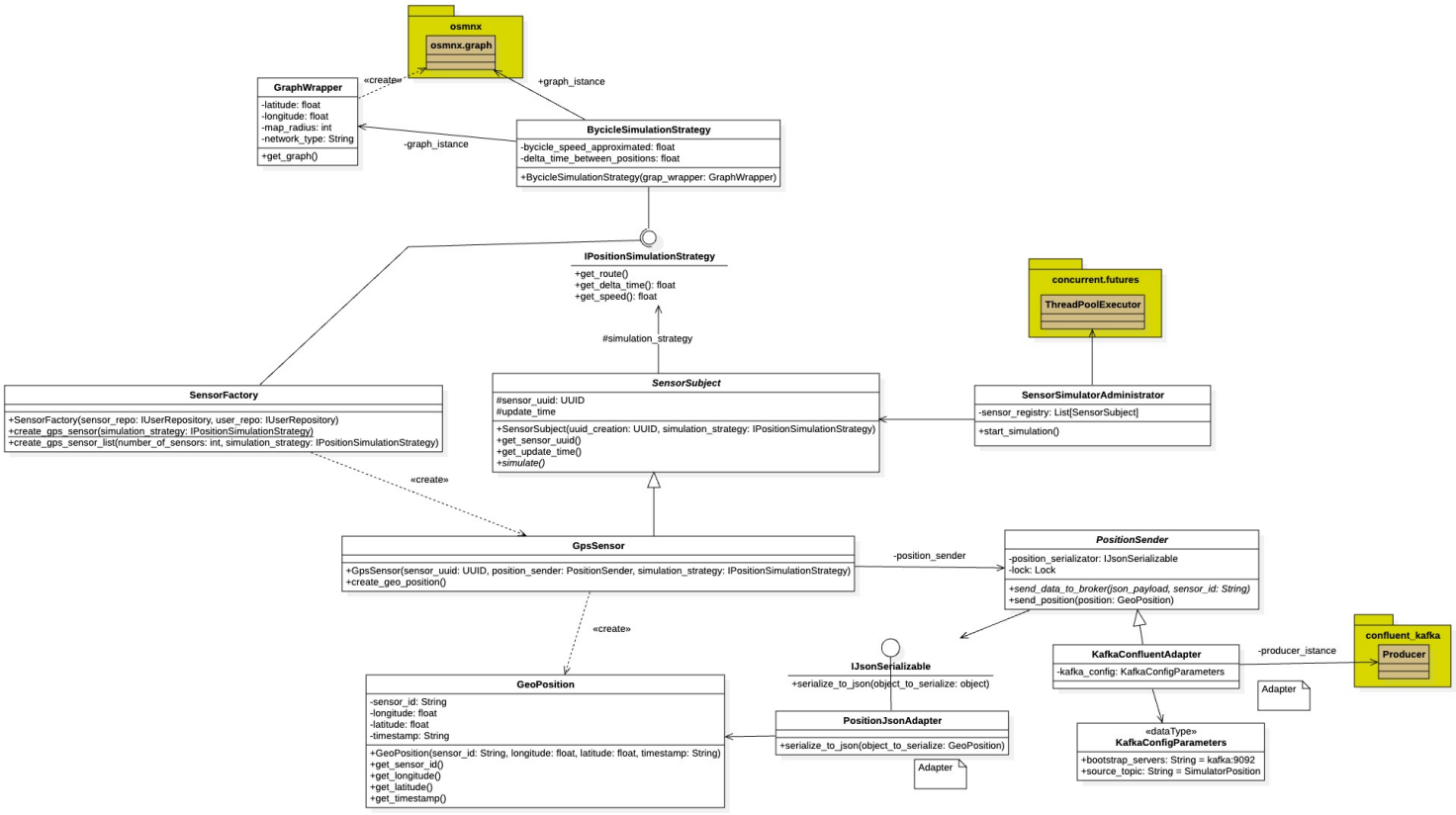

Figure 5: SimulationService Core

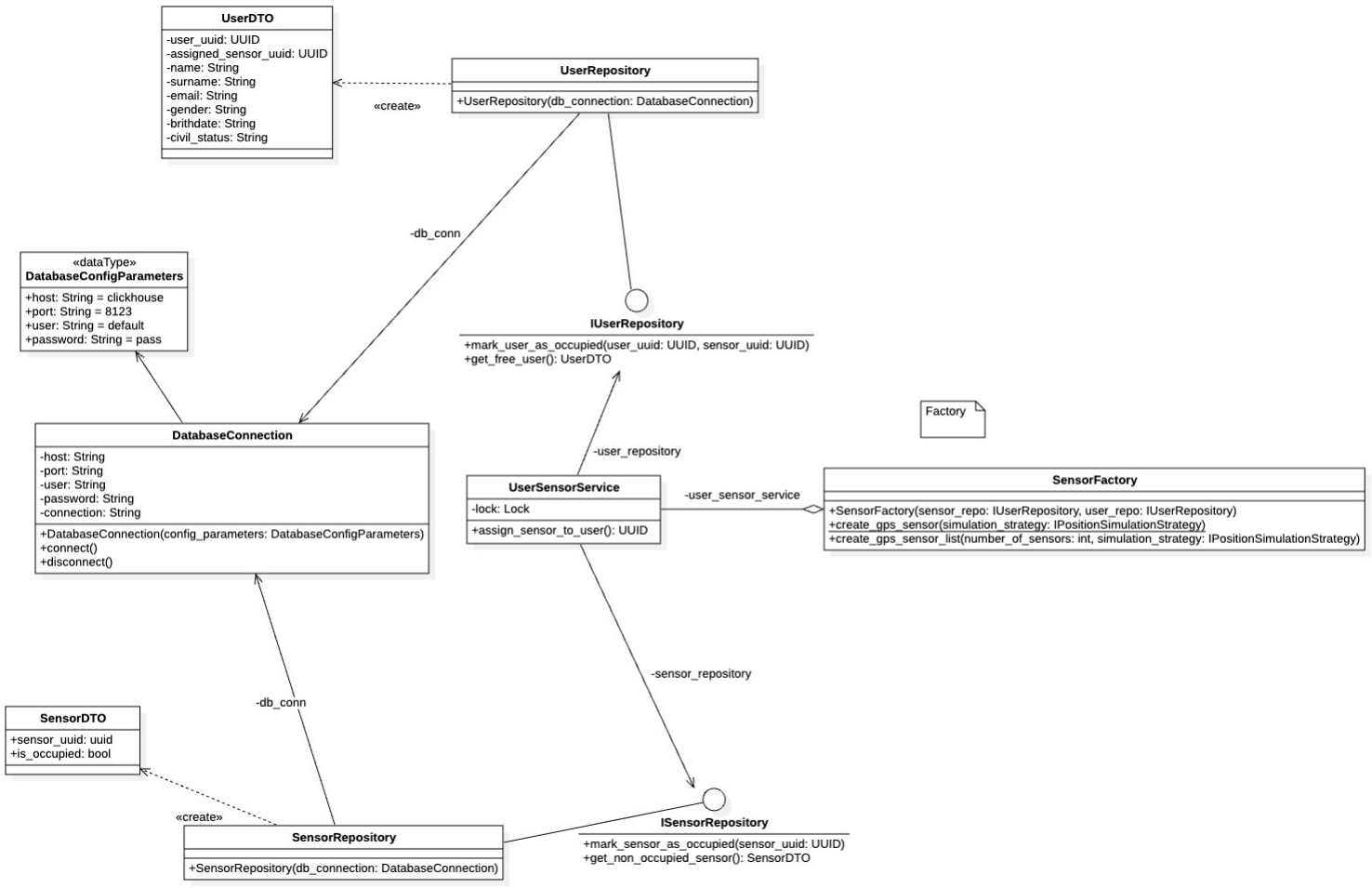

Figure 6: Factory di Sensori

4.5.1.2 Design Pattern - Strategy Pattern

4.5.1.2.1 Motivazioni e studio del design pattern

Il pattern Strategy è stato adottato per incrementare la flessibilità nella gestione di diverse modalità operative del sistema$_G$. Questa scelta architetturale permette di definire un'interfaccia comune per tutte le strategie implementate, consentendo di modificare il comportamento del sistema$_G$ selezionando una strategia specifica. Tale approccio aderisce al principio Open/Closed, agevolando l'aggiunta di nuove strategie senza intervenire su quelle esistenti.

4.5.1.2.2 Implementazione del design pattern

L'implementazione del pattern Strategy avviene tramite la creazione di:

1. Un'interfaccia che definisce i metodi comuni necessari per implementare diverse strategie per una specifica funzionalità;

2. Una o più classi concrete che implementano tale interfaccia, fornendo algoritmi specifici per realizzare la funzionalità in diversi modi.

4.5.1.2.3 Utilizzo

L'integrazione del pattern Strategy disaccoppia la logica specifica delle funzionalità dal codice client che le invoca, semplificando l'estensibilità del sistema$_G$. La possibilità di scegliere le strategie a runtime permette di adattare dinamicamente il comportamento dell'applicazione in base al contesto. Ciò si rivela

particolarmente vantaggioso in scenari che richiedono la sperimentazione o l'utilizzo di diverse modalità operative senza necessità di modifiche al nucleo del codice.

4.5.1.2.4 Integrazione del pattern

Il nostro sistema$_G$ adotta il pattern Strategy per la simulazione della posizione, definendo l'interfaccia `IPositionSimulationStrategy`. Questa scelta architetturale consente l'implementazione di diverse logiche di simulazione, garantendo flessibilità per molteplici scenari di utilizzo.

```
class IPositionSimulationStrategy(ABC):
    @abstractmethod
    def get_route(self):
        pass

    @abstractmethod
    def get_delta_time(self) -> float:
        pass

    @abstractmethod
    def get_speed(self) -> float:
        pass
```

Il funzionamento dei metodi della `IPositionSimulationStrategy` è il seguente:

- `get_route(...)`: Metodo astratto che deve restituire la sequenza di coordinate geografiche che rappresentano il percorso$_G$ simulato;
- `get_delta_time(...) -> float`: Metodo astratto che deve restituire l'intervallo di tempo (in secondi) tra due posizioni consecutive nella simulazione;
- `get_speed(...) -> float`: Metodo astratto che deve restituire la velocità (in metri al secondo) della simulazione.

```
class BycicleSimulationStrategy(IPositionSimulationStrategy):

    def __init__(self, graph_istance: GraphWrapper):
        self.__bycicle_speed_approximated = 15
        self.__delta_time_between_positions = 21
        self.__graph_istance = graph_istance.get_graph()

    def get_route(self):
        graph_returned = self.__graph_istance
        graph_nodes = list(graph_returned.nodes)
        starting_node = random.choice(graph_nodes)
        destination_node = random.choice(graph_nodes)

        shortest_route = osmnx.shortest_path(
            graph_returned,
            starting_node,
            destination_node,
            weight='length'
        )
        route_coords = [(graph_returned.nodes[node]["y"], graph_returned.nodes[node]["x"])
            for node in shortest_route]

        return route_coords

    def get_delta_time(self) -> float:
        return self.__delta_time_between_positions

    def get_speed(self) -> float:
        speed = self.__bycicle_speed_approximated / 3.6
```

```
29          return speed
```

Il funzionamento dei metodi della `BycicleSimulationStrategy` è il seguente:

- `__init__(...)`: Costruttore che inizializza la strategia di simulazione della bicicletta, ricevendo un'istanza di `GraphWrapper` (presumibilmente per accedere ai dati della mappa). Inizializza la velocità approssimativa della bicicletta e il delta temporale tra le posizioni;
- `get_route(...)`: Metodo che simula la generazione di un percorso$_G$. Utilizza la libreria OSMnx per ottenere un percorso$_G$ più breve casuale tra due nodi del grafo fornito e restituisce una lista di coordinate geografiche;
- `get_delta_time(...) -> float`: Metodo che restituisce il delta temporale predefinito per la simulazione della bicicletta;
- `get_speed(...) -> float`: Metodo che restituisce la velocità media approssimativa della bicicletta (convertita da km/h a m/s).

Attualmente, `BycicleSimulationStrategy` è la nostra unica implementazione concreta dell'interfaccia `IPositionSimulationStrategy`. Tuttavia, grazie all'adozione del pattern Strategy, il sistema$_G$ è facilmente estendibile con altre strategie di simulazione per diversi tipi di movimento (ad esempio, simulazione di un'auto, di un pedone, ecc.), semplicemente implementando nuove classi che rispettino il contratto definito da `IPositionSimulationStrategy`.

4.5.1.3 Design Pattern - Factory Pattern

4.5.1.3.1 Motivazioni e studio del design pattern

Il pattern Factory è stato introdotto nel nostro sistema$_G$ per centralizzare la creazione di oggetti di una determinata famiglia (in questo caso, i sensori). Questa scelta architetturale mira a disaccoppiare il codice client dalla necessità di conoscere e istanziare direttamente le classi concrete degli oggetti che utilizza. Delegando la responsabilità di creazione ad una factory, si ottiene una maggiore flessibilità nel processo$_G$ di istanziazione, si incapsula la logica potenzialmente complessa di creazione degli oggetti e si facilita l'introduzione di nuove varianti o tipologie di oggetti.

4.5.1.3.2 Implementazione del design pattern

Il pattern Factory viene implementato attraverso:

1. Una classe dedicata "Factory" che incapsula la logica di creazione di oggetti correlati;
2. Metodi specifici all'interno della "Factory" responsabili dell'istanza dei diversi tipi di oggetti che essa è progettata per produrre;
3. Una potenziale dipendenza da astrazioni (come interfacce o classi astratte) per la creazione delle istanze concrete;
4. La gestione, all'interno dei metodi di creazione, della logica necessaria per istanziare e configurare correttamente gli oggetti richiesti, inclusa la gestione di eventuali dipendenze o configurazioni specifiche;
5. La possibilità per la "Factory" di assicurare che gli oggetti creati rispettino determinati contratti o abbiano uno stato iniziale valido.

4.5.1.3.3 Utilizzo

L'integrazione del pattern Factory semplifica il processo$_G$ di ottenimento di oggetti per il codice client. Invece di istanziare direttamente le classi concrete degli oggetti di cui ha bisogno, il codice client interagisce con la Factory, invocando il metodo di creazione appropriato per il tipo di oggetto desiderato. Il client fornisce alla Factory eventuali parametri necessari per la creazione dell'oggetto. La Factory si occupa quindi di creare e restituire un'istanza dell'oggetto richiesto, completamente configurata e pronta per essere utilizzata. Questo approccio riduce l'accoppiamento tra il codice client e le implementazioni concrete degli oggetti, migliorando la manutenibilità e la testabilità del sistema$_G$, in quanto le dipendenze di creazione sono centralizzate e possono essere facilmente sostituite o testate isolatamente.

4.5.1.3.4 Integrazione del pattern

Il pattern Factory è implementato nel nostro sistema$_G$ attraverso la classe SensorFactory, la quale incapsula la logica di creazione degli oggetti GpsSensor. Questa classe definisce i seguenti metodi principali:

```
class SensorFactory:
    def __init__(self, sensor_repo: ISensorRepository, user_repo: IUserRepository):
    self.__user_sensor_service = UserSensorService(sensor_repo, user_repo)

    def create_gps_sensor(self, position_sender: PositionSender, simulation_strategy:
        IPositionSimulationStrategy) -> SensorSubject:
        uuid = self.__user_sensor_service.assign_sensor_to_user()
        return GpsSensor(uuid, position_sender, simulation_strategy)

    def create_gps_sensor_list(self, position_sender: PositionSender, simulation_strategy:
        IPositionSimulationStrategy, number_of_sensors: int) -> List[SensorSubject]:
        sensor_list = [self.create_gps_sensor(position_sender, simulation_strategy) for i in
            range(number_of_sensors)]
        return sensor_list
```

Il funzionamento dei metodi della `SensorFactory` è il seguente:

- __init__(...): Costruttore che inizializza la factory, ricevendo repository$_G$ per sensori e utenti per la gestione dell'assegnazione degli ID unici tramite `UserSensorService`;
- create_gps_sensor(...) -> SensorSubject: Crea una singola istanza di `GpsSensor`. Riceve un `PositionSender` e una `IPositionSimulationStrategy`, ottiene un UUID tramite `UserSensorService` e restituisce l'oggetto `GpsSensor` configurato con questi elementi;
- create_gps_sensor_list(...) -> List[SensorSubject]: Crea una lista contenente il numero specificato di istanze di `GpsSensor`, riutilizzando il metodo `create_gps_sensor(...)` per ogni elemento della lista. Richiede un `PositionSender`, una `IPositionSimulationStrategy` e il numero di sensori da creare.

In conclusione, la SensorFactory attualmente centralizza la creazione di sensori GPS, ma la sua progettazione modulare ne consente una facile estensione futura per supportare la creazione di ulteriori tipi di sensori, mantenendo la logica di istanziazione in un unico punto e rispettando il principio di singola responsabilità.

4.5.1.4 Design Pattern - Adapter Pattern

4.5.1.4.1 Motivazioni e studio del design pattern

Nel contesto della nostra architettura-esagonale$_G$, l'Adapter Pattern risulta essenziale per facilitare l'interazione tra la business logic e le componenti esterne (ad esempio, i servizi di pubblicazione su Kafka$_G$ tramite serializzazione JSON$_G$ oppure la comunicazione con il repository$_G$ di Clickhouse$_G$). Grazie a questo approccio, possiamo mantenere l'indipendenza tra i moduli interni e le librerie/framework di terze parti, riducendo i vincoli e semplificando la sostituzione futura di tali componenti senza impattare sul sistema$_G$. Questo pattern consente quindi di adattare interfacce incompatibili e promuove il riutilizzo del codice.

4.5.1.4.2 Implementazione del design pattern

L'implementazione del pattern Adapter avviene tramite la creazione di:

1. Una o più interfacce che definiscono i metodi necessari a interagire con l'architettura esagonale;
2. Una classe `adapter` concreta che implementa tali interfacce, convertendo gli oggetti e le chiamate tra il formato richiesto dalla business logic e quello utilizzato dalla componente esterna.

4.5.1.4.3 Utilizzo

L'integrazione del pattern Adapter disaccoppia la logica di interazione con librerie esterne dal codice principale, migliorando modularità ed estensibilità. Permette di sostituire le implementazioni esterne senza influenzare la business logic, grazie all'astrazione del processo$_G$ di interazione. Ciò facilita l'interoperabilità con diverse tecnologie e la manutenibilità. L'Adapter è particolarmente utile in caso di evoluzione dell'infrastruttura sottostante, poiché l'aggiornamento si limita alla sua modifica, preservando l'integrità del sistema$_G$.

4.5.1.4.4 Integrazione del pattern Adapter

Di seguito vengono mostrate diverse implementazioni concrete del pattern Adapter, ognuna focalizzata su uno specifico aspetto dell'interazione con sistemi esterni. Vedremo un Adapter per la serializzazione di oggetti `GeoPosition`, un Adapter che utilizza Kafka$_G$ per la pubblicazione di dati serializzati, e Adapter dedicati alla gestione della serializzazione e deserializzazione JSON$_G$ a livello di riga, evidenziando la versatilità di questo pattern nell'adattare diverse esigenze di integrazione all'interno dell'architettura esagonale.
Di seguito viene mostrata l'implementazione concreta dell'Adapter per la serializzazione in JSON$_G$ degli oggetti `GeoPosition`:

```
class PositionJsonAdapter(IJsonSerializable):

    def serialize_to_json(self, position_istance: GeoPosition):

        return JSON.dumps({
            'user_uuid': position_istance.get_sensor_id(),
            'latitude': float(position_istance.get_latitude()),
            'longitude': float(position_istance.get_longitude()),
            'received_at': position_istance.get_timestamp(),
        })
```

Il funzionamento dei metodi della `PositionJsonAdapter` è il seguente:

- `serialize_to_json(...)`: Metodo che prende in input un'istanza di `GeoPosition` e la serializza in una stringa JSON$_G$ contenente l'UUID del sensore, la latitudine, la longitudine e il timestamp.

Il componente di pubblicazione `KafkaConcluentAdapter`, utilizza l'Adapter per implementare i metodi previsti dalla porta Position Sender e serializzare i dati prima dell'invio a Kafka$_G$:

```
class KafkaConfluentAdapter(PositionSender):

    def __init__(self,
                kafka_config: KafkaConfigParameters,
                json_adapter_istance: "PositionJsonAdapter",
                producer_istance: Producer):
        super().__init__(json_adapter_istance)
        self.__kafka_config = kafka_config
        self.__producer = producer_istance

    def send_data_to_broker(self, json_payload, sensor_id: str):
        self.__producer.produce(self.__kafka_config.source_topic,
                                key = str(sensor_id),
                                value = json_payload.encode('utf-8'))
        self.__producer.flush()
```

Il funzionamento dei metodi della `KafkaConfluentAdapter` è il seguente:

- `__init__(...)`: Costruttore che inizializza l'Adapter, ricevendo la configurazione di Kafka$_G$, un Adapter JSON$_G$ per la serializzazione e un producer Kafka$_G$ per la gestione dell'invio dei messaggi al broker$_G$;
- `send_data_to_broker(...)`: Invia il payload JSON$_G$, precedentemente serializzato, al topic$_G$ Kafka$_G$ definito nella configurazione. Utilizza l'identificativo del sensore come chiave del messaggio e garantisce l'effettiva consegna al broker$_G$ tramite il metodo `flush()`.

L'Adapter per la deserializzazione di dati JSON$_G$ utilizza il modulo `JsonRowDeserializationSchema` per convertire i dati ricevuti nel formato interno corretto.

```
class JsonRowDeserializationAdapter:
    def __init__(self, row_type_config ):
        self.__row_type_info = row_type_config
```

Il funzionamento dei metodi della `JsonRowDeserializationAdapter` è il seguente:

- `__init__(...)`: Costruttore che inizializza l'Adapter, ricevendo la configurazione del tipo di riga JSON$_G$ da deserializzare e memorizzandola internamente per essere utilizzata.

Questo Adapter permette di gestire in modo astratto la configurazione del tipo di dati JSON$_G$ in ingresso, garantendo flessibilità nell'integrazione con altri componenti del sistema$_G$. Allo stesso modo, l'Adapter per la serializzazione utilizza il modulo `JsonRowSerializationSchema` per convertire i dati in formato JSON$_G$ prima dell'invio.

```
class JsonRowSerializationAdapter:
    def __init__(self, row_type_config_message ):
        self.__row_type_info_message = row_type_config_message

    def get_serialization_schema(self):
        return JsonRowSerializationSchema.builder()\
                                    .with_type_info(self.__row_type_info_message)\
                                    .build()

    def get_deserialization_schema(self):
        return JsonRowDeserializationSchema.builder() \
                                    .type_info(self.__row_type_info) \
                                    .build()
```

Il funzionamento dei metodi della `JsonRowSerializationAdapter` è il seguente:

- `__init__(...)`: Costruttore che inizializza l'Adapter, ricevendo la configurazione del tipo di riga JSON$_G$ da serializzare e memorizzandola internamente per definire lo schema JSON$_G$ in uscita;
- `get_serialization_schema(...)`: Restituisce un'istanza di `JsonRowSerializationSchema` configurata con le informazioni sul tipo di messaggio fornite durante l'inizializzazione dell'Adapter. Questo schema è incaricato di serializzare i dati in formato JSON$_G$;
- `get_deserialization_schema(...)`: Restituisce un'istanza di `JsonRowDeserializationSchema` configurata con le informazioni sul tipo di riga memorizzate. Questo schema è responsabile della deserializzazione dei dati JSON$_G$ in un formato utilizzabile dal sistema$_G$.

4.5.2 Classi, interfacce, metodi e attributi

4.5.2.1 SensorSimulationAdministrator

- **Descrizione**: Implementa la logica per gestire la simulazione parallela di sensori multipli utilizzando un ThreadPool. Si occupa di coordinare l'esecuzione simultanea delle simulazioni di tutti i sensori registrati e l'invio dei dati a un topic$_G$ Kafka$_G$;
- **Attributi**:
  - `__sensor_registry: List["SensorSubject"]` - Registro contenente tutti i sensori da simulare, memorizzati come una lista di oggetti SensorSubject.
- **Operazioni**:
  - `__init__(self, list_of_sensors: List["SensorSubject"])` - Costruttore che inizializza l'amministratore con una lista di sensori su cui eseguire la simulazione;
  - `start_simulation(self)` - Avvia la simulazione parallela di tutti i sensori registrati utilizzando un ThreadPool, assicurando che ogni sensore esegua il proprio metodo simulate() contemporaneamente per massimizzare l'efficienza.

4.5.2.2 SensorSubject

- **Descrizione**: Implementa una classe astratta ed è utilizzata per astrarre il sensore.
  - `_sensor_uuid: uuid` - Identificatore univoco del sensore;
  - `_simulation_strategy: IPositionSimulationStrategy` - Strategia utilizzata per simulare la posizione del sensore;
  - `_update_time: float` - Intervallo di tempo per gli aggiornamenti della simulazione.
- **Operazioni**:
  - `__init__(self, uuid_creation: uuid, simulation_strategy: "IPositionSimulationStrategy")` - Costruttore che inizializza il soggetto sensore con un UUID e una strategia di simulazione;
  - `get_sensor_uuid(self)` - Restituisce l'UUID del sensore;
  - `get_update_time(self) -> float` - Restituisce l'intervallo di tempo per gli aggiornamenti;
  - `simulate(self)` - Metodo astratto che deve essere implementato dalle sottoclassi per eseguire la simulazione dei dati del sensore.

4.5.2.3 GpsSensor

- **Descrizione**: Implementa un sensore GPS che eredita dalla classe astratta SensorSubject. Questa classe simula il movimento di un dispositivo GPS lungo un percorso$_G$ predefinito e invia le posizioni generate a un destinatario specifico;
- **Attributi**:
  - `__position_sender: PositionSender` - Componente responsabile dell'invio delle posizioni generate;
  - `__speed_mps: float` - Velocità del sensore in metri al secondo.
- **Operazioni**:
  - `__init__(self, uuid_creation: uuid, position_sender: PositionSender, simulation_strategy: IPositionSimulationStrategy)` - Costruttore che inizializza il sensore GPS con un UUID, un sender di posizione e una strategia di simulazione;
  - `simulate(self)` - Implementa il metodo astratto della classe padre. Simula il movimento del sensore calcolando posizioni intermedie tra i punti della rotta definita nella strategia di simulazione, e le invia attraverso il position sender con intervalli regolari;
  - `create_geo_position(self, latitude: float, longitude: float) -> GeoPosition` - Crea un oggetto GeoPosition con la latitudine e longitudine fornite, insieme all'UUID del sensore e al timestamp corrente.

4.5.2.4 GeoPosition

- **Descrizione**: Implementa una classe che rappresenta una posizione geografica nel mondo. Memorizza le coordinate (latitudine e longitudine), insieme all'identificatore del sensore che ha rilevato la posizione e al timestamp della rilevazione;
- **Attributi**:
  - `__sensor_id: str` - Identificatore del sensore che ha rilevato la posizione;
  - `__latitude: float` - Coordinata di latitudine della posizione;
  - `__longitude: float` - Coordinata di longitudine della posizione;
  - `__timestamp: str` - Timestamp che indica quando è stata rilevata la posizione.
- **Operazioni**:

- `__init__(self, sensor_id: str, latitude: float, longitude: float, timestamp: str)` - Costruttore che inizializza un oggetto posizione con l'ID del sensore, le coordinate geografiche e il timestamp;
- `get_sensor_id(self) -> str` - Restituisce l'identificatore del sensore come stringa;
- `get_latitude(self) -> float` - Restituisce il valore della latitudine;
- `get_longitude(self) -> float` - Restituisce il valore della longitudine;
- `get_timestamp(self) -> str` - Restituisce il timestamp della rilevazione.

4.5.2.5 IPositionSimulationStrategy

- **Descrizione**: Implementa un'interfaccia astratta che definisce il contratto per diverse strategie di simulazione della posizione, seguendo il pattern Strategy. Permette di astrarre diversi modi di generare dati di posizione per i sensori simulati;
- **Attributi**:
  - Nessun attributo definito a livello di interfaccia.
- **Operazioni**:
  - `get_route(self)` - Metodo astratto che deve essere implementato dalle sottoclassi per fornire il percorso$_G$ come sequenza di coordinate geografiche;
  - `get_delta_time(self) -> float` - Metodo astratto che deve essere implementato dalle sottoclassi per fornire l'intervallo di tempo tra aggiornamenti consecutivi della posizione;
  - `get_speed(self) -> float` - Metodo astratto che deve essere implementato dalle sottoclassi per fornire la velocità di spostamento del sensore simulato.

4.5.2.6 BycicleSimulationStrategy

- **Descrizione**: Implementa una strategia di simulazione specifica per biciclette, che estende l'interfaccia IPositionSimulationStrategy. Genera percorsi casuali su una rete stradale utilizzando dati geografici e calcola il percorso$_G$ più breve tra due punti casuali del grafo;
- **Attributi**:
  - `__bycicle_speed_approximated: float` - Velocità approssimativa della bicicletta in km/h;
  - `__delta_time_between_positions: float` - Intervallo di tempo in secondi tra posizioni consecutive;
  - `__graph_istance: Graph` - Istanza del grafo della rete stradale utilizzata per la generazione del percorso$_G$.
- **Operazioni**:
  - `__init__(self, graph_istance: GraphWrapper)` - Costruttore che inizializza la strategia con un'istanza di GraphWrapper contenente il grafo della rete stradale;
  - `get_route(self)` - Implementa il metodo dell'interfaccia per generare un percorso$_G$ casuale. Seleziona due nodi casuali dal grafo e calcola il percorso$_G$ più breve tra di essi, restituendo le coordinate geografiche dei nodi del percorso$_G$;
  - `get_delta_time(self) -> float` - Implementa il metodo dell'interfaccia per restituire l'intervallo di tempo tra gli aggiornamenti di posizione;
  - `get_speed(self) -> float` - Implementa il metodo dell'interfaccia per restituire la velocità della bicicletta convertita da km/h a m/s.

4.5.2.7 GraphWrapper

- **Descrizione**: Implementa un wrapper che nasconde i dettagli di implementazione di un grafo utilizzando la libreria OSMnx. Permette di ottenere un grafo della rete stradale basato su OpenStreetMap per una determinata posizione geografica;
- **Attributi**:
  - `__latitude: float` - Latitudine del punto centrale da cui generare il grafo;
  - `__longitude: float` - Longitudine del punto centrale da cui generare il grafo;
  - `__map_radius: int` - Raggio in metri intorno al punto centrale per definire l'estensione del grafo;
  - `__network_type: str` - Tipo di rete stradale da recuperare (es. "drive", "bike", "walk").
- **Operazioni**:
  - `__init__(self, latitude: float, longitude: float, map_radius: int, network_type: str)` - Costruttore che inizializza il wrapper con i parametri necessari per generare il grafo;
  - `get_graph(self) -> osmnx.graph` - Restituisce un grafo della rete stradale centrato sulle coordinate specificate con il raggio e il tipo di rete definiti, utilizzando la libreria OSMnx.

4.5.2.8 SensorFactory

- **Descrizione**: Implementa il pattern Factory per la creazione di sensori. Si occupa di istanziare oggetti sensore nascondendo i dettagli di implementazione e gestendo l'assegnazione degli UUID attraverso il servizio utente-sensore;
- **Attributi**:
  - `__user_sensor_service: UserSensorService` - Servizio che gestisce l'associazione tra sensori e utenti.
- **Operazioni**:
  - `__init__(self, sensor_repo: ISensorRepository, user_repo: IUserRepository)` - Costruttore che inizializza la factory con i repository$_G$ necessari per gestire sensori e utenti;
  - `create_gps_sensor(self, position_sender: PositionSender, simulation_strategy: IPositionSimulationStrategy) -> SensorSubject` - Crea un singolo sensore GPS assegnandogli un UUID tramite il servizio utente-sensore e configurandolo con il sender e la strategia di simulazione forniti;
  - `create_gps_sensor_list(self, position_sender: PositionSender, simulation_strategy: IPositionSimulationStrategy, number_of_sensors: int) -> List[SensorSubject]` - Crea una lista di sensori GPS del numero specificato, utilizzando lo stesso sender e la stessa strategia di simulazione per tutti.

4.5.2.9 UserSensorService

- **Descrizione**: Implementa un servizio che gestisce l'associazione tra sensori e utenti. Si occupa di assegnare sensori disponibili a utenti liberi, garantendo l'atomicità delle operazioni attraverso un meccanismo di lock per prevenire race condition in ambienti multi-thread;
- **Attributi**:
  - `__SensorRepository: ISensorRepository` - Repository$_G$ per l'accesso e la gestione dei dati relativi ai sensori;
  - `__UserRepository: IUserRepository` - Repository$_G$ per l'accesso e la gestione dei dati relativi agli utenti;
  - `__lock: threading.Lock` - Oggetto lock utilizzato per garantire l'accesso thread-safe durante le operazioni di assegnazione sensore-utente.

- **Operazioni**:
  - `__init__(self, sensor_repository: ISensorRepository, user_repository: IUserRepository)` - Costruttore che inizializza il servizio con i repository$_G$ necessari per gestire sensori e utenti;
  - `assign_sensor_to_user(self) -> uuid` - Assegna un sensore disponibile ad un utente libero. Utilizza un lock per garantire che l'operazione sia thread-safe. Recupera un sensore non occupato e un utente libero dai rispettivi repository$_G$, marca il sensore come occupato e lo associa all'utente. Registra dettagliatamente tutte le operazioni in un file di log. Restituisce l'UUID del sensore assegnato o None se l'assegnazione fallisce.

4.5.2.10 IUserRepository

- **Descrizione**: Implementa un'interfaccia astratta che definisce il contratto per i repository$_G$ di gestione degli utenti. Stabilisce i metodi necessari per gestire lo stato di occupazione degli utenti e la ricerca di utenti disponibili;
- **Attributi**:
  - Nessun attributo definito a livello di interfaccia.
- **Operazioni**:
  - `mark_user_as_occupied(self, user_uuid: uuid.UUID, sensor_uuid: uuid.UUID)` - Metodo astratto che deve essere implementato dalle sottoclassi per marcare un utente come occupato e associarlo a un sensore specifico tramite i rispettivi UUID;
  - `get_free_user(self) -> UserDTO` - Metodo astratto che deve essere implementato dalle sottoclassi per recuperare un utente non assegnato a nessun sensore. Restituisce un oggetto UserDTO rappresentante l'utente libero, o None se non ci sono utenti disponibili.

4.5.2.11 UserRepository

- **Descrizione**: Implementa la classe concreta che realizza l'interfaccia IUserRepository per la gestione degli utenti nel database$_G$. Fornisce l'accesso ai dati degli utenti e le operazioni per modificare il loro stato di assegnazione ai sensori;
- **Attributi**:
  - `__db_conn: DatabaseConnection` - Connessione al database$_G$ utilizzata per eseguire query$_G$ sui dati degli utenti.
- **Operazioni**:
  - `__init__(self, db_connection: DatabaseConnection)` - Costruttore che inizializza il repository$_G$ con una connessione al database$_G$;
  - `mark_user_as_occupied(self, user_uuid: uuid.UUID, sensor_uuid: uuid.UUID)` - Implementa il metodo dell'interfaccia per assegnare un sensore a un utente nel database$_G$. Esegue una query$_G$ SQL$_G$ che aggiorna il campo assigned_sensor_uuid con l'UUID del sensore specificato per l'utente con l'UUID indicato;
  - `get_free_user(self) -> UserDTO` - Implementa il metodo dell'interfaccia per recuperare un utente non assegnato ad alcun sensore. Esegue una query$_G$ SQL$_G$ che seleziona il primo utente con assigned_sensor_uuid impostato a NULL e restituisce un oggetto UserDTO contenente tutte le informazioni dell'utente. Se non viene trovato alcun utente disponibile, restituisce None.

4.5.2.12 UserDTO

- **Descrizione**: Implementa un oggetto di trasferimento dati (Data Transfer Object) per la classe User. Viene utilizzato per astrarre e incapsulare i dati degli utenti, facilitando il trasferimento delle informazioni tra i diversi strati dell'applicazione senza esporre i dettagli implementativi;
- **Attributi**:

- `user_uuid: uuid` - Identificatore univoco dell'utente;
- `assigned_sensor_uuid: uuid` - Identificatore univoco del sensore assegnato all'utente, può essere None se nessun sensore è assegnato;
- `name: str` - Nome dell'utente;
- `surname: str` - Cognome dell'utente;
- `email: str` - Indirizzo email dell'utente;
- `gender: str` - Genere dell'utente;
- `birthdate: str` - Data di nascita dell'utente;
- `civil_status: str` - Stato civile dell'utente.

- **Operazioni**:
  - `__init__(self, user_uuid: uuid, assigned_sensor_uuid: uuid, name: str, surname: str, email: str, gender: str, birthdate: str, civil_status: str)` - Costruttore che inizializza l'oggetto DTO con tutti i dati dell'utente.

4.5.2.13 ISensorRepository

- **Descrizione**: Implementa un'interfaccia astratta che definisce il contratto per i repository$_G$ di gestione dei sensori. Stabilisce i metodi necessari per gestire lo stato di occupazione dei sensori e la ricerca di sensori disponibili nel sistema$_G$;
- **Attributi**:
  - Nessun attributo definito a livello di interfaccia.
- **Operazioni**:
  - `mark_sensor_as_occupied(self, sensor_uuid: uuid.UUID)` - Metodo astratto che deve essere implementato dalle sottoclassi per marcare un sensore come occupato, utilizzando il suo UUID come identificatore;
  - `get_non_occupied_sensor(self) -> SensorDTO` - Metodo astratto che deve essere implementato dalle sottoclassi per recuperare un sensore non occupato dal repository$_G$. Restituisce un oggetto SensorDTO rappresentante il sensore disponibile, o None se non ci sono sensori liberi.

4.5.2.14 SensorRepository

- **Descrizione**: Implementa la classe concreta che realizza l'interfaccia ISensorRepository per la gestione dei sensori nel database$_G$. Fornisce l'accesso ai dati dei sensori e le operazioni per modificarne lo stato di occupazione;
- **Attributi**:
  - `__db_conn: DatabaseConnection` - Connessione al database$_G$ utilizzata per eseguire query$_G$ sui dati dei sensori.
- **Operazioni**:
  - `__init__(self, db_connection: DatabaseConnection)` - Costruttore che inizializza il repository$_G$ con una connessione al database$_G$;
  - `mark_sensor_as_occupied(self, sensor_uuid: uuid.UUID)` - Implementa il metodo dell'interfaccia per marcare un sensore come occupato nel database$_G$. Esegue una query$_G$ SQL$_G$ che aggiorna il campo is_occupied a true per il sensore con l'UUID specificato;
  - `get_non_occupied_sensor(self) -> SensorDTO` - Implementa il metodo dell'interfaccia per recuperare un sensore non occupato dal database$_G$. Esegue una query$_G$ SQL$_G$ che seleziona il primo sensore con is_occupied impostato a 0 e restituisce un oggetto SensorDTO contenente l'UUID del sensore e il suo stato di occupazione. Se non viene trovato alcun sensore disponibile, restituisce None.

4.5.2.15 SensorDTO

- **Descrizione**: Implementa un oggetto di trasferimento dati (Data Transfer Object) per la classe Sensor. Viene utilizzato per astrarre e incapsulare i dati dei sensori, facilitando il trasferimento delle informazioni tra i diversi strati dell'applicazione senza esporre i dettagli implementativi;
- **Attributi**:
  - `sensor_uuid: uuid` - Identificatore univoco del sensore;
  - `is_occupied: bool` - Flag che indica se il sensore è attualmente occupato (assegnato a un utente) o disponibile.
- **Operazioni**:
  - `__init__(self, sensor_uuid: uuid, is_occupied: bool)` - Costruttore che inizializza l'oggetto DTO con l'UUID del sensore e il suo stato di occupazione.

4.5.2.16 DatabaseConnection

- **Descrizione**: Implementa una classe che gestisce la connessione al database$_G$ Clickhouse$_G$. Fornisce metodi per stabilire e chiudere connessioni al database$_G$, incapsulando i dettagli di configurazione e gestione della connessione;
- **Attributi**:
  - `host: str` - Indirizzo del server Clickhouse$_G$;
  - `port: int` - Porta sulla quale il server Clickhouse$_G$ accetta connessioni;
  - `user: str` - Nome utente per l'autenticazione al database$_G$;
  - `password: str` - Password per l'autenticazione al database$_G$;
  - `connection` - Oggetto connessione al database$_G$ Clickhouse$_G$, inizialmente impostato a None.
- **Operazioni**:
  - `__init__(self, config_parameters: DatabaseConfigParameters)` - Costruttore che inizializza l'oggetto connessione con i parametri di configurazione del database$_G$ forniti;
  - `connect(self)` - Stabilisce una connessione al database$_G$ Clickhouse$_G$ utilizzando i parametri configurati e restituisce l'oggetto client di connessione;
  - `disconnect(self)` - Chiude la connessione al database$_G$ se attiva e reimposta l'attributo connection a None.

4.5.2.17 DatabaseConfigParameters

- **Descrizione**: Implementa una classe di dati (dataclass) che contiene i parametri di configurazione necessari per la connessione a un database$_G$ Clickhouse$_G$. Fornisce una struttura semplice per incapsulare e trasportare le impostazioni di configurazione del database$_G$ in modo tipizzato;
- **Attributi**:
  - `host: str` - Indirizzo del server Clickhouse$_G$. Il valore predefinito è "clickhouse";
  - `port: str` - Porta sulla quale il server Clickhouse$_G$ accetta connessioni. Il valore predefinito è "8123";
  - `user: str` - Nome utente per l'autenticazione al database$_G$. Il valore predefinito è "default";
  - `password: str` - Password per l'autenticazione al database$_G$. Il valore predefinito è "pass".
- **Operazioni**:
  - Nessuna operazione esplicita definita, in quanto si tratta di una dataclass che fornisce automaticamente costruttore, rappresentazione in stringa, confronto e altre funzionalità.

4.5.2.18 IJsonSerializable

- **Descrizione**: Implementa un'interfaccia astratta che definisce il contratto per le classi che devono essere serializzabili in formato $JSON_G$. Fornisce un metodo standard per la serializzazione di oggetti;
- **Attributi**:
  - Nessun attributo definito a livello di interfaccia.
- **Operazioni**:
  - `serialize_to_json(self, object_to_serialize: object)` - Metodo astratto che deve essere implementato dalle sottoclassi per serializzare un oggetto in formato $JSON_G$.

4.5.2.19 PositionJsonAdapter

- **Descrizione**: Implementa un adattatore che converte oggetti GeoPosition in formato $JSON_G$, seguendo il pattern Adapter. Realizza l'interfaccia IJsonSerializable per fornire una serializzazione standardizzata degli oggetti posizione;
- **Attributi**:
  - Nessun attributo specificato nella classe.
- **Operazioni**:
  - `serialize_to_json(self, object_to_serialize: GeoPosition)` - Implementa il metodo dell'interfaccia IJsonSerializable. Converte un oggetto GeoPosition in una stringa $JSON_G$ contenente l'UUID dell'utente (derivato dall'ID del sensore), le coordinate geografiche (latitudine e longitudine) e il timestamp della rilevazione.

4.5.2.20 PositionSender

- **Descrizione**: Implementa una classe astratta che funge da componente per l'invio di posizioni geografiche a un $broker_G$ di messaggi. Progettata per essere estesa da adattatori specifici come KafkaConfluentAdapter, gestisce la serializzazione dei dati di posizione e fornisce un meccanismo thread-safe per l'invio;
- **Attributi**:
  - `__position_serializator: IJsonSerializable` - Componente che si occupa della serializzazione degli oggetti GeoPosition in formato $JSON_G$;
  - `_lock: threading.Lock` - Meccanismo di lock per garantire l'accesso thread-safe alle risorse condivise durante l'invio dei dati.
- **Operazioni**:
  - `__init__(self, json_adapter_istance: IJsonSerializable)` - Costruttore che inizializza il sender con un adattatore $JSON_G$ per la serializzazione dei dati di posizione;
  - `send_data_to_broker(self, json_payload, sensor_id: str)` - Metodo astratto che deve essere implementato dalle sottoclassi per inviare i dati serializzati al $broker_G$ di messaggi specifico;
  - `send_position(self, position: GeoPosition)` - Metodo pubblico che gestisce il $processo_G$ di invio di una posizione. Serializza l'oggetto GeoPosition utilizzando l'adattatore $JSON_G$ e invoca il metodo astratto send_data_to_broker in modo thread-safe utilizzando un lock.

4.5.2.21 KafkaConfluentAdapter

- **Descrizione**: Implementa un adattatore concreto che estende PositionSender per inviare dati di posizione a un cluster Kafka$_G$ utilizzando la libreria Confluent. Realizza l'interfaccia di invio astratta fornendo un'implementazione specifica per il broker$_G$ Kafka$_G$;
- **Attributi**:
  - `__kafka_config: KafkaConfigParameters` - Parametri di configurazione per la connessione a Kafka$_G$, inclusi il topic$_G$ di destinazione e altre impostazioni specifiche;
  - `__producer: Producer` - Istanza del producer Confluent Kafka$_G$ utilizzato per inviare i messaggi al cluster Kafka$_G$.
- **Operazioni**:
  - `__init__(self, kafka_config: KafkaConfigParameters, JSON_adapter_istance: "PositionJsonAdapter", producer_istance: Producer)` - Costruttore che inizializza l'adattatore con i parametri di configurazione Kafka$_G$, un adattatore JSON$_G$ per la serializzazione e un'istanza del producer Confluent;
  - `send_data_to_broker(self, json_payload, sensor_id: str)` - Implementa il metodo astratto della classe base. Invia il payload JSON$_G$ al topic$_G$ Kafka$_G$ specificato nella configurazione, utilizzando l'UUID del sensore come chiave del messaggio e il payload serializzato come valore. Dopo l'invio, esegue un flush per garantire che il messaggio venga consegnato al broker$_G$.

4.5.2.22 KafkaConfigParameters

- **Descrizione**: Implementa una classe di dati (dataclass) che contiene i parametri di configurazione necessari per la connessione a un cluster Kafka$_G$. Fornisce una struttura semplice per incapsulare e trasportare le impostazioni di configurazione Kafka$_G$ in modo tipizzato;
- **Attributi**:
  - `bootstrap_servers: str` - L'indirizzo e la porta dei server bootstrap Kafka$_G$ a cui connettersi. Il valore predefinito è "kafka:9092";
  - `source_topic: str` - Il nome del topic$_G$ Kafka$_G$ su cui pubblicare i messaggi. Il valore predefinito è "SimulatorPosition".
- **Operazioni**:
  - Nessuna operazione esplicita definita, in quanto si tratta di una dataclass che fornisce automaticamente costruttore, rappresentazione in stringa, confronto e altre funzionalità.

4.5.3 Streaming Layer - Apache Kafka

4.5.3.1 Topic e partitioning

In questo progetto$_G$ si utilizzano due topic$_G$:

- `SimulatorPosition`, per pubblicare i dati generati dai sensori (simulator);
- `MessageElaborated`, per pubblicare gli annunci generati dall'LLM.

4.5.3.2 Producer e Consumer

4.5.3.2.1 Message keys

Le chiavi dei messaggi (key) determinano la partizione Kafka$_G$ a cui viene inviato ogni evento, bilanciando il carico tra i consumer. Una chiave può essere definita in base a uno o più campi del messaggio, ad esempio l'ID del sensore per i record di posizione. Inoltre, è fondamentale considerare come la scelta delle chiavi possa influenzare la distribuzione dei dati e le performance del sistema$_G$.

```
    key_type = Types.ROW_NAMED(['sensor_uuid'], [Types.STRING()])
```

4.5.3.3 Integrazione con Flink keyed stream

All'interno del job Flink$_G$, l'utilizzo della chiave su ogni record consente di creare un keyed stream in cui i dati, prima di essere elaborati, vengono raggruppati in base alla loro chiave. Questo permette di gestire le funzioni di stato in modo isolato per ogni chiave e di applicare trasformazioni o filtri specifici, migliorando l'efficacia del processing e riducendo i conflitti di stato tra utenti o sensori diversi.

4.5.3.4 Schema topic simulator position

I dati inviati dal producer sul topic$_G$ `SimulatorPosition` seguono questa struttura JSON$_G$:

```
    {
        "user_uuid": "UUID",
        "latitude": "Float64",
        "longitude": "Float64",
        "received_at": "String"
    }
```

4.5.3.5 Schema message elaborated

I messaggi sul topic$_G$ `messaggi` hanno il seguente formato:

```
    {
        "user_id": "UUID",
        "activity_id": "UUID",
        "message_id": "UUID",
        "message_text": "String",
        "activity_lat": "Float64",
        "activity_lon": "Float64",
        "creation_time": "String",
        "user_lat": "Float64",
        "user_lon": "Float64"
    }
```

4.5.3.6 Kafka poisoning

- **Descrizione del problema**
  Il sistema$_G$ di stream-processing$_G$ Kafka$_G$ risulta potenzialmente vulnerabile ad un attaccante che inserisca dati falsi o malformati al fine di alterare il comportamento del sistema$_G$, pertanto è necessario applicare delle strategie di mitigazione che verifichino origine e correttezza dei dati e limitino i potenziali danni;
- **Soluzioni**
  Alcune delle possibili soluzioni per la mitigazione di questa tipologia di attacchi sono le seguenti:
  - Validazione dei dati a livello di codice;
  - Uso del protocollo$_G$ TLS per la comunicazione sensori-sistema;
  - Autenticazione sensori mediante SASL;
  - Definizione di policies di access control.
- **Strategie di mitigazione in Dettaglio**
  - Validazione dei dati a livello di codice
    - **Descrizione**:
      La validazione dei dati a livello di codice consiste nel controllo del dato, ovvero quando si prelevano i dati dal topic$_G$ potrebbe capitare che siano dei dati malformati o malevoli. Facendo questo in modo mirato sul singolo dato, è possibile garantire che ogni informazione elaborata sia conforme agli standard attesi.
      Ad esempio se sappiamo per certo che una persona si muove tra i 3 e i 6 km/h, possiamo scartare i dati che superano questa soglia;
    - **Requisiti implementazione**:
      Sarà necessario implementare dei controlli quando si prelevano i dati dal topic$_G$ così da garantire che i dati al loro interno siano entro un range di valori ammissibili, questo dovrebbe garantire la validità del dato.
  - Uso del protocollo$_G$ TLS per la comunicazione sensori-sistema
    - **Descrizione**:
      Il protocollo$_G$ TLS fornisce una modalità di comunicazione tra client e server protetta da cifratura in grado di autenticare il server e garantire l'integrità e riservatezza dei dati in transito. Il protocollo$_G$ utilizza una chiave di cifratura asimmetrica certificata per stabilire la comunicazione iniziale per PoI$_G$ utilizzare cifratura simmetrica per il resto della sessione. Apache Kafka$_G$ dispone inoltre della possibilità di applicare 2-way TLS per introdurre un'ulteriore autenticazione del client;
    - **Requisiti implementazione**:
      Il protocollo$_G$ TLS è già implementato all'interno di Apache Kafka$_G$ per abilitarlo è però necessario inserire i certificati richiesti. È possibile adottare sia certificati interni che certificati garantiti da una Certification Authority.
  - Autenticazione sensori mediante SASL
    - **Descrizione**:
      Il protocollo$_G$ SASL fornisce la possibilità di integrare un ampio spettro di metodologie per l'autenticazione di messaggi in ingresso basata su sfide e risposte e può anche essere integrato con protocolli di trasporto che garantiscano riservatezza del messaggio;
    - **Requisiti implementazione**:
      Il protocollo$_G$ SASL è già implementato all'interno di Apache Kafka$_G$ ed è necessario abilitarlo configurando i meccanismi di autenticazione desiderati (come PLAIN, SCRAM o GSSAPI/Kerberos). È PoI$_G$ richiesta la configurazione dei parametri di autenticazione sia lato client che server, definendo credenziali e ruoli degli utenti nel cluster Kafka$_G$. Questa opzione offrirebbe una soluzione di sicurezza robusta per autenticare i sensori e garantire l'integrità dei dati senza la necessità di contratti.
  - Policies di access control
    - **Descrizione**:
      L'uso di access control lists permette di definire un insieme di regole volto a limitare la possibilità che un client compromesso abbia accesso ad informazioni sensibili o sia in grado di manomettere il sistema$_G$. Ogni regola definisce per un client o gruppo di client se questi sia autorizzato o meno a produrre o consumare elementi di un topic$_G$;

  - **Requisiti implementazione**:
    Apache Kafka$_G$ dispone di un sistema$_G$ integrato di gestione dei permessi che facilita l'implementazione di policy di sicurezza. È però necessario definire opportunamente le policies desiderate nel file di configurazione, assegnando i permessi specifici per ogni componente del sistema$_G$. Ad esempio, è possibile limitare i permessi di scrittura dei simulatori al solo topic$_G$ dei dati dei sensori, mentre il modulo di elaborazione potrà avere solo permessi di lettura su quel topic$_G$. La configurazione delle policies viene gestita tramite file ACL (Access Control List) che specificano dettagliatamente i permessi di ogni componente del sistema$_G$.

- **Conclusioni**
  La validazione dei dati a livello di codice è stata adottata per mitigare il rischio$_G$ di *Kafka poisoning*, in quanto non erano richieste misure di sicurezza avanzate dal proponente$_G$. Pur offrendo una protezione inferiore rispetto a soluzioni più sofisticate, questa strategia risulta semplice da integrare e fornisce una prima linea di difesa contro possibili iniezioni di dati malevoli, senza comportare carichi di lavoro eccessivi sul resto del progetto$_G$. È stato condotto comunque uno studio approfondito delle altre tecniche di mitigazione, così da valutare il carico di lavoro richiesto e quale fosse la scelta più adatta, lasciando comunque possibilità di implementarle in futuro per migliorare la sicurezza del sistema$_G$.

4.5.4 Processing Layer - PositionToMessageProcessor

4.5.4.1 Apache Flink

Nell'architettura NearYou, Flink$_G$ gestisce un job di streaming strutturato sfruttando le funzionalità offerte della DataStream API$_G$, approccio scelto per la sua flessibilità e per la ricchezza di operatori disponibili per la manipolazione dei flussi di dati. Il flusso di elaborazione è organizzato attraverso un componente centrale, il FlinkJobManager, che coordina l'intero ciclo-di-vita$_G$ del processing dei dati. Questo manager riceve i dati di posizione dagli utenti tramite il simulation module, li elabora attraverso una serie di trasformazioni, e infine produce messaggi pubblicitari personalizzati.
Il job è progettato secondo principi di modularità e dependency injection, con componenti intercambiabili che seguono interfacce ben definite. L'elaborazione avviene in diverse fasi sequenziali:

- Ricezione di eventi di posizione attraverso un source connector Kafka$_G$;
- Raggruppamento (key-by) per identificatore utente;
- Validazione dei dati in input;
- Applicazione di funzioni di mapping per la generazione dei messaggi;
- Filtraggio dei messaggi già visualizzati;
- Pubblicazione dei risultati su un topic$_G$ Kafka$_G$ di output.

La configurazione del job è ottimizzata per l'elaborazione in tempo reale con un livello di parallelismo adeguato al carico di lavoro previsto, impostato attraverso i parametri di configurazione dell'ambiente di esecuzione Flink$_G$. L'utilizzo della DataStream API$_G$ permette inoltre di definire operazioni di trasformazione in modo dichiarativo, aumentando la leggibilità del codice e facilitando la manutenzione.

4.5.4.1.1 Elaborazione dati e pattern di progettazione

Il cuore dell'elaborazione dati in Flink$_G$ è costituito dal pattern di trasformazione dello stream attraverso funzioni di mapping e filtraggio. Il componente principale di questa elaborazione è il PositionToMessageProcessor, che implementa un pattern di design funzionale per trasformare i dati di posizione in messaggi pubblicitari contestuali.
Questo processore integra diverse fonti di dati e servizi:

- Repository$_G$ di utenti per recuperare informazioni demografiche e preferenze;
- Repository$_G$ di attività per individuare punti di interesse nelle vicinanze;
- Repository$_G$ di messaggi per individuare eventuali messaggi già generati per la coppia utente-attività;

- Servizio LLM_G per generare testi pubblicitari personalizzati.

Un aspetto importante dell'elaborazione è il meccanismo di filtering, implementato attraverso il componente FilterMessageAlreadyDisplayed. Questa logica evita di inviare ripetutamente lo stesso messaggio quando l'utente rimane fermo o si muove minimamente, ottimizzando così sia l'esperienza utente che il consumo di risorse del sistema_G.
Il pattern di progettazione adottato consente una chiara separazione delle responsabilità: la logica di business è incapsulata nel processore, mentre l'infrastruttura di comunicazione è gestita dal job manager. Questo approccio facilita la manutenzione e l'evoluzione del sistema_G.

4.5.4.1.2 Integrazione con componenti esterni

Flink_G funge da elemento integratore tra i vari componenti dell'architettura NearYou, coordinando il flusso dei dati attraverso connettori specializzati:

- **Integrazione con Kafka_G**: Attraverso i connettori KafkaPositionReceiver e KafkaMessageWriter, Flink_G legge le posizioni degli utenti dal topic_G "SimulatorPosition" e pubblica i messaggi elaborati sul topic_G "MessageElaborated". La DataStream API_G fornisce connettori nativi per Kafka_G che semplificano questa integrazione, garantendo la consistenza dei tipi di dati e la corretta gestione delle configurazioni;
- **Interazione con Clickhouse_G**: Il rilevamento di prossimità ai punti di interesse nel raggio di generazione non avviene direttamente in Flink_G, bensì delegata a Clickhouse_G attraverso query_G geospaziali ottimizzate che sfruttano la funzione nativa geoDistance. Questo approccio sfrutta le capacità di calcolo geospaziale già presenti nel database_G, ottimizzando così le performance del sistema_G;
- **Comunicazione con servizi LLM_G**: Flink_G orchestra l'interazione con il servizio Groq_G per la generazione di testi pubblicitari, implementando meccanismi di rate limiting per gestire le restrizioni dell'API. Questo garantisce un utilizzo efficiente del servizio esterno, bilanciando la necessità di generare contenuti personalizzati con i vincoli imposti dal provider.

4.5.4.1.3 Serializzazione e deserializzazione dei messaggi

La gestione della serializzazione e deserializzazione è fondamentale nell'architettura Flink_G per garantire l'efficiente trasferimento dei dati tra i componenti del sistema_G streaming. Il sistema_G implementa serializzatori personalizzati che garantiscono coerenza e integrità dei dati durante l'elaborazione.

4.5.4.2 Diagrammi delle classi

Seguono i diagrammmi delle classi, suddiviso per comodita in 2 parti, rispetto alla classe `FlinkJobManager`

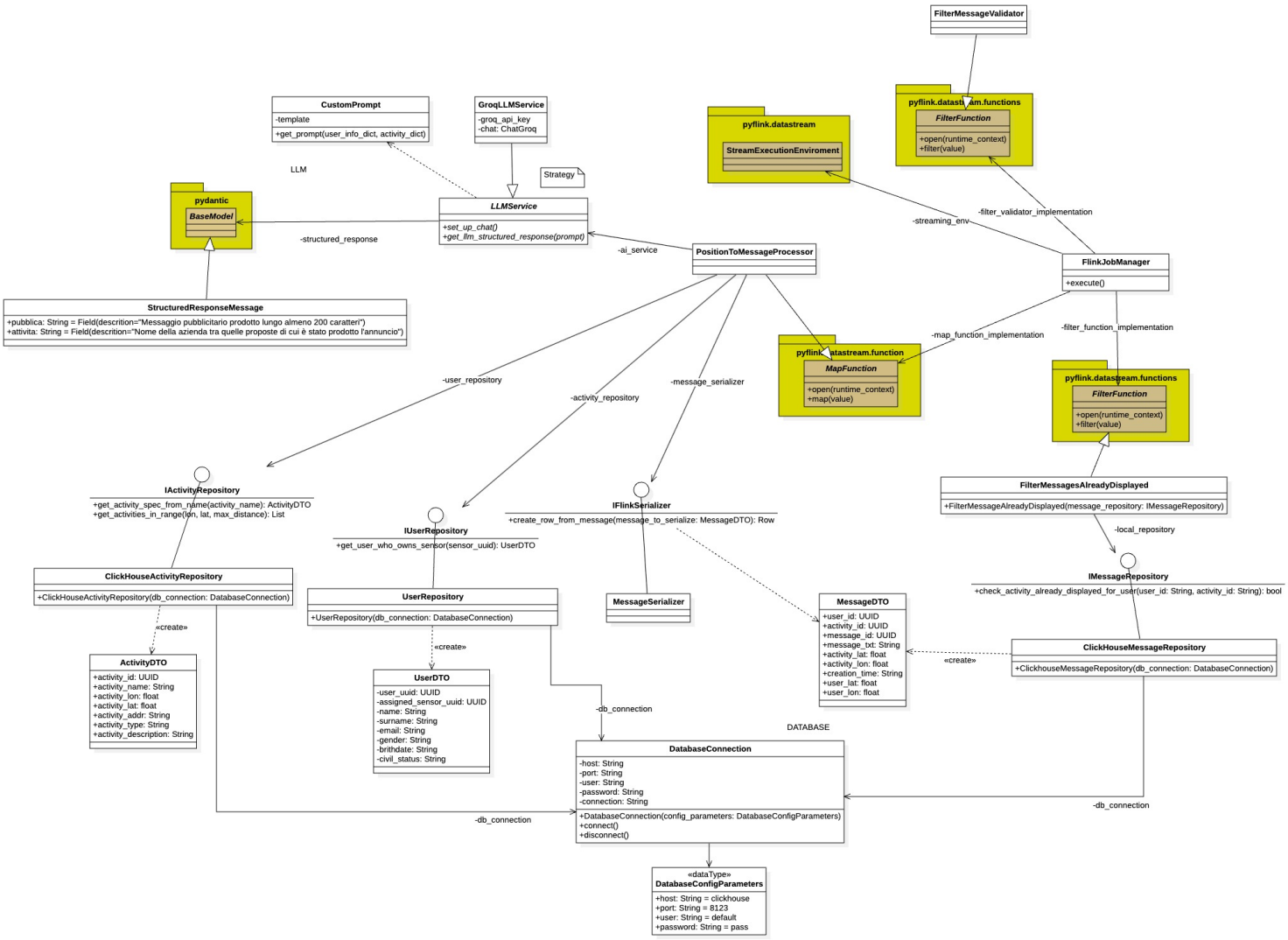

Figure 7: PositionToMessageProcessorService InBound/OutBound Ports con il Broker$_G$

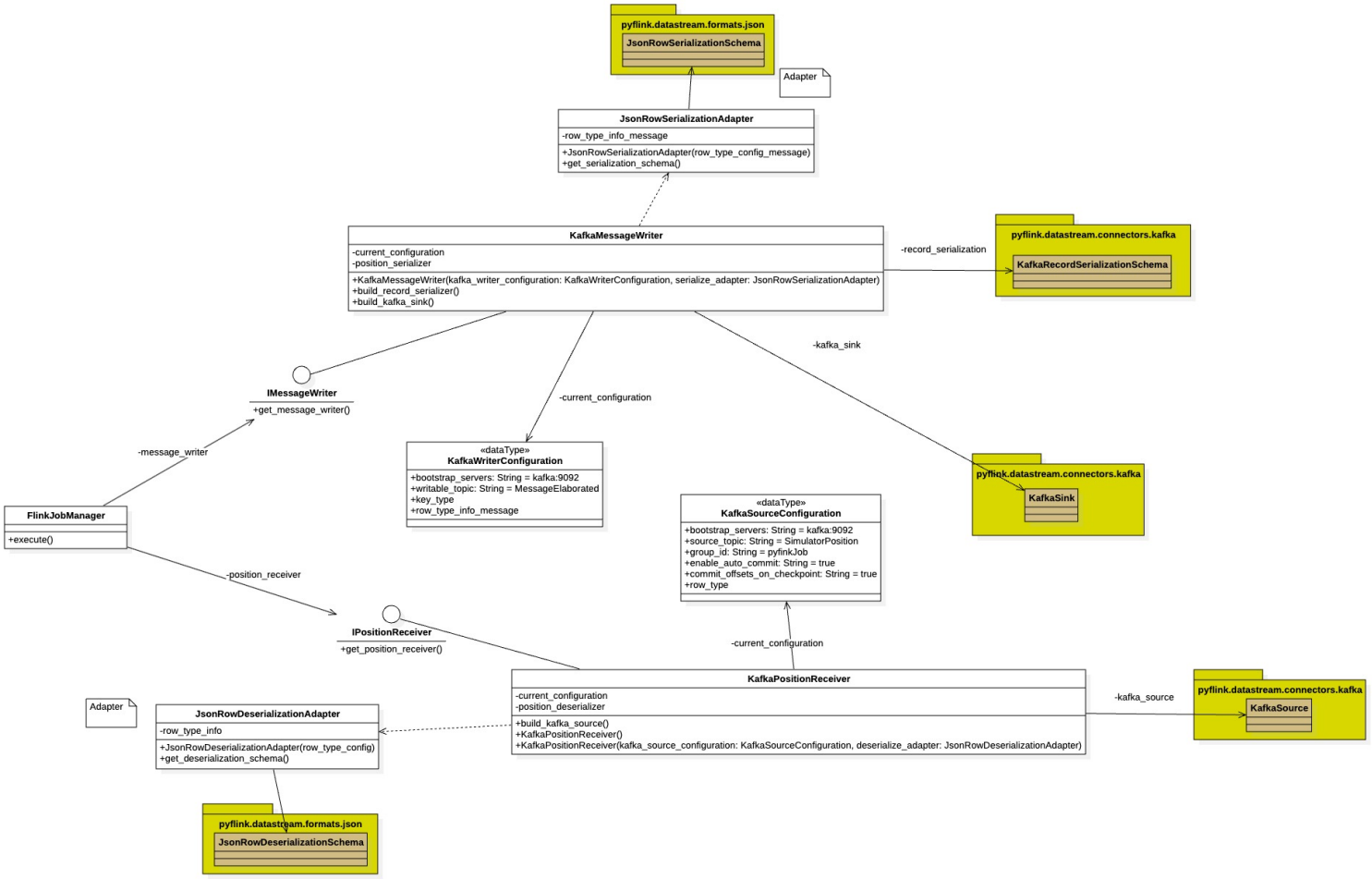

Figure 8: PositionToMessageProcessorService OutBound Ports

4.5.4.3 Design Pattern - Adapter Pattern

- **Motivazioni e studio del design-pattern$_G$**
  - Nel contesto della nostra architettura-esagonale$_G$, l'Adapter Pattern risulta essenziale per facilitare l'interazione tra la business logic e le componenti esterne (ad esempio, i servizi di pubblicazione su Kafka$_G$ o la ricezione di dati da esso);
  - Grazie a questo approccio, manteniamo l'indipendenza tra i moduli interni e le librerie di terze parti, riducendo i vincoli e semplificando la futura sostituzione di tali componenti senza impattare sul sistema$_G$;
  - Questo pattern consente di adattare interfacce incompatibili e promuove il riutilizzo del codice, proteggendo la logica di business dai dettagli implementativi delle tecnologie esterne.
- **Implementazione del design-pattern$_G$**
  - L'implementazione del pattern Adapter avviene tramite:
    1. Interfacce ben definite (`IMessageWriter` e `IPositionReceiver`) che dichiarano i metodi essenziali per l'interazione con l'architettura esagonale;
    2. Classi adapter concrete (`KafkaMessageWriter` e `KafkaPositionReceiver`) che implementano tali interfacce, incapsulando la complessità di conversione tra i formati interni e quelli richiesti da Kafka$_G$;
    3. Adapter di serializzazione/deserializzazione (`JsonRowSerializationAdapter` e `JsonRowDeserializationAdapter`) che gestiscono la conversione dei dati.

- **Integrazione del pattern**
  - `KafkaMessageWriter` agisce come adapter per l'invio di messaggi a Kafka$_G$:
    1. Riceve una configurazione (`KafkaWriterConfiguration`) e un adapter di serializzazione;
    2. Costruisce un serializzatore di record Kafka$_G$ configurandolo con il topic$_G$ di destinazione e gli schemi di serializzazione per chiave e valore;
    3. Crea un sink Kafka$_G$ con i server bootstrap e il serializzatore configurati;
    4. Espone il metodo `get_message_writer()` che restituisce il sink configurato, nascondendo tutti i dettagli di implementazione di Kafka$_G$.
  - `KafkaPositionReceiver` agisce come adapter per la ricezione di posizioni da Kafka$_G$:
    1. Riceve una configurazione (`KafkaSourceConfiguration`) e un adapter di deserializzazione;
    2. Costruisce una sorgente Kafka$_G$ configurandola con server bootstrap, topic$_G$, gruppo consumer e schema di deserializzazione;
    3. Configura proprietà specifiche come auto-commit e gestione degli offset;
    4. Espone il metodo `get_position_receiver()` che restituisce la sorgente configurata, nascondendo i dettagli di Kafka$_G$.
  - Grazie a questi adapter, il core dell'applicazione può interagire con Kafka$_G$ attraverso un'interfaccia semplificata e coerente, senza essere esposto ai dettagli implementativi della libreria Kafka$_G$. Se in futuro fosse necessario sostituire Kafka$_G$ con un altro broker$_G$ di messaggistica, basterebbe implementare nuovi adapter che rispettino le stesse interfacce.

4.5.4.4 Design Pattern - Strategy Pattern

- **Motivazioni e studio del design-pattern$_G$**
  - Nel contesto della nostra architettura, il Pattern Strategy risulta fondamentale per gestire diverse implementazioni di modelli linguistici (LLM) senza modificare il codice client che li utilizza;
  - Questo pattern permette di definire una famiglia di algoritmi (in questo caso, diverse implementazioni di servizi LLM$_G$), incapsularli in classi separate e renderli intercambiabili a runtime;
  - Grazie a questo approccio, possiamo estendere facilmente le capacità del sistema$_G$ aggiungendo nuovi servizi LLM$_G$ senza modificare la logica di business che li utilizza, garantendo una maggiore flessibilità e manutenibilità del codice.
- **Implementazione del design-pattern$_G$**
  - L'implementazione del pattern Strategy avviene tramite:
    1. Un'interfaccia astratta `LLMService` che definisce il contratto comune per tutti i servizi LLM$_G$, dichiarando i metodi essenziali come `set_up_chat()` e `get_llm_structured_response()`;
    2. Classi concrete (come `GroqLLMService`) che implementano l'interfaccia `LLMService`, fornendo implementazioni specifiche per interagire con diversi provider di LLM$_G$;
    3. Un meccanismo di configurazione che permette di selezionare la strategia appropriata a runtime.
- **Integrazione del pattern**
  - `LLMService` agisce come interfaccia strategica:
    1. Definisce un costruttore che riceve un modello strutturato di risposta (`structured_response`);
    2. Dichiara il metodo astratto `set_up_chat()` che deve essere implementato per inizializzare la comunicazione con il servizio LLM$_G$ specifico;
    3. Dichiara il metodo astratto `get_llm_structured_response(prompt)` che deve essere implementato per ottenere risposte strutturate dal LLM$_G$.
  - `GroqLLMService` rappresenta una strategia concreta:
    1. Implementa l'interfaccia `LLMService` fornendo un'implementazione specifica per il servizio Groq$_G$;

2. Configura parametri specifici come la chiave API$_G$, il modello ("Gemma2-9b-it"), la temperatura e altri parametri propri di Groq$_G$;
3. Implementa `set_up_chat()` impostando un limitatore di frequenza e inizializzando il client ChatGroq;
4. Implementa `get_llm_structured_response(prompt)` utilizzando la funzionalità `with_structured_output` di ChatGroq per ottenere risposte nel formato desiderato.

– Il client (in questo caso, `PositionToMessageProcessor`) interagisce con l'interfaccia `LLMService` senza conoscere quale implementazione specifica viene utilizzata:

1. Riceve un'istanza di `LLMService` tramite dependency injection;
2. Chiama il metodo `get_llm_structured_response()` sull'interfaccia, delegando l'implementazione concreta alla strategia selezionata;
3. Non ha bisogno di conoscere i dettagli implementativi di Groq$_G$ o di qualsiasi altro servizio LLM$_G$.

– Grazie a questo pattern, il sistema$_G$ può facilmente supportare nuovi servizi LLM$_G$ (come OpenAI, Claude, ecc.) semplicemente creando nuove classi che implementano l'interfaccia `LLMService`, senza modificare il codice che utilizza questi servizi. Ciò garantisce un'elevata estensibilità e facilità di manutenzione.

4.5.4.5 Classi, interfacce, metodi e attributi:

4.5.4.6 FlinkJobManager

- **Descrizione**: Implementa un gestore per job Apache Flink$_G$ che configura e orchestra un pipeline di elaborazione dati in streaming. Costruisce un flusso di dati completo con operazioni di ricezione, trasformazione e invio di messaggi;
- **Attributi**:
  – `__streaming_env: StreamExecutionEnvironment` - Ambiente di esecuzione Flink$_G$ per l'elaborazione in streaming;
  – `__populated_datastream` - Stream di dati iniziale popolato dalla sorgente di posizioni;
  – `__keyed_stream` - Stream partizionato (keyed) in base all'identificatore della posizione;
  – `__validated_stream` - Stream filtrato contenente solo le posizioni validate;
  – `__mapped_stream` - Stream con dati trasformati dal formato di input al formato di output;
  – `__filtered_stream` - Stream finale filtrato prima dell'invio al sink.
- **Operazioni**:
  – `__init__(self, streaming_env_istance: StreamExecutionEnvironment, map_function_implementation: MapFunction, filter_validator_implementation: FilterFunction, filter_function_implementation: FilterFunction, position_receiver_istance: IPositionReceiver, message_sender_istance: IMessageWriter)` - Costruttore che configura l'intero pipeline di elaborazione dati. Inizializza l'ambiente di streaming, crea uno stream di dati dalla sorgente, applica operazioni di chiave, validazione, mappatura e filtraggio, e configura il sink per l'output;
  – `execute(self)` - Avvia l'esecuzione del job Flink$_G$ con l'identificatore "Flink Job".

4.5.4.7 IMessageWriter

- **Descrizione**: Implementa un'interfaccia astratta che definisce il contratto per i componenti di scrittura dei messaggi. Rappresenta una porta in uscita (outbound port) nel sistema$_G$, responsabile di fornire un meccanismo standardizzato per scrivere messaggi verso sistemi esterni;
- **Attributi**:
  – Nessun attributo definito a livello di interfaccia.

- **Operazioni**:
  - `get_message_writer(self)` - Metodo astratto che deve essere implementato dalle sottoclassi per fornire l'istanza concreta dello scrittore di messaggi. L'implementazione dipenderà dal sistema$_G$ di destinazione specifico (ad esempio, un sink Kafka$_G$ o un altro sistema$_G$ di messaggistica).

4.5.4.8 KafkaMessageWriter

- **Descrizione**: Implementa l'interfaccia IMessageWriter per fornire un adattatore specifico per la scrittura di messaggi su Apache Kafka$_G$. Configura e costruisce un sink Kafka$_G$ per l'integrazione con il pipeline di elaborazione dati di Apache Flink$_G$;
- **Attributi**:
  - `__current_configuration: KafkaWriterConfiguration` - Configurazione contenente i parametri necessari per la connessione a Kafka$_G$ e la serializzazione dei messaggi;
  - `__position_serializer` - Schema di serializzazione per convertire gli oggetti di posizione in formato JSON$_G$;
  - `__record_serializer: KafkaRecordSerializationSchema` - Schema di serializzazione per i record Kafka$_G$ che include sia la chiave che il valore;
  - `__kafka_sink: KafkaSink` - Componente sink di Flink$_G$ configurato per scrivere su Kafka$_G$.
- **Operazioni**:
  - `__init__(self, kafka`$_G$`_writer_configuration: KafkaWriterConfiguration, serialize_adapter: JsonRowSerializationAdapter)` - Costruttore che inizializza l'adattatore con la configurazione Kafka$_G$ e l'adattatore di serializzazione JSON$_G$, e costruisce il serializzatore di record e il sink Kafka$_G$;
  - `build_record_serializer(self)` - Metodo privato che costruisce lo schema di serializzazione per i record Kafka$_G$, configurando il topic$_G$ di destinazione, lo schema di serializzazione della chiave e dello schema di serializzazione del valore;
  - `build_kafka_sink(self)` - Metodo privato che costruisce il sink Kafka$_G$ utilizzando i server bootstrap e il serializzatore di record precedentemente configurati;
  - `get_message_writer(self)` - Implementazione del metodo dell'interfaccia che restituisce l'istanza del sink Kafka$_G$ configurato, pronto per essere utilizzato nel pipeline Flink$_G$.

4.5.4.9 JsonRowSerializationAdapter

- **Descrizione**: Implementa un adattatore per la serializzazione di dati in formato JSON$_G$. Incapsula la configurazione e la creazione di uno schema di serializzazione JSON$_G$ per l'uso nei flussi di dati di Apache Flink$_G$;
- **Attributi**:
  - `__row_type_info_message` - Informazioni sul tipo di riga che definisce la struttura dei dati da serializzare. Specifica lo schema e i tipi di dati per il processo$_G$ di serializzazione JSON$_G$.
- **Operazioni**:
  - `__init__(self, row_type_config_message)` - Costruttore che inizializza l'adattatore con le informazioni di tipo necessarie per definire la struttura dei messaggi da serializzare;
  - `get_serialization_schema(self)` - Restituisce uno schema di serializzazione JSON$_G$ configurato con le informazioni di tipo fornite. Lo schema creato può essere utilizzato per convertire oggetti Flink$_G$ Row in stringhe JSON$_G$ formattate secondo lo schema definito.

4.5.4.10 KafkaWriterConfiguration

- **Descrizione**: Implementa una classe di dati (dataclass) che contiene la configurazione necessaria per scrivere messaggi su Apache Kafka$_G$ dal pipeline Flink$_G$. Definisce sia i parametri di connessione sia la struttura dei dati da serializzare;
- **Attributi**:
  - `bootstrap_servers: str` - Indirizzo e porta dei server bootstrap Kafka$_G$. Il valore predefinito è "kafka:9092";
  - `writable_topic: str` - Nome del topic$_G$ Kafka$_G$ su cui scrivere i messaggi elaborati. Il valore predefinito è "MessageElaborated";
  - `key_type` - Definizione del tipo di chiave per i record Kafka$_G$, strutturata come una riga con un singolo campo 'user_uuid' di tipo stringa;
  - `row_type_info_message` - Definizione completa dello schema dei messaggi, strutturata come una riga con nove campi che rappresentano le informazioni dell'utente, dell'attività e del messaggio, con i relativi tipi di dati (stringhe per identificatori e messaggio, float per coordinate geografiche).
- **Operazioni**:
  - Nessuna operazione esplicita definita, in quanto si tratta di una dataclass che fornisce automaticamente costruttore, rappresentazione in stringa, confronto e altre funzionalità.

4.5.4.11 IPositionReceiver

- **Descrizione**: Implementa un'interfaccia astratta che definisce il contratto per i componenti di ricezione delle posizioni. Rappresenta una porta in entrata (inbound port) nel sistema$_G$, responsabile di fornire un meccanismo standardizzato per ricevere dati di posizione da fonti esterne;
- **Attributi**:
  - Nessun attributo definito a livello di interfaccia.
- **Operazioni**:
  - `get_position_receiver(self)` - Metodo astratto che deve essere implementato dalle sottoclassi per fornire l'istanza concreta del ricevitore di posizioni. L'implementazione dipenderà dalla fonte di dati specifica (ad esempio, un source Kafka$_G$ o un altro sistema$_G$ di messaggistica).

4.5.4.12 KafkaPositionReceiver

- **Descrizione**: Implementa l'interfaccia IPositionReceiver per fornire un adattatore specifico per la ricezione di posizioni da Apache Kafka$_G$. Configura e costruisce una fonte Kafka$_G$ per l'integrazione con il pipeline di elaborazione dati di Apache Flink$_G$;
- **Attributi**:
  - `__current_configuration: KafkaSourceConfiguration` - Configurazione contenente i parametri necessari per la connessione a Kafka$_G$ e la deserializzazione dei messaggi;
  - `__position_deserializer` - Schema di deserializzazione per convertire i messaggi JSON$_G$ ricevuti in oggetti di posizione;
  - `__kafka_source: KafkaSource` - Componente source di Flink$_G$ configurato per leggere da Kafka$_G$.
- **Operazioni**:
  - `__init__(self, kafka_source_configuration: KafkaSourceConfiguration, deserialize_adapter: JsonRowDeserializationAdapter)` - Costruttore che inizializza l'adattatore con la configurazione Kafka$_G$ e l'adattatore di deserializzazione JSON$_G$, e costruisce la fonte Kafka$_G$;

- `build_kafka_source(self) -> KafkaSource` - Metodo privato che costruisce la fonte Kafka$_G$ configurando i server bootstrap, il topic$_G$ di origine, l'ID del gruppo di consumatori, il deserializzatore e altre proprietà specifiche come l'auto-commit e il commit degli offset sui checkpoint;
- `get_position_receiver(self)` - Implementazione del metodo dell'interfaccia che restituisce l'istanza della fonte Kafka$_G$ configurata, pronta per essere utilizzata nel pipeline Flink$_G$.

4.5.4.13 JsonRowDeserializationAdapter

- **Descrizione**: Implementa un adattatore per la deserializzazione di dati in formato JSON$_G$. Incapsula la configurazione e la creazione di uno schema di deserializzazione JSON$_G$ per l'uso nei flussi di dati di Apache Flink$_G$;
- **Attributi**:
  - `__row_type_info` - Informazioni sul tipo di riga che definisce la struttura dei dati da deserializzare. Specifica lo schema e i tipi di dati per il processo$_G$ di deserializzazione JSON$_G$.
- **Operazioni**:
  - `__init__(self, row_type_config)` - Costruttore che inizializza l'adattatore con le informazioni di tipo necessarie per definire la struttura dei messaggi da deserializzare;
  - `get_deserialization_schema(self)` - Restituisce uno schema di deserializzazione JSON$_G$ configurato con le informazioni di tipo fornite. Lo schema creato può essere utilizzato per convertire stringhe JSON$_G$ in oggetti Flink$_G$ Row strutturati secondo lo schema definito.

4.5.4.14 KafkaSourceConfiguration

- **Descrizione**: Implementa una classe di dati (dataclass) che contiene la configurazione necessaria per leggere messaggi da Apache Kafka$_G$ nel pipeline Flink$_G$. Definisce sia i parametri di connessione sia la struttura dei dati da deserializzare;
- **Attributi**:
  - `bootstrap_servers: str` - Indirizzo e porta dei server bootstrap Kafka$_G$. Il valore predefinito è "kafka:9092";
  - `source_topic: str` - Nome del topic$_G$ Kafka$_G$ da cui leggere i messaggi di posizione. Il valore predefinito è "SimulatorPosition";
  - `group_id: str` - Identificatore del gruppo di consumatori Kafka$_G$. Il valore predefinito è "pyfinkJob";
  - `enable_auto_commit: str` - Flag che indica se abilitare il commit automatico degli offset. Il valore predefinito è "true";
  - `commit_offsets_on_checkpoint: str` - Flag che indica se commettere gli offset durante i checkpoint Flink$_G$. Il valore predefinito è "true";
  - `row_type` - Definizione dello schema di deserializzazione delle posizioni, strutturato come una riga con quattro campi: 'user_uuid' (string), 'latitude' (float), 'longitude' (float) e 'received_at' (string).
- **Operazioni**:
  - Nessuna operazione esplicita definita, in quanto si tratta di una dataclass che fornisce automaticamente costruttore, rappresentazione in stringa, confronto e altre funzionalità.

4.5.4.15 FilterMessageValidator

- **Descrizione**: Implementa la classe FilterFunction di Apache Flink$_G$ per filtrare messaggi Kafka$_G$ invalidi o potenzialmente dannosi. Esegue diversi controlli di validazione sui dati ricevuti per garantire l'integrità e la sicurezza del pipeline di elaborazione;
- **Attributi**:

- Nessun attributo specifico definito nella classe.

- **Operazioni**:
  - `open(self, runtime_context)` - Metodo che viene chiamato all'inizializzazione della funzione di filtro. Non implementa operazioni specifiche in questa versione;
  - `filter(self, value)` - Implementa il metodo dell'interfaccia FilterFunction per determinare quali messaggi devono essere mantenuti nel flusso. Esegue diversi controlli di validazione:
    * Verifica che latitudine e longitudine siano valori numerici e rientrino nei range geografici validi (-90 ¡= lat ¡= 90, -180 ¡= lon ¡= 180);
    * Controlla che il timestamp sia in un formato valido ('%Y-%m-%d %H:%M:%S').;
    * Verifica che l'ID utente sia un UUID valido di versione 4;
    * Analizza tutti i valori di tipo stringa per identificare pattern sospetti di SQL_G injection (come "–", ";", comandi SQL_G come "DROP", "DELETE", ecc.).

    Restituisce True solo se tutti i controlli di validazione vengono superati, altrimenti restituisce False per scartare il messaggio dal flusso.

4.5.4.16 PositionToMessageProcessor

- **Descrizione**: Implementa la classe MapFunction di Apache $Flink_G$ per trasformare dati di posizione in messaggi personalizzati. Utilizza un servizio di intelligenza artificiale per generare contenuti pubblicitari contestuali basati sulla posizione dell'utente e sulle attività disponibili nelle vicinanze;

- **Attributi**:
  - `ai_service: LLMService` - Servizio di modello di linguaggio (LLM) utilizzato per generare contenuti pubblicitari personalizzati;
  - `__user_repository: IUserRepository` - $Repository_G$ per accedere alle informazioni sugli utenti;
  - `__activity_repository: IActivityRepository` - $Repository_G$ per recuperare attività commerciali nelle vicinanze dell'utente;
  - `__message_serializer: IFlinkSerializable` - Componente per serializzare i messaggi nel formato richiesto da $Flink_G$;
  - `prompt_creator: CustomPrompt` - Generatore di prompt per l'interazione con il servizio LLM_G.

- **Operazioni**:
  - `__init__(self, ai_chatbot_service: LLMService, user_repository: IUserRepository, activity_repository: IActivityRepository, message_serializer: IFlinkSerializable)` - Costruttore che inizializza il processore con i servizi e $repository_G$ necessari;
  - `open(self, runtime_context)` - Metodo chiamato all'inizializzazione del pipeline $Flink_G$. Configura il servizio LLM_G e inizializza il generatore di prompt;
  - `map(self, value)` - Implementa il metodo dell'interfaccia MapFunction per trasformare i dati di posizione in messaggi. Esegue il seguente $processo_G$:
    * Recupera le informazioni dell'utente associato al sensore;
    * Trova le attività commerciali nel raggio di 300 metri dalla posizione;
    * Se non ci sono attività nelle vicinanze, restituisce un messaggio segnaposto;
    * Altrimenti, genera un prompt personalizzato basato sull'utente e sulle attività;
    * Invoca il servizio LLM_G per ottenere una risposta strutturata;
    * Recupera le informazioni dettagliate sull'attività selezionata;
    * Crea un oggetto MessageDTO con il contenuto pubblicitario generato;
    * Serializza il messaggio nel formato Row di $Flink_G$ per l'elaborazione successiva.

4.5.4.17 LLMService

- **Descrizione**: Implementa un'interfaccia astratta che definisce il contratto per servizi di modelli linguistici (LLM). Fornisce una struttura comune per interagire con diversi modelli di linguaggio e ottenere risposte strutturate in un formato predefinito.
- **Attributi**:
  - `_llm_structured_response: BaseModel` - Modello Pydantic che definisce la struttura attesa per la risposta del modello linguistico.
- **Operazioni**:
  - `__init__(self, structured_response: BaseModel)` - Costruttore che inizializza il servizio con un modello Pydantic che definisce la struttura della risposta attesa dal modello linguistico;
  - `set_up_chat(self)` - Metodo astratto che deve essere implementato dalle sottoclassi per inizializzare e configurare la sessione di chat con il modello linguistico;
  - `get_llm_structured_response(self, prompt)` - Metodo astratto che deve essere implementato dalle sottoclassi per inviare un prompt al modello linguistico e ottenere una risposta strutturata secondo il modello Pydantic definito.

4.5.4.18 CustomPrompt

- **Descrizione**: Implementa un generatore di prompt personalizzati per l'interazione con modelli linguistici. Utilizza un template predefinito per costruire richieste strutturate che mirano a ottenere messaggi pubblicitari contestualizzati in base alle informazioni dell'utente e alle attività commerciali disponibili;
- **Attributi**:
  - `__template: Template` - Template di stringa che definisce la struttura del prompt, con variabili per inserire dinamicamente informazioni sull'utente e sulle attività commerciali.
- **Operazioni**:
  - `__init__(self)` - Costruttore che inizializza il generatore di prompt con un template predefinito. Il template include istruzioni per il modello linguistico su come generare un messaggio pubblicitario personalizzato, con criteri specifici per la selezione dell'attività e vincoli sul formato della risposta;
  - `get_prompt(self, user_info_dict, activity_dict)` - Genera un prompt personalizzato sostituendo le variabili del template con le informazioni specifiche dell'utente e l'elenco delle attività disponibili. Formatta le attività come una lista puntata e restituisce il prompt completo pronto per essere inviato al modello linguistico.

4.5.4.19 StructuredResponseMessage

- **Descrizione**: Implementa un modello Pydantic che definisce la struttura della risposta attesa dal modello linguistico. Garantisce che le risposte generate soddisfino un formato predefinito, facilitando l'elaborazione e la validazione automatica dei dati ricevuti;
- **Attributi**:
  - `pubblicita: str` - Campo che contiene il messaggio pubblicitario generato dal modello linguistico. Deve essere lungo almeno 200 caratteri come specificato nella descrizione del campo;
  - `attivita: str` - Campo che contiene il nome dell'azienda selezionata tra quelle proposte per cui è stato prodotto l'annuncio pubblicitario.
- **Operazioni**:
  - Nessuna operazione esplicita definita, in quanto si tratta di un modello Pydantic che fornisce automaticamente funzionalità di validazione, serializzazione, deserializzazione e altre funzionalità di gestione dei dati.

4.5.4.20 GroqLLMService

- **Descrizione**: Implementa una classe concreta che estende l'interfaccia LLMService per interagire specificamente con l'API Groq$_G$. Configura e gestisce una connessione a un modello linguistico di Groq$_G$, con controllo della frequenza delle richieste e gestione delle risposte strutturate;
- **Attributi**:
  - `__groq_api_key: str` - Chiave API$_G$ per autenticarsi al servizio Groq$_G$, recuperata dalle variabili d'ambiente;
  - `__chat: ChatGroq` - Istanza del client di chat Groq$_G$ configurata con parametri specifici per l'interazione con il modello linguistico.
- **Operazioni**:
  - `__init__(self, structured_response)` - Costruttore che inizializza il servizio ereditando dalla classe base e configurando la chiave API$_G$ Groq$_G$ dalle variabili d'ambiente;
  - `set_up_chat(self)` - Implementa il metodo astratto della classe base per inizializzare il client di chat Groq$_G$. Configura un rate limiter per controllare la frequenza delle richieste API$_G$ (circa una ogni 15 secondi) e inizializza il client ChatGroq con parametri specifici come il modello "Gemma2-9b-it", temperatura di generazione, e altre configurazioni per la gestione delle richieste;
  - `get_llm_structured_response(self, prompt)` - Implementa il metodo astratto della classe base per inviare un prompt al modello Groq$_G$ e ottenere una risposta strutturata. Utilizza la funzionalità "with_structured_output" del client ChatGroq per forzare la risposta nel formato definito dal modello Pydantic specificato nella classe base.

4.5.4.21 IActivityRepository

- **Descrizione**: Implementa un'interfaccia astratta che definisce il contratto per i repository$_G$ di gestione delle attività commerciali. Stabilisce i metodi necessari per recuperare informazioni sulle attività in base al nome o alla posizione geografica;
- **Attributi**:
  - Nessun attributo definito a livello di interfaccia.
- **Operazioni**:
  - `get_activity_spec_from_name(self, activity_name) -> ActivityDTO` - Metodo astratto che deve essere implementato dalle sottoclassi per recuperare le specifiche dettagliate di un'attività commerciale in base al suo nome. Restituisce un oggetto ActivityDTO contenente tutte le informazioni sull'attività;
  - `get_activities_in_range(self, lon, lat, max_distance) -> list` - Metodo astratto che deve essere implementato dalle sottoclassi per recuperare un elenco di attività commerciali situate entro una distanza massima specificata da una data posizione geografica. Prende in input le coordinate (longitudine e latitudine) e la distanza massima, e restituisce una lista di attività nelle vicinanze.

4.5.4.22 ClickhouseActivityRepository

- **Descrizione**: Implementa la classe concreta che realizza l'interfaccia IActivityRepository per recuperare informazioni sulle attività commerciali da un database$_G$ Clickhouse$_G$. Fornisce metodi per cercare attività in base alla vicinanza geografica e al nome;
- **Attributi**:
  - `__db_conn: DatabaseConnection` - Connessione al database$_G$ Clickhouse$_G$ utilizzata per eseguire query$_G$ sui dati delle attività.
- **Operazioni**:

- `__init__(self, db_connection: DatabaseConnection)` - Costruttore che inizializza il repository$_G$ con una connessione al database$_G$;
- `get_activities_in_range(self, lon, lat, max_distance) -> list` - Implementa il metodo dell'interfaccia per trovare attività commerciali entro una distanza specificata da una posizione geografica. Esegue una query$_G$ SQL$_G$ che utilizza la funzione geoDistance per calcolare la distanza tra la posizione fornita e ogni attività, restituendo quelle che si trovano entro la distanza massima specificata. Il risultato comprende nome, indirizzo, tipo, descrizione e distanza calcolata;
- `get_activity_spec_from_name(self, activity_name) -> ActivityDTO` - Implementa il metodo dell'interfaccia per recuperare i dettagli completi di un'attività in base al suo nome. Esegue una query$_G$ SQL$_G$ che cerca un'attività con il nome esatto fornito e costruisce un oggetto ActivityDTO con tutti i dati recuperati (UUID, nome, coordinate, indirizzo, tipo e descrizione). Se nessuna attività corrisponde al nome, restituisce un ActivityDTO vuoto.

4.5.4.23 ActivityDTO

- **Descrizione**: Implementa un oggetto di trasferimento dati (Data Transfer Object) per la classe Activity. Viene utilizzato per astrarre e incapsulare i dati delle attività commerciali, facilitando il trasferimento delle informazioni tra i diversi strati dell'applicazione senza esporre i dettagli implementativi;
- **Attributi**:
  - `activity_id: uuid` - Identificatore univoco dell'attività commerciale;
  - `activity_name: str` - Nome dell'attività commerciale;
  - `activity_lon: float` - Longitudine della posizione geografica dell'attività;
  - `activity_lat: float` - Latitudine della posizione geografica dell'attività;
  - `activity_addr: str` - Indirizzo fisico dell'attività commerciale;
  - `activity_type: str` - Categoria o tipo di attività commerciale (es. Ristorante, Negozio, Servizio);
  - `activity_description: str` - Descrizione dettagliata dell'attività commerciale.
- **Operazioni**:
  - `__init__(self, activity_id: uuid = uuid.uuid4(), activity_name: str = "", activity_lon: float = 0.0, activity_lat: float = 0.0, activity_addr: str ="", activity_type: str = "", activity_description: str ="")` - Costruttore che inizializza l'oggetto DTO con tutti i dati dell'attività, fornendo valori predefiniti per tutti i parametri in modo da poter istanziare anche un oggetto vuoto.

4.5.4.24 IUserRepository

- **Descrizione**: Implementa un'interfaccia astratta che definisce il contratto per i repository$_G$ di gestione degli utenti nel contesto dell'applicazione Flink$_G$. Stabilisce il metodo necessario per recuperare le informazioni di un utente in base all'identificatore del sensore associato;
- **Attributi**:
  - Nessun attributo definito a livello di interfaccia.
- **Operazioni**:
  - `get_user_who_owns_sensor(self, sensor_uuid) -> UserDTO` - Metodo astratto che deve essere implementato dalle sottoclassi per recuperare i dati dell'utente proprietario di un sensore specifico. Prende in input l'UUID del sensore e restituisce un oggetto UserDTO contenente tutte le informazioni dell'utente associato.

4.5.4.25 ClickhouseUserRepository

- **Descrizione**: Implementa la classe concreta che realizza l'interfaccia IUserRepository per recuperare informazioni sugli utenti da un database$_G$ Clickhouse$_G$. Fornisce un metodo per trovare un utente in base all'UUID del sensore a lui assegnato, recuperando anche i suoi interessi;
- **Attributi**:
  - `__db_conn: DatabaseConnection` - Connessione al database$_G$ Clickhouse$_G$ utilizzata per eseguire query$_G$ sui dati degli utenti.
- **Operazioni**:
  - `__init__(self, db_connection: DatabaseConnection)` - Costruttore che inizializza il repository$_G$ con una connessione al database$_G$;
  - `get_user_who_owns_sensor(self, sensor_uuid) -> UserDTO` - Implementa il metodo dell'interfaccia per recuperare i dati dell'utente associato a un sensore specifico. Esegue una query$_G$ SQL$_G$ complessa che unisce le tabelle user e user_interest per ottenere tutti i dati dell'utente e i suoi interessi in un'unica operazione. La query$_G$ filtra gli utenti in base all'UUID del sensore fornito, raggruppa i risultati per i campi dell'utente e utilizza la funzione groupArray per aggregare gli interessi in un array. Restituisce un oggetto UserDTO popolato con tutti i dati recuperati, incluso l'elenco degli interessi, o None se nessun utente è associato al sensore specificato.

4.5.4.26 UserDTO

- **Descrizione**: Implementa un oggetto di trasferimento dati (Data Transfer Object) per la classe User nel contesto dell'applicazione Flink$_G$. Viene utilizzato per astrarre e incapsulare i dati degli utenti, facilitando il trasferimento delle informazioni tra i diversi strati dell'applicazione senza esporre i dettagli implementativi;
- **Attributi**:
  - `user_uuid: uuid` - Identificatore univoco dell'utente;
  - `assigned_sensor_uuid: uuid` - Identificatore univoco del sensore assegnato all'utente;
  - `name: str` - Nome dell'utente;
  - `surname: str` - Cognome dell'utente;
  - `email: str` - Indirizzo email dell'utente;
  - `gender: str` - Genere dell'utente;
  - `birthdate: str` - Data di nascita dell'utente;
  - `civil_status: str` - Stato civile dell'utente;
  - `interests: list[str]` - Lista degli interessi dell'utente, opzionale (default None).
- **Operazioni**:
  - `__init__(self, user_uuid: uuid, assigned_sensor_uuid: uuid, name: str, surname: str, email: str, gender: str, birthdate: str, civil_status: str, interests: list[str] = None)` - Costruttore che inizializza l'oggetto DTO con tutti i dati dell'utente, includendo una lista opzionale di interessi che può essere None se non specificata.

4.5.4.27 IMessageRepository

- **Descrizione**: Implementa un'interfaccia astratta che definisce il contratto per i repository$_G$ di gestione dei messaggi. Stabilisce il metodo necessario per verificare se un'attività è già stata mostrata a un utente specifico, permettendo così di evitare ripetizioni nei messaggi pubblicitari;
- **Attributi**:
  - Nessun attributo definito a livello di interfaccia.

- **Operazioni**:
  - `check_activity_already_displayed_for_user(self, user_id: str, activity_id: str) -> bool` - Metodo astratto che deve essere implementato dalle sottoclassi per verificare se una specifica attività commerciale è già stata mostrata a un determinato utente. Prende in input l'identificatore dell'utente e l'identificatore dell'attività, e restituisce un valore booleano: True se l'attività è già stata mostrata all'utente, False altrimenti.

4.5.4.28 ClickhouseMessageRepository

- **Descrizione**: Implementa la classe concreta che realizza l'interfaccia IMessageRepository per gestire i messaggi in un database$_G$ Clickhouse$_G$. Fornisce funzionalità per verificare se un messaggio pubblicitario relativo a una specifica attività è già stato mostrato a un determinato utente;
- **Attributi**:
  - `__db_conn: DatabaseConnection` - Connessione al database$_G$ Clickhouse$_G$ utilizzata per eseguire query$_G$ sui dati dei messaggi.
- **Operazioni**:
  - `__init__(self, db_connection: DatabaseConnection)` - Costruttore che inizializza il repository$_G$ con una connessione al database$_G$;
  - `check_activity_already_displayed_for_user(self, user_id: str, activity_id: str) -> bool` - Implementa il metodo dell'interfaccia per verificare se una specifica attività è già stata mostrata a un determinato utente. Esegue una query$_G$ SQL$_G$ che cerca nella tabella dei messaggi record con l'UUID dell'utente e l'UUID dell'attività specificati. Restituisce True se trova almeno un record corrispondente (indicando che l'attività è già stata mostrata all'utente), False altrimenti;
  - `get_user_last_message(self, user_id) -> MessageDTO` - Metodo commentato (non attivo) che recupererebbe l'ultimo messaggio inviato a un utente specifico. La query$_G$ SQL$_G$ cercherebbe tutti i messaggi per l'utente specificato, ordinandoli per data di creazione in ordine decrescente e limitando il risultato a un solo record (il più recente). Restituirebbe un oggetto MessageDTO popolato con tutti i dati del messaggio.

4.5.4.29 MessageDTO

- **Descrizione**: Implementa un oggetto di trasferimento dati (Data Transfer Object) per la classe Message. Viene utilizzato per astrarre e incapsulare i dati dei messaggi pubblicitari, facilitando il trasferimento delle informazioni tra i diversi strati dell'applicazione e mantenendo le relazioni con utenti e attività commerciali;
- **Attributi**:
  - `user_id: str` - Identificatore univoco dell'utente a cui è destinato il messaggio;
  - `activity_id: str` - Identificatore univoco dell'attività commerciale a cui si riferisce il messaggio;
  - `message_id: str` - Identificatore univoco del messaggio stesso;
  - `message_text: str` - Contenuto testuale del messaggio pubblicitario;
  - `activity_lat: float` - Coordinata di latitudine dell'attività commerciale;
  - `activity_lon: float` - Coordinata di longitudine dell'attività commerciale;
  - `creation_time: str` - Timestamp che indica quando è stato creato il messaggio;
  - `user_lat: float` - Coordinata di latitudine dell'utente al momento della creazione del messaggio;
  - `user_lon: float` - Coordinata di longitudine dell'utente al momento della creazione del messaggio.
- **Operazioni**:

- `__init__(self, user_id: uuid = uuid.uuid4(), activity_id: uuid = uuid.uuid4(), message_id: uuid = uuid.uuid4(), message_text: str = "", activity_lat: float = 0.0, activity_lon: float = 0.0, creation_time: str = "", user_lat: float = 0.0, user_lon: float = 0.0)` - Costruttore che inizializza l'oggetto DTO con tutti i dati del messaggio, fornendo valori predefiniti per tutti i parametri in modo da poter istanziare anche un oggetto vuoto. Converte automaticamente gli UUID in stringhe per facilitare la serializzazione.

4.5.4.30 DatabaseConnection

- **Descrizione**: Implementa una classe che gestisce la connessione al database$_G$ Clickhouse$_G$ nel contesto dell'applicazione Flink$_G$. Fornisce metodi per stabilire e chiudere connessioni al database$_G$, incapsulando i dettagli di configurazione e gestione della connessione;
- **Attributi**:
  - `host: str` - Indirizzo del server Clickhouse$_G$, derivato dai parametri di configurazione;
  - `port: int` - Porta sulla quale il server Clickhouse$_G$ accetta connessioni, derivata dai parametri di configurazione;
  - `user: str` - Nome utente per l'autenticazione al database$_G$, derivato dai parametri di configurazione;
  - `password: str` - Password per l'autenticazione al database$_G$, derivata dai parametri di configurazione;
  - `connection` - Oggetto connessione al database$_G$ Clickhouse$_G$, inizialmente impostato a None.
- **Operazioni**:
  - `__init__(self, config_parameters: DatabaseConfigParameters)` - Costruttore che inizializza l'oggetto connessione con i parametri di configurazione del database$_G$ forniti;
  - `connect(self)` - Stabilisce una connessione al database$_G$ Clickhouse$_G$ utilizzando i parametri configurati e restituisce l'oggetto client di connessione. Utilizza la libreria clickhouse$_G$_connect per creare il client con i parametri di host, porta, nome utente e password;
  - `disconnect(self)` - Chiude la connessione al database$_G$ se attiva e reimposta l'attributo connection a None.

4.5.4.31 DatabaseConfigParameters

- **Descrizione**: Implementa una classe di dati (dataclass) che contiene i parametri di configurazione necessari per la connessione a un database$_G$ Clickhouse$_G$ nel contesto dell'applicazione Flink$_G$. Fornisce una struttura semplice per incapsulare e trasportare le impostazioni di configurazione del database$_G$ in modo tipizzato;
- **Attributi**:
  - `host: str` - Indirizzo del server Clickhouse$_G$. Il valore predefinito è "clickhouse";
  - `port: str` - Porta sulla quale il server Clickhouse$_G$ accetta connessioni. Il valore predefinito è "8123";
  - `user: str` - Nome utente per l'autenticazione al database$_G$. Il valore predefinito è "default";
  - `password: str` - Password per l'autenticazione al database$_G$. Il valore predefinito è "pass".
- **Operazioni**:
  - Nessuna operazione esplicita definita, in quanto si tratta di una dataclass che fornisce automaticamente costruttore, rappresentazione in stringa, confronto e altre funzionalità.

4.5.4.32 IFlinkSerializable

- **Descrizione**: Implementa un'interfaccia astratta che definisce il contratto per la serializzazione di oggetti in formato Row di Apache Flink$_G$. Stabilisce il metodo necessario per convertire oggetti di dominio (in particolare MessageDTO) nel formato di riga utilizzato da Flink$_G$ per elaborare i dati nel pipeline;
- **Attributi**:
  - Nessun attributo definito a livello di interfaccia.
- **Operazioni**:
  - `create_row_from_message(self, message_to_serialize: MessageDTO) -> Row` - Metodo astratto che deve essere implementato dalle sottoclassi per convertire un oggetto MessageDTO in un oggetto Row di Flink$_G$. Prende in input un'istanza di MessageDTO contenente i dati del messaggio e restituisce un oggetto Row che rappresenta questi dati in un formato compatibile con l'elaborazione Flink$_G$.

4.5.4.33 MessageSerializer

- **Descrizione**: Implementa la classe concreta che realizza l'interfaccia IFlinkSerializable per la serializzazione di oggetti MessageDTO in oggetti Row di Apache Flink$_G$. Fornisce una conversione strutturata dei dati di messaggio nel formato richiesto dal pipeline di elaborazione Flink$_G$;
- **Attributi**:
  - Nessun attributo specifico definito nella classe.
- **Operazioni**:
  - `create_row_from_message(self, message_to_serialize: MessageDTO) -> Row` - Implementa il metodo dell'interfaccia per convertire un oggetto MessageDTO in un oggetto Row di Flink$_G$. Crea una nuova istanza di Row inserendo ordinatamente tutti i campi del messaggio: identificatore dell'utente, identificatore dell'attività, identificatore del messaggio, testo del messaggio, coordinate geografiche dell'attività (latitudine e longitudine), timestamp di creazione, e coordinate geografiche dell'utente (latitudine e longitudine). Garantisce che gli identificatori siano convertiti in stringhe per una corretta serializzazione.

4.5.4.34 FilterMessageAlreadyDisplayed

- **Descrizione**: Implementa la classe FilterFunction di Apache Flink$_G$ per filtrare messaggi pubblicitari che sono già stati mostrati a un utente specifico o che non contengono informazioni geografiche valide. È parte del pipeline di elaborazione che garantisce che gli utenti non ricevano ripetutamente lo stesso messaggio pubblicitario;
- **Attributi**:
  - `__local_repository: IMessageRepository` - Repository$_G$ che fornisce metodi per verificare se un messaggio pubblicitario è già stato mostrato a un utente.
- **Operazioni**:
  - `__init__(self, message_repository: IMessageRepository)` - Costruttore che inizializza il filtro con un repository$_G$ di messaggi per accedere ai dati storici;
  - `open(self, runtime_context)` - Metodo che viene chiamato all'inizializzazione della funzione di filtro. Non implementa operazioni specifiche in questa versione;
  - `filter(self, value)` - Implementa il metodo dell'interfaccia FilterFunction per determinare quali messaggi devono essere mantenuti nel flusso. Esegue due controlli principali:
    - Verifica se l'attività pubblicizzata nel messaggio è già stata mostrata all'utente utilizzando il metodo del repository$_G$;
    - Controlla se le coordinate dell'attività sono valide (diverse da zero).

Restituisce False (scartando il messaggio) se l'attività è già stata mostrata all'utente o se le coordinate sono (0,0), indicando un messaggio segnaposto o non valido. Altrimenti restituisce True, permettendo al messaggio di proseguire nel pipeline.

Include anche una versione commentata di un algoritmo alternativo che avrebbe utilizzato l'ultimo messaggio mostrato all'utente per determinare se le stesse coordinate geografiche sono state già utilizzate.

4.5.5 ClickHouse

Nel progetto$_G$ Clickhouse$_G$ è stato scelto come database$_G$ data la possibilità di gestire sia dati time series sia dati geospaziali. Inoltre, questa soluzione ha offerto numerosi vantaggi in termini di performance e scalabilità del sistema$_G$.

4.5.5.1 Architettura MergeTree

Uno dei fattori che migliora le performance di Clickhouse$_G$ è il motore MergeTree. Ovvero a differenza dei database$_G$ tradizionali, MergeTree è ottimizzato per l'inserimento di grandi volumi di dati e query$_G$ analitiche complesse grazie alle seguenti caratteristiche:

- **Archiviazione colonnare**: I dati sono organizzati per colonne anziché per righe, permettendo:
  - Compressione più efficiente: le tecniche di archiviazione colonnare consentono una compressione da 10 a 100 volte superiore rispetto ai database$_G$ orientati a righe, grazie all'ottimizzazione di dati simili in ogni colonna;
  - I/O significativamente ridotto: le query$_G$ che richiedono solo un sottoinsieme di colonne leggono solo i dati necessari, riducendo l'input/output complessivo;
  - Elaborazione vettoriale che sfrutta le istruzioni SIMD (Single Instruction, Multiple Data): questo approccio permette al processore di eseguire la stessa operazione su più dati contemporaneamente, migliorando notevolmente le performance nelle operazioni di calcolo e aggregazione.
- **Organizzazione in parti (parts)**: I dati vengono inseriti in parti separate che vengono PoI$_G$ fuse in background:
  - Le nuove scritture vengono gestite in "parti" temporanee separate, garantendo che ogni batch di dati sia isolato, così da evitare conflitti e consentire una rapida acquisizione dei dati;
  - Un processo$_G$ di merge$_G$ periodico raccoglie queste parti piccole e le unisce in unità più grandi, ottimizzando la compressione e l'organizzazione dei dati per query$_G$ più efficienti;
  - Questo approccio permette di realizzare inserimenti massivi senza bloccare le operazioni di lettura, assicurando alta disponibilità e bassa latenza per le query$_G$.
- **Indici sparsi**: Ogni parte contiene un indice sparso per le colonne di ordinamento:
  - L'indice segmenta i dati in blocchi di 8192 righe, il che facilita una migliore organizzazione interna e favorisce una compressione più efficiente;
  - Per ogni blocco vengono registrati i valori minimi e massimi delle chiavi di ordinamento, fornendo un riassunto che consente di identificare rapidamente i range interessanti durante le query$_G$;
  - Durante l'esecuzione delle query$_G$, l'uso dei min/max permette di saltare interi blocchi che non contengono valori rilevanti, riducendo l'I/O e migliorando le prestazioni.
- **Partizionamento primario**: I dati vengono suddivisi fisicamente in base a una chiave di partizione:

```
1       PARTITION BY toYYYYMMDD(received_at)
```

  - Ogni partizione viene memorizzata in una directory separata, il che consente una gestione isolata dei dati e semplifica il backup e il ripristino di segmenti specifici;
  - Le query$_G$ possono escludere intere partizioni non rilevanti, riducendo il carico I/O e migliorando notevolmente le prestazioni durante le operazioni di ricerca;

. Questa organizzazione agevola operazioni di manutenzione mirate (come l'eliminazione o il trasferimento di dati storici), minimizzando l'impatto sul sistema$_G$ complessivo.

- **Ordinamento primario**: I dati all'interno di ogni partizione sono fisicamente ordinati:

```
        ORDER BY (sensor_id, received_at)
```

. L'ordinamento fisico accelera le ricerche per intervallo su queste colonne, consentendo di individuare rapidamente i record richiesti;

. Raggruppare valori simili migliora la compressione, riducendo lo spazio su disco e ottimizzando i tempi di lettura;

. Inoltre, definire l'ordine dei dati facilita la creazione e l'utilizzo efficiente degli indici sparsi, ottimizzando le prestazioni complessive delle query$_G$.

4.5.5.2 Schema del database

Lo schema del database$_G$ comprende le seguenti tabelle principali:

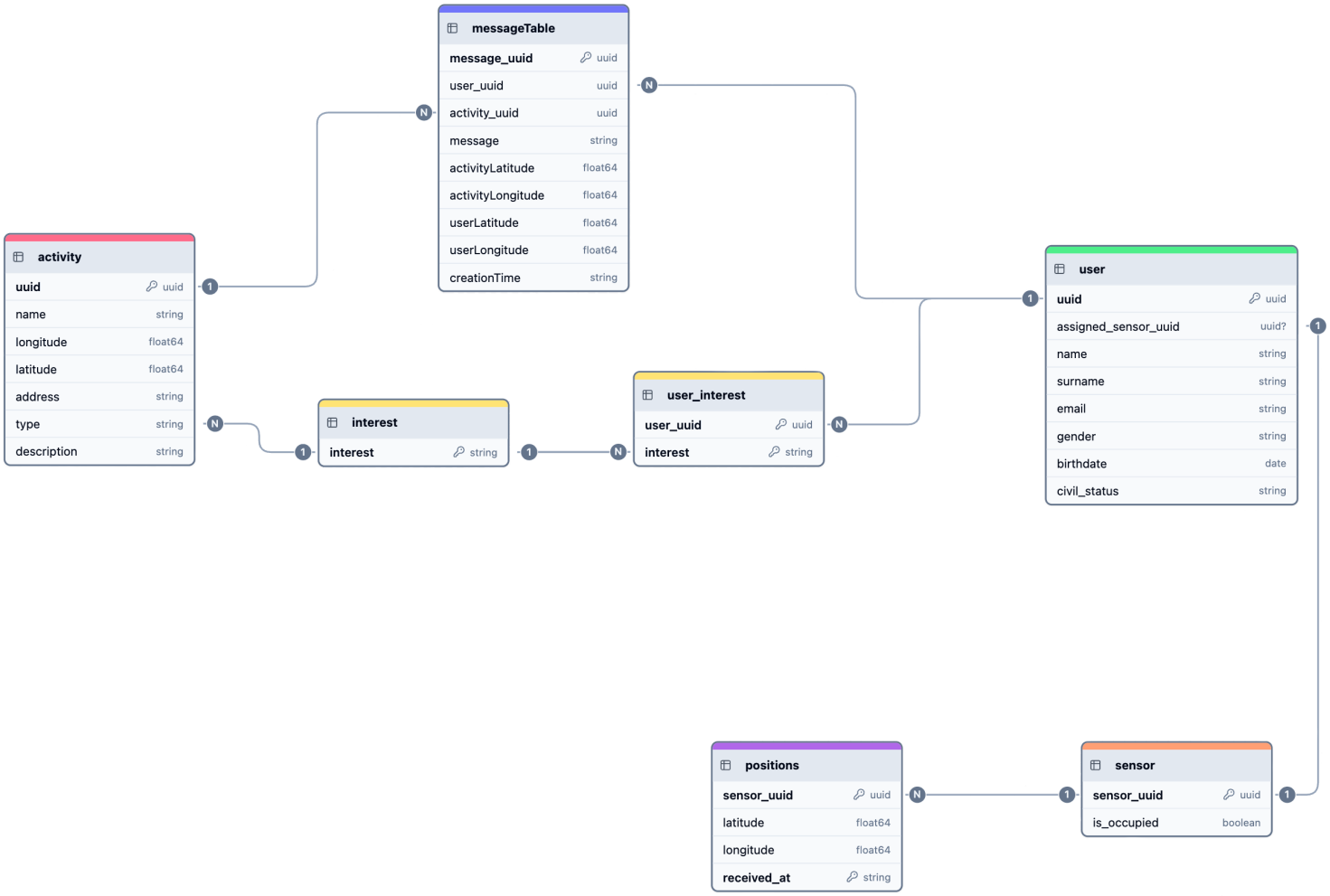

Figure 9: Schema del database$_G$

- **attivita**: Contiene i punti di interesse con coordinate geografiche e altri dati come nome, descrizione e tipo.

```
CREATE TABLE nearyou.activity(
    activity_uuid UUID,
    name String,
    longitude Float64,
    latitude Float64,
    address String,
    type String,
    description String,
    PRIMARY KEY(activity_uuid)
) ENGINE = MergeTree()
ORDER BY activity_uuid;
```

Questa tabella archivia le informazioni sui punti di interesse come negozi, ristoranti e attrazioni turistiche per cui verranno generati i messaggi pubblicitari personalizzati.

- **interesseUtente**: Contiene gli utenti e le loro preferenze.

```
CREATE TABLE nearyou.user_interest(
    user_uuid UUID,
    interest String,
    PRIMARY KEY(user_uuid, interest)
) ENGINE = MergeTree()
ORDER BY (user_uuid, interest);
```

Questa tabella serve a collegare gli utenti ai loro interessi, consentendo di scegliere le attività per cui generare i messaggi in base agli interessi.

- **messageFromLLM**: Memorizza i messaggi pubblicitari generati dall'LLM e successivamente pubblicati sul topic_G kafka_G.

```
CREATE TABLE nearyou.messageTableKafka
(
    user_uuid UUID,
    activity_uuid UUID,
    message_uuid UUID,
    message String,
    activityLatitude Float64,
    activityLongitude Float64,
    creationTime String,
    userLatitude Float64,
    userLongitude Float64
) ENGINE = Kafka()
SETTINGS
    kafka_broker_list = 'kafka:9092',
    kafka_topic_list = 'MessageElaborated',
    kafka_group_name = 'clickhouseConsumerMessage',
    kafka_format = 'JSONEachRow';
```

Questa tabella funziona da consumer per il topic_G Kafka_G "MessageElaborated", permettendo di ricevere e memorizzare i messaggi pubblicitari generati dall'LLM in tempo reale.

```
CREATE TABLE nearyou.messageTable
(
    user_uuid UUID,
    activity_uuid UUID,
    message_uuid UUID,
    message String,
    activityLatitude Float64,
    activityLongitude Float64,
    creationTime String,
```

```
10          userLatitude Float64,
11          userLongitude Float64
12      )
13      ENGINE = MergeTree()
14      PARTITION BY toYYYYMM(toDateTime(creationTime)) -- Partizione per mese basato
            sul timestamp di creazione
15      PRIMARY KEY (message_uuid, toStartOfMinute(toDateTime(creationTime)),
            creationTime)
16      TTL toDateTime(creationTime) + INTERVAL 1 MONTH -- I dati saranno conservati
            per 1 mese
17      SETTINGS index_granularity = 8192;
```

Questa tabella serve da storage per i messaggi pubblicitari generati e pubblicati sul topic$_G$ Kafka$_G$. Contiene informazioni dettagliate dei messaggi, inclusi gli ID degli utenti e delle attività, delle coordinate geografiche e il timestamp di creazione.

- **positionFromSimulator**: Memorizza le posizioni degli utenti generate dal simulatore e pubblicate sul topic$_G$ kafka$_G$.

```
1       CREATE TABLE nearyou.positionsKafka (
2           user_uuid UUID,
3           latitude Float64,
4           longitude Float64,
5           received_at String
6       ) ENGINE = Kafka()
7       SETTINGS
8           kafka_broker_list = 'kafka:9092',
9           kafka_topic_list = 'SimulatorPosition',
10          kafka_group_name = 'clickhouseConsumePositions',
11          kafka_format = 'JSONEachRow';
```

Questa tabella funziona da consumer per il topic$_G$ Kafka$_G$ "SimulatorPosition", permettendo di ricevere e memorizzare le posizioni degli utenti in tempo reale.

```
1       CREATE TABLE nearyou.positions
2       (
3           user_uuid UUID,
4           latitude Float64,
5           longitude Float64,
6           received_at String
7       )
8       ENGINE = MergeTree()
9       PARTITION BY toYYYYMM(toDateTime(received_at)) -- Partizioniamo per mese in
            base al campo received_at
10      PRIMARY KEY (user_uuid, toStartOfMinute(toDateTime(received_at)), received_at)
11      TTL toDateTime(received_at) + INTERVAL 1 MONTH -- I dati vengono conservati per
             un mese
12      SETTINGS index_granularity = 8192;
```

Questa tabella funziona da storage per i dati di posizione degli utenti e viene alimentata dalla tabella Kafka$_G$ positionsKafka.

- **sensor**: Memorizza informazioni sui sensori utilizzati per raccogliere i dati di posizione.

```
1       CREATE TABLE nearyou.sensor
2       (
3           sensor_uuid UUID PRIMARY KEY,
4           is_occupied Boolean DEFAULT false
5       ) ENGINE = MergeTree()
6       ORDER BY sensor_uuid;
```

Questa tabella contiene informazioni sui sensori utilizzati per raccogliere i dati di posizione degli utenti. Ogni sensore ha un UUID univoco e uno stato di occupazione che può essere aggiornato in tempo reale. Ogni sensore viene assegnato regolarmente a un utente, e il suo stato di occupazione viene aggiornato in base alla posizione dell'utente.

- **tuttiInteressi**: Memorizza un elenco di tutti gli interessi disponibili nel sistema$_G$.

```
CREATE TABLE nearyou.interest(
    interest String,
    PRIMARY KEY(interest)
) ENGINE = MergeTree()
ORDER BY (interest);
```

Questa tabella contiene un elenco di tutti gli interessi disponibili nel sistema$_G$. Ogni interesse è rappresentato da una stringa e viene utilizzato per associare le preferenze degli utenti alle attività pubblicitarie.

- **utente**: Memorizza informazioni sui profili utente.

```
CREATE TABLE nearyou.user(
    user_uuid UUID,
    assigned_sensor_uuid UUID NULL,
    name String,
    surname String,
    email String,
    gender String,
    birthdate Date DEFAULT toDate(now()),
    civil_status String,
    PRIMARY KEY(user_uuid)
) ENGINE = MergeTree()
ORDER BY user_uuid;
```

Questa tabella contiene informazioni sui profili utente, inclusi nome, cognome, email, genere e stato civile. Ogni utente ha un UUID univoco e può essere associato a un sensore specifico.

4.5.5.2.1 Index Granularity

```
SETTINGS index_granularity = 8192;
```

L'opzione `index_granularity` in Clickhouse$_G$ determina la dimensione del blocco per l'indicizzazione dei dati. Questo parametro definisce quante righe appartengono a ciascun blocco indice. Con un valore di 8192, Clickhouse$_G$ crea un indice per ogni blocco di 8192 righe, memorizzando i valori minimi e massimi delle colonne di ordinamento in ogni blocco. Tale meccanismo permette al motore di saltare rapidamente blocchi di dati non pertinenti durante l'esecuzione delle query$_G$, ottimizzando così le prestazioni.

4.5.5.2.2 Partizionamento

In questo progetto$_G$ il partizionamento viene fatto su base temporale. In questo caso le tabelle positions e messageTable sono partizionate per giorno:

```
PARTITION BY toYYYYMMDD(received_at)
```

Questa strategia di partizionamento:

- Facilita l'eliminazione automatica dei dati storici con TTL;
- Migliora le performance delle query$_G$ che filtrano per intervalli temporali;
- Consente una gestione efficiente dello storage.

4.5.5.2.3 Time-To-Live (TTL)

Il meccanismo TTL in Clickhouse$_G$ è una funzionalità essenziale per la gestione di grandi volumi di dati, consentendo un controllo automatico sul loro ciclo-di-vita$_G$ senza necessità di interventi manuali o script esterni. Clickhouse$_G$ integra il TTL direttamente nel processo$_G$ di merging di MergeTree, offrendo tre diverse tipologie di gestione:

- **TTL a livello di tabella**: Rimuove intere righe dopo un determinato periodo

```
ALTER TABLE positions
MODIFY TTL received_at + INTERVAL 30 DAY;
```

- **TTL a livello di colonna**: Permette l'anonimizzazione graduale

```
ALTER TABLE positions
MODIFY COLUMN precise_location
TTL received_at + INTERVAL 7 DAY SET NULL;
```

- **TTL multi-fase con storage tiering**: Ottimizza i costi di archiviazione

```
ALTER TABLE positions
MODIFY TTL
    received_at + INTERVAL 7 DAY TO VOLUME 'hot',
    received_at + INTERVAL 30 DAY TO VOLUME 'cold',
    received_at + INTERVAL 90 DAY DELETE;
```

Il TTL viene applicato durante le operazioni di merge$_G$:

1. Durante il merge$_G$, il motore verifica la condizione TTL per ogni riga o colonna;
2. Se la condizione è soddisfatta (ad es. il dato è più vecchio del limite), viene applicata l'azione corrispondente;
3. Se tutte le righe in una partizione vengono eliminate dal TTL, l'intera partizione viene rimossa.

In questo progetto$_G$ il TTL viene usato solo a livello di tabella per eliminare automaticamente i dati più vecchi di 30 giorni: Ad esempio nella tabella delle posizioni:

```
ALTER TABLE positions MODIFY TTL received_at + INTERVAL 30 DAY;
```

Questa strategia bilancia efficacemente le esigenze di performance con l'ottimizzazione dei costi di storage, mantenendo uno storico per un periodo abbastanza lungo.

4.5.5.2.4 Materialized Views

Le materialized view rappresentano un elemento fondamentale nell'architettura di questo progetto$_G$, consentendo di trasformare in tempo reale i dati provenienti dalle tabelle Kafka$_G$ in tabelle Clickhouse$_G$ ottimizzate per le query$_G$. I benefici principali dell'utilizzo delle materialized view sono:

- **Prestazioni ottimizzate:** Poiché i dati vengono pre-aggregati e trasformati al momento dell'inserimento, le query$_G$ risultano più veloci, riducendo il carico computazionale durante l'esecuzione della query$_G$;
- **Persistenza automatica:** Le view accumulano continuamente i dati dai flussi Kafka$_G$, assicurando che ogni nuovo messaggio sia immediatamente reso disponibile in Clickhouse$_G$ per ulteriori analisi;
- **Aggiornamento in tempo reale:** Non appena nuovi messaggi arrivano in Kafka$_G$, la materialized view li elabora automaticamente e li inserisce nella tabella di destinazione, mantenendo sempre aggiornato lo stato dei dati;

- **Integrazione trasparente con Kafka$_G$:** Utilizzando l'engine Kafka$_G$ per definire le tabelle sorgente, come detto nella sezione di Kafka$_G$ Clickhouse$_G$ può "iscriversi" automaticamente ai topic$_G$ Kafka$_G$. In questo modo, l'engine si occuperà di gestire i dati e dopo dato che saranno disponibili anche nella materialized view si potranno trasformare e trattare coem se fosse una tabella normale.

Ecco alcuni esempi:

```
CREATE MATERIALIZED VIEW nearyou.mv_positions TO nearyou.positions
AS
SELECT
    user_uuid,
    latitude,
    longitude,
    received_at
FROM nearyou.positionsKafka;
```

```
CREATE MATERIALIZED VIEW nearyou.mv_messageTable TO nearyou.messageTable
AS
SELECT
    user_uuid,
    activity_uuid,
    message_uuid,
    message,
    activityLatitude,
    activityLongitude,
    creationTime,
    userLatitude,
    userLongitude
FROM nearyou.messageTableKafka;
```

In pratica, una volta che una materialized view è definita, Clickhouse$_G$ continua a monitorare la tabella Kafka$_G$ di origine e, ogni volta che arrivano nuovi dati, li elabora e li scrive nella tabella target (ad es. `positions` o `messageTable`). Questo meccanismo permette di sfruttare il flusso dati di Kafka$_G$ senza perdere i vantaggi di una storage engine tradizionale, garantendo l'accesso in tempo reale a dati puliti e pre-aggregati.

4.5.5.2.5 Funzionalità geospaziali

In questo progetto$_G$ sono state usate le funzioni geospaziali offerte da Clickhouse$_G$ per calcolare distanze e rilevare la prossimità tra punti geografici. **geoDistance**: Calcola la distanza in metri tra due punti geografici
Ad esempio un implementazione della funzione `geoDistance` si trova nella query$_G$ dei messaggi, dove questa funzione è stata usata per calcolare la distanza tra la posizione dell'utente e la posizione originale del messaggio. Ecco il frammento della query$_G$ in cui viene usata questa funzione:

```
WHERE
    geoDistance(m.activityLongitude, m.activityLatitude, p.longitude, p.latitude) < 300
```

4.5.6 Grafana

Grafana$_G$ è una piattaforma di visualizzazione e analisi dati utilizzata in questo progetto$_G$ per rappresentare graficamente le informazioni raccolte dal sistema$_G$ e consentire il monitoraggio in tempo reale delle attività.

4.5.6.1 Utenti

Innanzitutto per fare l'accesso è necessario un account Grafana$_G$. Dato che lo scopo di questo progetto$_G$ è quello di creare una dashboard$_G$ da amministratore la scelta è stata quella di usare l'account admin per l'accesso. Le credenziali per questo utente sono:

- **Username**: admin;
- **Password**: admin.

Questo utente ha accesso completo a entrambe le dashboard$_G$ del progetto$_G$.

4.5.6.2 Dashboards

La visualizzazione in questo progetto$_G$ è stata separata in due dashboard$_G$ differenti:

. Dashboard$_G$ generale: Si occupa di una visuale complessiva del sistema$_G$, mostrando le ultime posizioni e gli ultimi messaggi di ogni singolo utente. Inoltre si vedono anche tutte le attività commerciali disposte per la mappa; In questa mappa è anche presente una leaderboard che mostra le attività più popolari in base al numero di messaggi pubblicitari generati;

. Dashboard$_G$ specifica: Questa dashboard$_G$ è dedicata a un singolo utente e mostra ogni singola posizione che compone il suo percorso$_G$, ogni singolo messaggio pubblicitario generato per questo utente e tutte le attività presenti sulla mappa. È inoltre presente una tabella che mostra tutti i dati anagrafici dell'utente.

4.5.6.3 Dashboard generale

La dashboard$_G$ generale è quindi divisa in due widget:

- **Mappa geospaziale**: Un pannello$_G$ interattivo che visualizza una mappa OpenStreetMap su cui vengono mostrate le seguenti informazioni:
  . Marker$_G$ verde scuro a cerchio che rappresentano solo l'ultima posizione per ogni singolo utente;
  . Marker$_G$ arancioni quadrati che mostrano l'ultimo messaggio pubblicitario generato per ogni singolo utente;
  . Marker$_G$ rossi a cerchio che indicano le attività commerciali nel territorio.
- **Tabella delle attività più popolari**: Una classifica in tempo reale delle attività commerciali ordinate per numero di messaggi pubblicitari generati, con le seguenti informazioni:
  . Nome dell'attività;
  . Categoria o tipologia dell'attività;
  . Indirizzo dell'attività;
  . Descrizione dell'attività;
  . Conteggio totale dei messaggi generati.

Facendo hover o premendo sulla posizione di un utente comparirà il popup con le informazioni dell'utente. Fra queste informazioni sarà presente anche un link che porta l'amministratore nella dashboard$_G$ specifica che fa riferimento all'utente su cui ha premuto in origine.

4.5.6.3.1 Dashboard specifica

La dashboard$_G$ specifica è dedicata all'analisi dettagliata di un singolo utente, identificato tramite una variabile della dashboard$_G$ `user_id`. Questa dashboard$_G$ permette di:

- Evidenziare con marker$_G$ specifici:
  . Visualizzare l'intero percorso$_G$ storico dell'utente sulla mappa, con punti che rappresentano le posizioni registrate nel tempo (viola);
  . Distinguere la prima e l'ultima posizione registrata dalle altre posizioni (verde scuro e chiaro);
  . Le attività commerciali vicine al percorso$_G$ (rosso);
  . I messaggi pubblicitari generati lungo il percorso$_G$ (arancione).
- Analizzare i dati temporali delle posizioni e dei messaggi correlati;
- **Tabella dei dati anagrafici dell'utente**: Una tabella che mostra le seguenti informazioni:

. Nome dell'utente;
. Cognome dell'utente;
. Indirizzo email;
. Genere dell'utente;
. Data di nascità;
. Stato Civile;
. Interessi dell'utente.

4.5.6.4 Querying Clickhouse

L'integrazione tra Grafana$_G$ e Clickhouse$_G$ è realizzata tramite query$_G$ SQL$_G$ ottimizzate per le performance. Di seguito sono riportate le principali query$_G$ utilizzate nelle dashboard$_G$, che consentono di estrarre e visualizzare i dati in tempo reale.

- **Query per la dashboard$_G$ generale:** La dashboard$_G$ generale contiene query$_G$ per visualizzare dati aggregati che consentono di monitorare l'intera piattaforma;
  - **Posizioni attuali degli utenti:** Questa query$_G$ recupera l'ultima posizione nota per ciascun utente registrato nel sistema$_G$. Utilizza una subquery con ROW_NUMBER() per selezionare solo la posizione più recente per ciascun sensore.

```
SELECT
    user_uuid,
    latitude,
    longitude,
    received_at
FROM (
    SELECT
    user_uuid,
    latitude,
    longitude,
    received_at,
    ROW_NUMBER() OVER (PARTITION BY user_uuid ORDER BY received_at DESC
        ) AS row_num
    FROM "nearyou"."positions"
)
WHERE row_num = 1;
```

  - **Messaggi pubblicitari recenti:** Questa query$_G$ complessa recupera i messaggi pubblicitari più recenti per ciascun utente, includendo informazioni contestuali come il nome dell'utente e dell'attività. Filtra i messaggi in base alla prossimità utilizzando la funzione geospaziale `geoDistance`.

```
SELECT
    m.userLatitude AS latitude,
    m.userLongitude AS longitude,
    m.creationTime AS creazione_time,
    u.name AS userName,
    u.surname AS userSurname,
    a.name AS activityName,
    m.message AS message
FROM (
    SELECT
    user_uuid,
    message_uuid,
    message,
    activity_uuid,
    activityLatitude,
    activityLongitude,
    creationTime,
```

```
                userLatitude,
                userLongitude,
                ROW_NUMBER() OVER (PARTITION BY user_uuid ORDER BY toDateTime(
                    creationTime) DESC) AS rn
                FROM nearyou.messageTable
            ) AS m
            INNER JOIN nearyou.user u ON m.user_uuid = u.user_uuid
            INNER JOIN nearyou.activity a ON m.activity_uuid = a.activity_uuid
            INNER JOIN (
                SELECT
                user_uuid,
                latitude,
                longitude,
                received_at,
                ROW_NUMBER() OVER (PARTITION BY user_uuid ORDER BY received_at DESC
                    ) AS rn
                FROM nearyou.positions
            ) AS p ON u.assigned_sensor_uuid = p.user_uuid AND p.rn = 1
            WHERE
                m.rn = 1
                AND geoDistance(m.activityLongitude, m.activityLatitude, p.
                    longitude, p.latitude) < 300
            ORDER BY m.creationTime DESC;
```

- **Lista delle attività:** Questa query$_G$ recupera l'elenco completo delle attività (punti di interesse) dal database$_G$.

```
            SELECT * FROM "nearyou"."activity" LIMIT 1000
```

- **Attività più popolari:** Questa query$_G$ genera una classifica delle attività commerciali in base al numero di messaggi pubblicitari generati per ciascuna di esse.

```
            SELECT
                a.name AS nome_attivita,
                a.address AS indirizzo,
                a.type AS tipologia,
                a.description as descrizione,
                COUNT(m.message_uuid) AS numero_messaggi
            FROM nearyou.activity a
            INNER JOIN nearyou.messageTable m ON a.activity_uuid = m.activity_uuid
            GROUP BY a.activity_uuid, a.name, a.type, a.address, a.description
            HAVING COUNT(m.message_uuid) > 0
            ORDER BY numero_messaggi DESC
```

- **Query per la dashboard$_G$ specifica utente:** La dashboard$_G$ specifica per utente utilizza query$_G$ che filtrano i dati in base all'utente selezionato, identificato tramite la variabile `$user_id`;

  - **Storico posizioni dell'utente:** Questa query$_G$ recupera l'intero storico delle posizioni per un utente specifico, consentendo di visualizzare il suo percorso$_G$ completo sulla mappa.

```
            SELECT * FROM nearyou.positions
            WHERE user_uuid = toUUID('${user_id}')
            LIMIT 1000
```

  - **Prima e ultima posizione dell'utente:** Questa query$_G$ utilizza UNION ALL per combinare due subquery che identificano la prima e l'ultima posizione registrata per l'utente, consentendo di evidenziarle sulla mappa.

```
(
    SELECT *
    FROM nearyou.positions
    WHERE user_uuid = toUUID('${user_id}')
    ORDER BY received_at ASC
    LIMIT 1
)
UNION ALL
(
    SELECT *
    FROM nearyou.positions
    WHERE user_uuid = toUUID('${user_id}')
    ORDER BY received_at DESC
    LIMIT 1
);
```

- **Messaggi per l'utente specifico:** Questa query$_G$ recupera i messaggi pubblicitari generati per un utente specifico, includendo informazioni contestuali come le coordinate dell'utente e dell'attività al momento della generazione.

```
SELECT
    m.userLatitude,
    m.userLongitude,
    m.creationTime,
    u.name,
    u.surname,
    a.name,
    m.message
FROM nearyou.messageTable m
INNER JOIN nearyou.user u ON m.user_uuid = u.user_uuid
INNER JOIN nearyou.activity a ON m.activity_uuid = a.activity_uuid
WHERE u.assigned_sensor_uuid = toUUID('${user_id}')
LIMIT 1000;
```

- **Dati anagrafici dell'utente:** Questa query$_G$ recupera i dati anagrafici dell'utente e i suoi interessi, utilizzando la funzione ARRAY_AGG per aggregare gli interessi multipli in un array.

```
SELECT
    u.name,
    u.surname,
    u.email,
    u.gender,
    u.birthdate,
    u.civil_status,
    ARRAY_AGG(ui.interest) AS interests_json
FROM
    nearyou.user u
JOIN
    nearyou.user_interest ui ON u.user_uuid = ui.user_uuid
WHERE
    u.assigned_sensor_uuid = toUUID('${user_id}')
GROUP BY
    u.name, u.surname, u.email, u.gender, u.birthdate, u.civil_status,
        u.user_uuid;
```

Queste query$_G$ rappresentano un insieme completo di interrogazioni utilizzate per alimentare le dashboard$_G$ Grafana$_G$ e consentire la visualizzazione di informazioni in tempo reale sulla posizione degli utenti, le attività commerciali e i messaggi pubblicitari generati.

4.5.6.5 Variabili dashboard

In questo progetto$_G$ è stata usata una variabile dashboard$_G$ per filtrare i dati in base all'utente selezionato.

Questa variabile è definita come `user_id` e viene utilizzata per personalizzare le query$_G$ SQL$_G$ in modo da visualizzare solo i dati pertinenti a un singolo utente. Ad esempio, la variabile viene utilizzata nella query$_G$ per le posizioni correnti dell'utente:

```
SELECT *
FROM nearyou.positions
WHERE user_uuid = toUUID('${user_id}')
```

4.5.6.6 Trasformazioni e array interessi

Per la tabella dell'utente nella tabella della dashboard$_G$ specifica per far vedere gli interessi dell'utente è stata usata la funzione `ARRAY_AGG` per aggregare gli interessi dell'utente in un array JSON$_G$. Questo approccio consente di visualizzare gli interessi in un formato strutturato e facilmente leggibile ma comparivano tutti in una singola cella. Quindi sono state usate delle trasformazioni di Grafana$_G$ per visualizzare un interesse per cella e rinominare la colonna in "Interesse0", "Interesse1", ecc.

- **Estrazione campi**: Utilizzando la trasformazione "extractFields" per estrarre i valori dall'array JSON$_G$:

```
{
    "id": "extractFields",
    "options": {
        "delimiter": ",",
        "format": "auto",
        "jsonPaths": [
                {
                "alias": "interest",
                "path": ""
                }
            ],
        "keepTime": false,
        "replace": false,
        "source": "interests_json"
    }
}
```

- **Organizzazione**: Per nascondere il campo originale dell'array e mantenere solo i campi estratti:

```
{
    "id": "organize",
    "options": {
        "excludeByName": {
            "interests_json": true
        },
        "includeByName": {},
        "indexByName": {},
        "renameByName": {}
    }
}
```

- **Ridenominazione colonne**: Usando la trasformazione "renameByRegex" per applicare un pattern di ridenominazione:

```
{
    "id": "renameByRegex",
    "options": {
        "regex": "^(\\d+)$",
        "renamePattern": "interesse $1"
```

```
            }
        }
```

4.5.6.7 Connettore Clickhouse

Grafana$_G$ si integra con Clickhouse$_G$ attraverso un connettore nativo specifico, configurato come datasource all'interno della piattaforma:

```
{
    "name": "ClickHouse",
    "type": "grafana-clickhouse-datasource",
    "uid": "ee5wustcp8zr4b",
    "jsonData": {
        "defaultDatabase": "nearyou",
        "port": 9000,
        "host": "clickhouse",
        "username": "default",
        "tlsSkipVerify": false
    },
    "secureJsonData": {
        "password": "pass"
    }
}
```

Il connettore nativo offre vantaggi significativi rispetto alle alternative generiche:

- **Porta con sé tutte le funzionalità di Clickhouse$_G$**:
  . Supporto per query$_G$ SQL$_G$ avanzate;
  . Funzioni geospaziali integrate;
  . Ottimizzazione automatica delle performance.
- **Configurazione semplice e veloce**:
  . Configurazione del datasource in pochi passaggi;
  . Nessuna necessità di scrivere codice personalizzato per l'integrazione;
  . Possibilità di configurare il provisioning automatico.

La visualizzazione in tempo reale è garantita attraverso un intervallo di aggiornamento automatico configurato a 10 secondi:

```
{
    "refresh": "10s"
}
```

Questa impostazione assicura che i dati visualizzati nelle dashboard$_G$ siano costantemente aggiornati, permettendo di monitorare in tempo reale:

- Spostamenti degli utenti;
- Generazione di nuovi messaggi pubblicitari;
- Statistiche di generazione di messaggi per le attività più popolari.

4.5.6.8 Provisioning automatico

Il provisioning automatico di Grafana$_G$ è implementato attraverso file di configurazione YAML che vengono caricati all'avvio del container$_G$, garantendo la disponibilità immediata di datasource e dashboard$_G$ preconfiguraate.

- **Provisioning del datasource Clickhouse$_G$**:

```
apiVersion: 1
datasources:
  - name: Clickhouse
    type: grafana-clickhouse-datasource
    uid: "ee5wustcp8zr4b"
    jsonData:
      defaultDatabase: nearyou
      port: 9000
      host: clickhouse
      username: default
      tlsSkipVerify: false
    secureJsonData:
      password: pass
```

- **Provisioning delle dashboard**$_G$:

```
apiVersion: 1
providers:
  - name: "Dashboard provider"
    orgId: 1
    type: file
    disableDeletion: false
    updateIntervalSeconds: 10
    options:
      path: /var/lib/grafana/dashboards
      foldersFromFilesStructure: true
```

Le dashboard$_G$ sono definite in file JSON$_G$ che vengono copiati nella directory specificata nel provider durante la build$_G$ dell'immagine Docker$_G$, garantendo che siano immediatamente disponibili all'avvio del container$_G$ Grafana$_G$. Questa configurazione automatizzata elimina la necessità di setup manuale e assicura la coerenza dell'ambiente di visualizzazione in tutti i deployment del sistema$_G$.

4.5.7 Best practices architetturali

4.5.7.1 PEP8 - Stile di codifica Python

Il progetto$_G$ aderisce alle linee guida PEP8, lo standard di stile di codifica Python$_G$, garantendo coerenza e leggibilità. Questo include:

- Indentazione di 4 spazi (non tab);
- Lunghezza massima delle linee di 79 caratteri;
- Spaziatura coerente attorno agli operatori;
- Convenzioni di nomenclatura: `snake_case` per variabili e funzioni, `PascalCase` per classi;
- Docstring per moduli, classi e funzioni.

Il rispetto di PEP8 è verificato attraverso l'uso di linting automatizzato, come evidenziato dal badge PyLint nel README del progetto$_G$ (punteggio 7.7/10.0).

4.5.7.2 Principi SOLID

L'architettura del software è progettata seguendo i principi SOLID:

- **Single Responsibility Principle**: L'architettura di questo progetto$_G$ rivela una comprensione profonda del Single Responsibility Principle (SRP), uno dei capisaldi dei principi SOLID. Ciò che colpisce immediatamente è il modo in cui l'intero sistema$_G$ sia stato concepito come un ecosistema di componenti indipendenti ma interconnessi, ognuno con una responsabilità chiara e ben definita;

  All'interno di ciascun modulo, l'aderenza al SRP continua a manifestarsi nella progettazione dei repository$_G$. Ogni repository$_G$ è dedicato ad un singolo tipo di entità. Il `ClickhouseUserRepository` si concentra esclusivamente sui dati degli utenti, mentre l'`ClickhouseActivityRepository` gestisce

solo le attività. Questa specializzazione permette di incapsulare tutta la logica relativa ad un'entità in un unico contenitore, rendendo il codice più prevedibile e facilitando i futuri interventi di manutenzione.

Particolarmente interessante è l'implementazione del pattern Strategy nel `SimulationService`. L'interfaccia IPositionSimulationStrategy definisce un contratto che viene PoI$_G$ implementato da classi specifiche come `BycicleSimulationStrategy`. Questo approccio consente di modificare gli algoritmi di simulazione senza dover intervenire sul resto del codice, incarnando perfettamente il principio di responsabilità singola.

I servizi specializzati come `GroqLLMService` e `UserSensorService` testimoniano ulteriormente l'impegno verso il SRP. Il primo si occupa esclusivamente dell'interazione con i modelli linguistici, mentre il secondo gestisce solo le relazioni tra sensori e utenti. Questa specializzazione rende il codice non solo più leggibile, ma anche più resiliente ai cambiamenti.

Nei processori Flink$_G$ osserviamo lo stesso principio applicato alle operazioni di streaming. Ogni processore ha un compito specifico: trasformare posizioni in messaggi, validare messaggi o filtrare quelli già visualizzati. Questa "catena di responsabilità" permette di ragionare su ciascun passaggio indipendentemente, facilitando debugging e testing;

- **Open/Closed Principle**: Al cuore dell'architettura troviamo un sistema$_G$ di interfacce che stabilisce contratti chiari tra i diversi componenti. La presenza di interfacce come `IUserRepository`, `IMessageRepository` e `IActivityRepository` consente di definire comportamenti astrati che possono essere implementati in molteplici modi. Questa scelta progettuale permette di introdurre nuove implementazioni (come potrebbe essere un `PostgresUserRepository` accanto all'esistente `ClickhouseUserRepository`) senza alterare il codice client che interagisce con queste interfacce.

  Particolarmente significativa è l'implementazione del pattern Strategy nel modulo di simulazione. L'interfaccia `IPositionSimulationStrategy` definisce un contratto generico per gli algoritmi di simulazione delle posizioni, mentre implementazioni concrete come `BycicleSimulationStrategy` ne forniscono realizzazioni specifiche. Questo design consente di introdurre nuove strategie di simulazione, magari per automobili, pedoni, senza modificare il codice esistente. Il sistema$_G$ può accogliere nuove funzionalità semplicemente estendendo il repertorio di strategie disponibili.

  Il paradigma dell'elaborazione streaming adottato con Apache Flink$_G$ manifesta in modo efficace il principio Open/Closed. La catena di processori come `PositionToMessageProcessor`, `FilterMessageValidator` e `FilterMessageAlreadyDisplayed` è costruita in modo tale che nuovi comportamenti di elaborazione possano essere introdotti aggiungendo nuovi processori, senza alterare quelli esistenti. Ogni processore ha una responsabilità ben definita e la pipeline di elaborazione può essere estesa aggiungendo nuovi anelli alla catena.

  La struttura dei DTO (Data Transfer Objects) come `UserDTO, ActivityDTO e MessageDTO` è progettata per essere facilmente estensibile. Nuovi attributi o proprietà possono essere aggiunti a queste classi senza influenzare il codice che utilizza solo le proprietà esistenti.

  I test$_G$ unitari, infine, rivelano come l'architettura faciliti l'estensibilità: l'uso di mock e stub per le dipendenze dimostra che i componenti sono progettati per accettare implementazioni alternative delle loro dipendenze, un prerequisito essenziale per un sistema$_G$ che rispetta il principio Open/-Closed;

- **Liskov Substitution Principle**: Le classi derivate possono sostituire le classi base senza alterare il comportamento, consentendo l'uso intercambiabile delle diverse strategie di simulazione.

  Questo principio trova una chiara applicazione nel pattern Strategy adottato. La relazione tra `IPositionSimulationStrategy` e `BycicleSimulationStrategy` ne è un esempio esemplare:
  l'interfaccia definisce un contratto preciso per la simulazione delle posizioni, mentre la strategia per biciclette lo implementa fedelmente. Questo design garantisce che nuove strategie, come la simulazione di automobili, pedoni o altri mezzi di trasporto, possano essere integrate senza modificare il codice client. Di conseguenza, il sistema$_G$ rimane flessibile ed estensibile, assicurando che il codice che utilizza una strategia di simulazione continui a funzionare correttamente indipendentemente dall'implementazione specifica;

- **Interface Segregation Principle**: Le interfacce del sistema$_G$ sono progettate per essere specifiche e mirate, garantendo una chiara separazione delle responsabilità. Un esempio evidente è dato dai processori del flusso Flink$_G$, come `PositionToMessageProcessor`, `FilterMessageValidator`

e `FilterMessageAlreadyDisplayed`. Ciascuno di essi implementa un'interfaccia focalizzata esclusivamente sulla propria funzione all'interno della pipeline di elaborazione. Questo approccio favorisce la modularità, consentendo a ogni componente di svolgere il proprio compito senza essere appesantito da metodi superflui o non pertinenti alla sua responsabilità;

- **Dependency Inversion Principle**: I componenti del sistema$_G$ seguono il principio DIP, affidandosi ad astrazioni anziché a implementazioni concrete. Un esempio chiaro di questo approccio è il pattern Strategy adottato nel *SimulationService*. I moduli responsabili dell'orchestrazione della simulazione dipendono dall'interfaccia `IPositionSimulationStrategy` anziché da specifiche implementazioni come `BycicleSimulationStrategy`. Questa separazione consente di sostituire facilmente le strategie di simulazione, mantenendo il sistema$_G$ flessibile ed estensibile senza necessità di modificare il codice esistente.

  Particolarmente sofisticata è l'applicazione del DIP nei servizi di comunicazione con sistemi esterni. Il `GroqLLMService`, ad esempio, non espone i dettagli dell'implementazione specifica del provider LLM$_G$ ai suoi client. Invece, offre un'interfaccia astratta che incapsula l'interazione con il modello linguistico. Questo design consente di sostituire il provider sottostante – magari passando da Groq$_G$ ad altri fornitori – senza ripercussioni sui componenti che utilizzano questo servizio.

4.5.7.3 Dependency Injection

La Dependency Injection (DI) è un pattern di progettazione software che implementa il principio di Inversione del Controllo (IoC). Questo pattern permette di separare la creazione di un oggetto dal suo utilizzo, consentendo di "iniettare" le dipendenze nei componenti anziché crearle all'interno di essi.
Nel progetto$_G$ NearYou, questo pattern è stato implementato per ottenere un'architettura software modulare, testabile e manutenibile. La DI ha permesso di gestire efficacemente le complesse dipendenze tra i vari componenti del sistema$_G$, facilitando l'integrazione fra i moduli di simulazione, elaborazione eventi e storage.

4.5.7.3.1 Constructor Injection nel Progetto

La tecnica di Constructor Injection è stata scelta come approccio principale per implementare la DI in questo progetto$_G$. Con questa tecnica, le dipendenze vengono fornite attraverso i parametri del costruttore della classe.
I vantaggi della Constructor Injection sono:

- **Dipendenze obbligatorie**: Il costruttore richiede esplicitamente tutte le dipendenze necessarie;
- **Immutabilità**: Le dipendenze possono essere memorizzate in campi privati finali;
- **Testabilità**: Facilita la sostituzione delle dipendenze reali con mock durante i test$_G$;
- **Visibilità**: Rende esplicite le dipendenze di una classe.

4.6 Implementazione nel FlinkProcessor

Un esempio significativo di Constructor Injection nel progetto$_G$ è la classe `FlinkJobManager`, responsabile della configurazione e gestione del flusso di elaborazione dei dati:

Listing 1: Esempio di Constructor Injection in FlinkJobManager

```
class FlinkJobManager:
    def __init__(self,
                 streaming_env_istance : StreamExecutionEnvironment,
                 map_function_implementation : MapFunction,
                 filter_validator_implementation : FilterFunction,
                 filter_function_implementation : FilterFunction,
                 position_receiver_istance: IPositionReceiver,
                 message_sender_istance: IMessageWriter
                 ):
        self.__streaming_env = streaming_env_istance
```

```
        self.__populated_datastream = self.__streaming_env.from_source(
            position_receiver_istance.get_position_receiver(),
                                                              WatermarkStrategy.
                                                                  for_monotonous_timestamps()
                                                                  ,
                                                              "Positions Source")
        self.__keyed_stream = self.__populated_datastream.key_by(lambda x: x[0], key_type=
            Types.STRING())
        self.__validated_stream = self.__keyed_stream.filter(filter_validator_implementation
            )
        self.__mapped_stream = self.__validated_stream.map(map_function_implementation,
                                                           output_type=KafkaWriterConfiguration().
                                                               row_type_info_message)
        self.__filtered_stream = self.__mapped_stream.filter(filter_function_implementation)
        self.__filtered_stream.sink_to(message_sender_istance.get_message_writer())
```

Questo esempio mostra come il costruttore riceva tutte le dipendenze necessarie come parametri, ognuna tipizzata con la relativa interfaccia. Questo approccio garantisce che:

1. Tutte le dipendenze siano fornite all'inizializzazione dell'oggetto;
2. Le concrete implementazioni possano essere facilmente sostituite;
3. La classe non debba conoscere i dettagli di implementazione delle sue dipendenze.

4.6.0.0.1 Implementazione nel SimulationModule

Nel modulo di simulazione, il pattern di Constructor Injection è utilizzato ampiamente, come nel caso di `SensorFactory`:

Listing 2: Esempio di Constructor Injection in SensorFactory

```
class SensorFactory:
    def __init__(self, sensor_repository: ISensorRepository, user_repository:
        IUserRepository):
        self.__sensor_repository = sensor_repository
        self.__user_repository = user_repository

    def create_gps_sensor(self, position_sender: PositionSender,
                          simulation_strategy: IPositionSimulationStrategy) -> GpsSensor:
        # Implementazione che utilizza le dipendenze iniettate
```

4.6.0.0.2 Integrazione con il Sistema di Testing

La Dependency Injection tramite costruttore ha semplificato notevolmente l'implementazione dei test$_G$ unitari. Utilizzando librerie come `unittest.mock`, è possibile sostituire facilmente le dipendenze reali con mock:

Listing 3: Test di un componente con Constructor Injection

```
def setUp(self):
    self.mock_sensor_repository = Mock(spec=ISensorRepository)
    self.mock_user_repository = Mock(spec=IUserRepository)
    self.mock_position_sender = Mock(spec=PositionSender)
    self.mock_simulation_strategy = Mock(spec=IPositionSimulationStrategy)
    self.mock_user_sensor_service = Mock(spec=UserSensorService)

    self.patcher = patch('Models.SensorFactory.UserSensorService',
                         return_value=self.mock_user_sensor_service)
    self.patcher.start()

```

```
12      self.sensor_factory = SensorFactory(self.mock_sensor_repository, self.
            mock_user_repository)
```

Questo approccio ha contribuito a raggiungere una copertura di test$_G$ vicina al 100%, come evidenziato dai badge del progetto$_G$, rendendo il codice più robusto e affidabile.

4.6.0.0.3 Implementazione nella Pratica

La composizione delle dipendenze nel progetto$_G$ avviene tipicamente al punto di ingresso dell'applicazione, come mostrato in `main.py`:

Listing 4: Composizione delle dipendenze in main.py

```
1  json_adapter: PositionJsonAdapter = PositionJsonAdapter()
2  kafka_confluent_adapter: PositionSender = KafkaConfluentAdapter(
3      KafkaConfigParameters(),
4      json_adapter,
5      Producer({'bootstrap.servers': KafkaConfigParameters().bootstrap_servers})
6  )
7
8  graph_wrapper: GraphWrapper = GraphWrapper()
9  bicycle_simulation_strategy: IPositionSimulationStrategy = BycicleSimulationStrategy(
       graph_wrapper)
10
11 # Database connection
12 db_connection = DatabaseConnection(DatabaseConfigParameters())
13 sensor_repository: ISensorRepository = SensorRepository(db_connection)
14 user_repository: IUserRepository = UserRepository(db_connection)
15
16 # Factory
17 sensor_factory: SensorFactory = SensorFactory(sensor_repository, user_repository)
```

5 Architettura di deployment

Da capitolato$_G$ era stato definito l'uso di un'architettura containerizzata e anche valutando alcune alternative risultava comunque la scelta più ragionevole. Al fine di implementare ed eseguire l'intero stack tecnologico ed i componenti del modello architetturale del sistema$_G$ seguendo questa architettura, 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$_G$ del progetto$_G$ oltre che nella sezione 2.3.

5.1 Panoramica dell'infrastruttura

5.1.1 Ambiente Docker dei Componenti Principali

5.1.1.1 Zookeeper Service

- **Descrizione**: Implementa un servizio di coordinamento distribuito essenziale per l'infrastruttura Kafka$_G$. Fornisce sincronizzazione, centralizzazione della configurazione, gestione dei nomi, e leader election per il sistema$_G$ distribuito;
- **Configurazioni**:
  - `hostname: zookeeper` - Nome host assegnato al container$_G$ all'interno della rete Docker$_G$, utilizzato dagli altri servizi per riferirsi a Zookeeper;
  - `container_name: zookeeper` - Nome esplicito assegnato al container$_G$ per facilitarne l'identificazione e la gestione;
  - `image: confluentinc/cp-zookeeper:7.6.0` - Immagine Docker$_G$ ufficiale di Confluent Inc. per Zookeeper, versione 7.6.0, garantendo compatibilità con il broker$_G$ Kafka$_G$ utilizzato;
  - `environment` - Variabili d'ambiente che configurano il comportamento di Zookeeper:
    - `ZOOKEEPER_CLIENT_PORT: 2181` - Porta sulla quale Zookeeper accetta connessioni dai client;
    - `ZOOKEEPER_TICK_TIME: 2000` - Unità di tempo base in millisecondi utilizzata per il timeout e la sincronizzazione.
  - `ports: - 2181:2181` - Mappatura della porta, espone la porta 2181 del container$_G$ alla porta 2181 dell'host per consentire connessioni esterne.
- **Controlli e Disponibilità**:
  - `healthcheck` - Verifica automatica dello stato di salute del servizio:
    - `test: nc -z localhost 2181 || exit -1` - Comando che verifica se la porta 2181 è in ascolto, segnalando un errore in caso contrario;
    - `interval: 10s` - Frequenza di esecuzione del controllo (ogni 10 secondi);
    - `timeout: 5s` - Tempo massimo di attesa per l'esecuzione del controllo;
    - `retries: 3` - Numero di tentativi prima di considerare il controllo fallito.
  - `profiles: ["test","develop","prod"]` - Attivazione del servizio in tutti gli ambienti di esecuzione (test, sviluppo e produzione), indicando la sua importanza fondamentale per l'intero sistema$_G$.

5.1.1.2 Kafka Service

- **Descrizione**: Implementa un broker$_G$ di messaggistica distribuito che gestisce lo scambio di dati tra i componenti dell'applicazione. Fornisce un sistema$_G$ publish-subscribe scalabile che consente la comunicazione asincrona tra il simulatore di posizione e il processore Flink$_G$;
- **Configurazioni**:
  - `image: confluentinc/cp-kafka:7.6.0` - Immagine Docker$_G$ ufficiale di Confluent Inc. per Kafka$_G$, versione 7.6.0;
  - `hostname: kafka`$_G$ - Nome host assegnato al container$_G$ all'interno della rete Docker$_G$;

- `container_name: kafka`$_G$ - Nome esplicito assegnato al container$_G$ per facilitarne l'identificazione;
- `depends_on` - Dipendenze del servizio:
  - `zookeeper: condition: service_healthy` - Attende che Zookeeper sia completamente funzionante prima dell'avvio;
  - `clickhouse: condition: service_started` - Attende l'avvio di Clickhouse$_G$.
- `ports: - 29092:29092` - Espone la porta 29092 per connessioni da applicazioni esterne alla rete Docker$_G$;
- `environment` - Variabili d'ambiente che configurano il comportamento di Kafka$_G$:
  - `KAFKA_BROKER_ID: 1` - Identificatore univoco del broker$_G$ Kafka$_G$;
  - `KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181` - Indirizzo di connessione a Zookeeper;
  - `KAFKA_ADVERTISED_LISTENERS` - Configurazione degli endpoint pubblicizzati per le connessioni interne ed esterne;
  - `KAFKA_LISTENER_SECURITY_PROTOCOL_MAP` - Mappa dei protocolli di sicurezza per i diversi listener;
  - `KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT` - Nome del listener utilizzato per la comunicazione tra broker$_G$;
  - `KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1` - Fattore di replica per il topic$_G$ degli offset, impostato a 1 per un ambiente di sviluppo single-node;
  - `KAFKA_AUTO_CREATE_TOPICS_ENABLE: "true"` - Abilita la creazione automatica dei topic$_G$ quando vengono referenziati.

- **Inizializzazione e Script**:
  - `command` - Script di inizializzazione eseguito all'avvio del container$_G$:
    - Rimuove eventuali nodi Kafka$_G$ esistenti in Zookeeper per evitare conflitti di ID;
    - Avvia il processo$_G$ Kafka$_G$ in background;
    - Crea esplicitamente due topic$_G$ cruciali per l'applicazione:
      - `MessageElaborated` - Topic$_G$ che conterrà i messaggi elaborati dal processore Flink$_G$;
      - `SimulatorPosition` - Topic$_G$ che conterrà i dati di posizione generati dal simulatore.
    - Configura entrambi i topic$_G$ con una partizione e un fattore di replica pari a 1 (adatto per ambienti di sviluppo).

- **Controlli e Disponibilità**:
  - `healthcheck` - Verifica automatica dello stato di salute del servizio:
    - `test` - Verifica che entrambi i topic$_G$ "MessageElaborated" e "SimulatorPosition" siano stati creati correttamente;
    - `interval: 10s` - Frequenza di esecuzione del controllo;
    - `timeout: 5s` - Tempo massimo di attesa per l'esecuzione del controllo;
    - `retries: 5` - Numero di tentativi prima di considerare il controllo fallito;
    - `start_period: 10s` - Periodo iniziale di grazia prima di iniziare i controlli.
  - `profiles: ["test","develop","prod"]` - Attivazione del servizio in tutti gli ambienti di esecuzione, essendo un componente fondamentale dell'architettura.

5.1.1.3 Kafdrop Service

- **Descrizione**: Implementa un'interfaccia web user-friendly per il monitoraggio e la gestione di Apache Kafka$_G$. Fornisce una visualizzazione grafica del cluster Kafka$_G$, dei topic$_G$, dei consumer group e dei messaggi, facilitando il debug e l'analisi del flusso di dati;

- **Configurazioni**:
  - `container_name: kafdrop` - Nome esplicito assegnato al container$_G$ per facilitarne l'identificazione;
  - `image: obsidiandynamics/kafdrop` - Immagine Docker$_G$ ufficiale di Kafdrop, un'interfaccia web open-source per Kafka$_G$;

- `restart: "no"` - Politica di riavvio configurata per non riavviare automaticamente il container$_G$ in caso di errori, adatta per un'interfaccia di amministrazione non critica;
- `ports: - "9080:9000"` - Mappatura delle porte che espone l'interfaccia web Kafdrop sulla porta 9080 dell'host, mentre internamente utilizza la porta 9000;
- `environment` - Variabili d'ambiente che configurano il comportamento di Kafdrop:
  - `KAFKA_BROKERCONNECT: "kafka:9092"` - Specifica l'indirizzo del broker$_G$ Kafka$_G$ a cui Kafdrop si connetterà, utilizzando il nome del servizio kafka$_G$ all'interno della rete Docker$_G$ e la porta 9092.
- `depends_on` - Dipendenze del servizio:
  - `kafka: condition: service_healthy` - Attende che il servizio Kafka$_G$ sia completamente funzionante (passando il healthcheck) prima di avviare Kafdrop.

- **Funzionalità principali**:
  - Visualizzazione della struttura del cluster Kafka$_G$, inclusi broker$_G$ e topic$_G$;
  - Monitoraggio dei dettagli dei topic$_G$, come partizioni, offset, e messaggi;
  - Esplorazione e visualizzazione dei messaggi all'interno dei topic$_G$;
  - Visualizzazione dei gruppi di consumatori e dei loro offset;
  - Interfaccia web intuitiva per amministratori e sviluppatori.
- **Disponibilità**:
  - `profiles: ["develop","prod"]` - Il servizio è attivato solo negli ambienti di sviluppo e produzione, ma non in quello di test$_G$, indicando che è destinato principalmente come strumento di supporto per sviluppatori e operatori, piuttosto che come un componente fondamentale per i test$_G$ automatizzati.

5.1.1.4 Grafana Service

- **Descrizione**: Implementa una piattaforma di visualizzazione e analisi dati open-source che permette di creare dashboard$_G$ interattive per monitorare l'andamento dell'applicazione. Fornisce una rappresentazione grafica dei dati archiviati in Clickhouse$_G$, facilitando l'interpretazione e l'analisi dei flussi di messaggi e delle posizioni degli utenti.
- **Configurazioni**:
  - `container_name: grafana` - Nome esplicito assegnato al container$_G$ per facilitarne l'identificazione;
  - `image: grafana/grafana:latest` - Immagine Docker$_G$ ufficiale di Grafana$_G$, utilizzando l'ultima versione disponibile;
  - `ports: - "3000:3000"` - Mappatura delle porte che espone l'interfaccia web Grafana$_G$ sulla porta 3000 dell'host;
  - `environment` - Variabili d'ambiente che configurano il comportamento di Grafana$_G$:
    - `GF_SECURITY_ADMIN_PASSWORD: admin` - Imposta la password dell'utente amministratore al valore "admin";
    - `GF_INSTALL_PLUGINS: "grafana-clickhouse-datasource"` - Installa automaticamente il plugin per la connessione a Clickhouse$_G$ come fonte dati.
  - `volumes` - Configurazione dei volumi che collegano directory locali a percorsi all'interno del container$_G$:
    - `./Grafana/DashboardProv:/etc/grafana/provisioning/dashboards` - Monta i file di configurazione per il provisioning automatico delle dashboard$_G$;
    - `./Grafana/Dashboards:/var/lib/grafana/dashboards` - Monta le definizioni JSON$_G$ delle dashboard$_G$ personalizzate;
    - `./Grafana/DatasourceProv:/etc/grafana/provisioning/datasources` - Monta la configurazione delle fonti dati, in particolare la connessione a Clickhouse$_G$.
- **Funzionalità principali**:

- Visualizzazione in tempo reale dei dati provenienti da Clickhouse$_G$;
- Dashboard$_G$ personalizzate per monitorare metriche chiave dell'applicazione;
- Pannelli di controllo per monitorare lo stato del sistema$_G$;
- Integrazione completa con Clickhouse$_G$ grazie al plugin dedicato;
- Provisioning automatico di dashboard$_G$ e fonti dati attraverso file di configurazione.

- **Disponibilità**:
  - `profiles: ["develop","prod"]` - Il servizio è attivato solo negli ambienti di sviluppo e produzione, ma non in quello di test$_G$, essendo considerato uno strumento di supporto per monitoraggio e analisi piuttosto che un componente essenziale per i test$_G$ automatizzati.

5.1.1.5 ClickHouse Service

- **Descrizione**: Implementa un database$_G$ colonnare ad alte prestazioni, ottimizzato per carichi di lavoro analitici e query$_G$ su grandi volumi di dati. Costituisce il layer di persistenza dell'applicazione, memorizzando dati di utenti, sensori, attività commerciali e messaggi pubblicitari generati;
- **Configurazioni**:
  - `image: clickhouse/clickhouse-server:24.10` - Immagine Docker$_G$ ufficiale di Clickhouse$_G$, versione 24.10, che garantisce stabilità e prestazioni;
  - `hostname: clickhouse` - Nome host assegnato al container$_G$ all'interno della rete Docker$_G$;
  - `container_name: clickhouse` - Nome esplicito assegnato al container$_G$ per facilitarne l'identificazione;
  - `ports` - Mappatura delle porte tra host e container$_G$:
    - `"8123:8123"` - Espone la porta HTTP di Clickhouse$_G$ per query$_G$ REST e interfaccia web;
    - `"9000:9000"` - Espone la porta nativa di Clickhouse$_G$ per connessioni client dirette.
  - `environment` - Variabili d'ambiente che configurano il comportamento di Clickhouse$_G$:
    - `CLICKHOUSE_DB: nearyou` - Crea automaticamente un database$_G$ chiamato "nearyou" all'avvio;
    - `CLICKHOUSE_USER: default` - Configura l'utente predefinito del sistema$_G$;
    - `CLICKHOUSE_PASSWORD: pass` - Imposta la password per l'utente predefinito;
    - `CLICKHOUSE_DEFAULT_ACCESS_MANAGEMENT: 0` - Disabilita la gestione degli accessi predefinita, utilizzando configurazioni personalizzate.
  - `volumes` - Configurazione dei volumi:
    - `./StorageData:/docker-entrypoint-initdb.d` - Monta la directory locale ./StorageData nella directory di inizializzazione di Clickhouse$_G$, dove gli script SQL$_G$ vengono eseguiti automaticamente al primo avvio per creare tabelle, indici e popolare dati iniziali.
- **Funzionalità principali**:
  - Archiviazione efficiente di grandi volumi di dati in formato colonnare;
  - Esecuzione di query$_G$ analitiche ad alte prestazioni;
  - Supporto per funzioni geospaziali utilizzate per calcolare distanze tra utenti e attività;
  - Integrazione con Grafana$_G$ per la visualizzazione dei dati;
  - Inizializzazione automatica dello schema e dei dati all'avvio tramite script SQL$_G$;
  - Supporto per strutture dati complesse come array (utilizzati per gli interessi degli utenti).
- **Disponibilità**:
  - `profiles: ["test","develop","prod"]` - Il servizio è attivato in tutti gli ambienti (test, sviluppo e produzione), evidenziando il suo ruolo critico come componente infrastrutturale dell'applicazione.

5.1.1.6 Position Simulator Service

- **Descrizione**: Implementa un servizio di simulazione che genera realistici dati di posizione degli utenti e li pubblica sul broker$_G$ Kafka$_G$. Utilizza strategie di simulazione basate su grafi stradali per creare percorsi plausibili, simulando il movimento di utenti nel contesto urbano;
- **Configurazioni**:
  - `container_name: positions` - Nome esplicito assegnato al container$_G$ per facilitarne l'identificazione;
  - `build: ./SimulationModule` - Specificazione della directory che contiene il Dockerfile e il codice sorgente necessario per costruire l'immagine personalizzata;
  - `depends_on` - Dipendenze del servizio:
    - `kafka: condition: service_healthy` - Attende che il servizio Kafka$_G$ sia completamente funzionante (passando il healthcheck) prima di avviare il simulatore.
  - `mem_limit: 4G` - Comentato, ma pronto per limitare la memoria utilizzata dal container$_G$ a 4GB, se necessario.
- **Dockerfile specifico**:
  - `FROM python:3.8` - Utilizza l'immagine base Python$_G$ 3.8 ufficiale;
  - `Prerequisiti` - Installa pacchetti di sistema$_G$ essenziali:
    - `gdal-bin` e `libgdal-dev` - Librerie per la manipolazione di dati geospaziali;
    - `build-essential` - Strumenti di compilazione per dipendenze native;
    - `librdkafka-dev` - Libreria client Kafka$_G$ nativa richiesta da confluent-kafka.
  - `Variabili d'ambiente` - Configura GDAL per la corretta compilazione delle estensioni Python$_G$;
  - `Dipendenze Python`$_G$ - Installa i pacchetti Python$_G$ necessari da requirements.txt:
    - `geopy` - Geocodifica e calcoli di distanza;
    - `osmnx` - Accesso ai dati OpenStreetMap e manipolazione di grafi stradali;
    - `gpxpy` - Parsing e generazione di file GPX;
    - `confluent_kafka` - Client Kafka$_G$ per la pubblicazione dei messaggi;
    - `scikit-learn` - Algoritmi di machine learning per simulazione avanzata;
    - `clickhouse-connect` - Client per la connessione al database$_G$ Clickhouse$_G$.
- **Disponibilità**:
  - `profiles: ["test","develop","prod"]` - Il servizio è attivato in tutti gli ambienti (test, sviluppo e produzione), evidenziando il suo ruolo essenziale nell'architettura dell'applicazione.

5.1.1.7 Flink Service

- **Descrizione**: Implementa un motore di elaborazione di stream in tempo reale basato su Apache Flink$_G$ che processa i dati di posizione provenienti da Kafka$_G$, li arricchisce con informazioni contestuali e genera messaggi pubblicitari personalizzati. Rappresenta il cuore computazionale dell'applicazione, realizzando la logica di business principale;
- **Configurazioni**:
  - `container_name: flink` - Nome esplicito assegnato al container$_G$ per facilitarne l'identificazione;
  - `build: ./FlinkProcessor` - Specificazione della directory che contiene il Dockerfile e il codice sorgente dell'applicazione Flink$_G$;
  - `volumes` - Configurazione dei volumi:
    - `.env:/app/.env` - Monta il file di variabili d'ambiente nella directory dell'applicazione, consentendo l'accesso a chiavi API$_G$ e altre configurazioni sensibili.
  - `restart: on-failure:5` - Politica di riavvio che tenta di riavviare il container$_G$ fino a 5 volte in caso di errore;

- `deploy` - Configurazioni di deployment:
  - `resources.limits.cpus: '4.00'` - Limita l'utilizzo della CPU a 4 core;
  - `resources.limits.memory: 4G` - Limita l'utilizzo della memoria a 4 GB.
- `depends_on` - Dipendenze del servizio:
  - `kafka: condition: service_healthy` - Attende che il servizio Kafka$_G$ sia completamente funzionante prima di avviare l'elaborazione.

- **Dockerfile specifico**:
  - `FROM apache/flink:1.18.1-scala_2.12-java11` - Utilizza l'immagine base di Apache Flink$_G$ ufficiale con Java 11 e Scala 2.12;
  - `Ambiente Python`$_G$ - Installa Python$_G$ 3 e pip per supportare PyFlink:
    - `apt-get install python3 python3-pip python3-dev` - Installa l'ambiente Python$_G$;
    - `ln -s /usr/bin/python3 /usr/bin/python` - Crea un symlink per rendere Python$_G$ 3 il default.
  - `Dipendenze Python`$_G$ - Installa le dipendenze Python$_G$ dal file requirements.txt:
    - `PyFlink` - API$_G$ Python$_G$ per Apache Flink$_G$;
    - `clickhouse-connect` - Client per la connessione al database$_G$ Clickhouse$_G$;
    - `langchain` e `langchain-groq` - Framework$_G$ per l'integrazione di modelli linguistici;
    - `python-dotenv` - Gestione delle variabili d'ambiente;
    - `pydantic` - Validazione e serializzazione dei dati.
  - `Connettori Flink`$_G$ - Scarica e installa i connettori Java necessari:
    - `flink-sql-connector-kafka` - Connettore per integrare Kafka$_G$ con Flink$_G$;
    - `clickhouse-jdbc` - Driver JDBC per Clickhouse$_G$;
    - `flink-connector-jdbc` - Connettore JDBC generico per Flink$_G$.
  - `WORKDIR /app` - Imposta la directory di lavoro all'interno del container$_G$;
  - `CMD [ "python", "main.py" ]` - Comando di avvio che esegue lo script principale dell'applicazione.

- **Disponibilità**:
  - `profiles: ["test","develop","prod"]` - Il servizio è attivato in tutti gli ambienti (test, sviluppo e produzione), evidenziando il suo ruolo fondamentale nell'architettura dell'applicazione.

5.1.1.8 Test Service

- **Descrizione**: Implementa un servizio dedicato all'esecuzione automatizzata dei test$_G$ dell'applicazione, includendo test$_G$ unitari, di integrazione e di sistema$_G$. Fornisce un ambiente isolato ma completo per verificare il corretto funzionamento di tutti i componenti e la loro interazione, generando report dettagliati sulla copertura e qualità del codice;

- **Configurazioni in docker-compose**:
  - `container_name: test` - Nome esplicito assegnato al container$_G$ per facilitarne l'identificazione;
  - `build` - Configurazione della build$_G$ dell'immagine Docker$_G$:
    - `context: ./` - Utilizza la directory root del progetto$_G$ come contesto per la build$_G$;
    - `dockerfile: Tests/Dockerfile` - Specifica il Dockerfile da utilizzare, situato nella directory Tests.
  - `volumes` - Configurazione dei volumi (commentata, ma disponibile per l'attivazione):
    - `./.github/reports:/app/reports:rw` - Montaggio della directory dei report per l'integrazione con CI/CD.
  - `depends_on` - Dipendenze del servizio:
    - `kafka: condition: service_healthy` - Attende che il servizio Kafka$_G$ sia completamente funzionante prima di eseguire i test$_G$.

- **Dockerfile specifico**:
  - `FROM apache/flink:1.18.1-scala_2.12-java11` - Utilizza l'immagine base di Apache Flink$_G$ per garantire compatibilità con l'ambiente di produzione;
  - `WORKDIR /app` - Imposta la directory di lavoro all'interno del container$_G$;
  - `Configurazione delle dipendenze Flink`$_G$:
    - Scarica e installa i connettori Java necessari: Kafka$_G$, Clickhouse$_G$ JDBC, e JDBC generico.
  - `Struttura delle directory di test`$_G$:
    - Crea struttura di directory per i moduli SimulationModule, FlinkProcessor, IntegrationTests e SystemTests;
    - Copia i file di configurazione dei test$_G$: .coveragerc, pylintrc, pytest.ini.
  - `Ambiente Python`$_G$:
    - Installa Python$_G$ 3, pip e le dipendenze necessarie da requirements.txt;
    - Installa strumenti specifici per i test$_G$: pytest, pytest-cov, coveralls, pylint.
  - `Preparazione e configurazione dei report`:
    - Crea directory per i report (/app/reports) con permessi appropriati.
- **Funzionalità di test$_G$**:
  - `Analisi statica` - Esecuzione di pylint per verificare la qualità del codice:
    - Utilizza configurazione personalizzata dal file pylintrc;
    - Genera report in formato parseable per l'integrazione con CI/CD.
  - `Test unitari e di integrazione` - Esecuzione di pytest con varie opzioni:
    - Test$_G$ per SimulationModule, FlinkProcessor e IntegrationTests;
    - Misurazione della copertura del codice con analisi dei branch$_G$;
    - Generazione di report XML per la copertura.
  - `Generazione di report` - Esecuzione dello script getReports.py che probabilmente formatta o aggrega i risultati dei test$_G$ per facilitarne la lettura.
- **Disponibilità**:
  - `profiles: ["test"]` - Il servizio è attivato solo nell'ambiente di test$_G$, evidenziando la sua funzione specifica per la verifica della qualità del codice piuttosto che per l'operatività dell'applicazione.

5.1.2 Dipendenze tra componenti

Le interazioni tra i vari componenti avvengono attraverso Kafka$_G$, che garantisce l'invio e la ricezione di messaggi in modo affidabile e resiliente.

- **Generazione di dati:**
  - **Container sensor-simulator:**
    . Esegue il simulatore dei sensori di posizione degli utenti;
    . Implementa diverse strategie di movimento per generare dati realistici;
    . Produce dati nel formato JSON$_G$ definito e li invia al broker$_G$ Kafka$_G$.
- **Gestione messaggi:**
  - **Container kafka$_G$:**
    . Esegue Apache Kafka$_G$ per la gestione del flusso di dati in tempo reale;
    . Gestisce i topic$_G$ dedicati per i diversi tipi di messaggi (posizioni, PoI$_G$, messaggi pubblicitari);
    . Accessibile agli altri container$_G$ tramite l'indirizzo kafka$_G$:9092.

  - **Componenti di supporto:**
    . **Container zookeeper:**
      - Esegue il servizio di coordinamento per Kafka$_G$;
      - Gestisce lo stato distribuito del sistema$_G$;
      - Accessibile dagli altri container$_G$ attraverso l'indirizzo zookeeper:2181.
    . **Container kafka-ui:**
      - Fornisce un'interfaccia web per il monitoraggio e la gestione di Kafka$_G$;
      - Espone la porta 8080 per accedere alla dashboard$_G$ di amministrazione.

- **Elaborazione dei dati:**
  - **Container flink-jobmanager:**
    . Coordina l'esecuzione dei job di elaborazione dati in tempo reale;
    . Gestisce l'allocazione delle risorse e la pianificazione dei task;
    . Espone la porta 8081 per l'interfaccia di amministrazione.
  - **Container flink-taskmanager:**
    . Esegue i task di elaborazione dati assegnati dal jobmanager;
    . Implementa gli algoritmi di proximity detection per identificare punti di interesse rilevanti;
    . Integra il servizio LLM$_G$ per la generazione di messaggi pubblicitari personalizzati.

- **Storage:**
  - **Container clickhouse$_G$:**
    . Esegue Clickhouse$_G$ come database$_G$ column-oriented ad alte prestazioni;
    . Memorizza i dati degli utenti, posizioni, punti di interesse e messaggi pubblicitari;
    . La banca dati è accessibile agli altri container$_G$ tramite l'indirizzo clickhouse$_G$:8123 e 9000.

- **Visualizzazione:**
  - **Container grafana$_G$:**
    . Esegue Grafana$_G$ come piattaforma di visualizzazione e monitoraggio;
    . Offre dashboard$_G$ interattive per l'analisi dei dati di posizione e messaggi pubblicitari;
    . Espone la porta 3000 all'esterno per permettere l'accesso alle dashboard$_G$;
    . Consente l'integrazione con vari datasource per la visualizzazione dei dati.

5.2 Continuous Integration

- **Descrizione e approccio**: Il progetto$_G$ implementa una robusta pipeline di Continuous-integration$_G$ (CI) basata su Github-actions$_G$, progettata per automatizzare il testing, la valutazione della qualità del codice e la generazione di report dettagliati ad ogni push sul branch$_G$ principale o apertura di pull-request$_G$. Questo approccio integrato garantisce che ogni modifica al codebase venga rigorosamente verificata prima dell'integrazione, mantenendo elevati standard qualitativi durante tutto il ciclo di sviluppo.

- **Workflow e automatizzazione**:
  - La pipeline CI viene attivata automaticamente in risposta a eventi Git$_G$ specifici (come push su main e pull-request$_G$), creando un ciclo di feedback$_G$ immediato per gli sviluppatori;
  - L'intero stack applicativo viene costruito in un ambiente isolato utilizzando Docker-compose$_G$ con il profilo "test", garantendo che i test$_G$ vengano eseguiti in un ambiente identico a quello di produzione;
  - I container$_G$ vengono orchestrati per eseguire in sequenza, con controlli di dipendenza che assicurano che componenti come Kafka$_G$ e Clickhouse$_G$ siano completamente inizializzati prima dell'esecuzione dei test$_G$;
  - Al completamento dei test$_G$, i report vengono estratti dal container$_G$ e archiviati come artefatti permanenti del repository$_G$, creando una cronologia consultabile dell'evoluzione qualitativa del progetto$_G$.

- **Misure di qualità e reporting**:
  - Il sistema$_G$ genera e traccia metriche complete sulla qualità del codice, tra cui:
    - Copertura dei test$_G$ (sia a livello di linee che di branch$_G$) con report in formato XML compatibile con servizi esterni come Coveralls;
    - Analisi statica tramite pylint con regole personalizzate definite in un file di configurazione dedicato;
    - Metriche di complessità del codice come fan-in, fan-out, numero di attributi, parametri e lunghezza delle funzioni, visualizzate attraverso grafici generati automaticamente.
  - I risultati vengono visualizzati dinamicamente nel README del progetto$_G$ attraverso badge aggiornati ad ogni esecuzione, fornendo un'istantanea immediata dello stato del progetto$_G$;
  - L'integrazione con servizi esterni come Coveralls consente il monitoraggio delle tendenze nel tempo e il confronto con benchmark di settore.
- **Trasparenza e comunicazione**:
  - I report generati non sono solo archiviati ma anche visualizzati attraverso grafici che evidenziano l'evoluzione delle metriche nel tempo, facilitando l'identificazione di trend e potenziali problemi;
  - Questi grafici vengono automaticamente aggiornati e inseriti nel repository$_G$, creando una documentazione$_G$ visiva accessibile a tutti i membri del team;
  - I badge nel README forniscono un'indicazione immediata della salute del progetto$_G$, incentivando il mantenimento di standard elevati e facilitando la comunicazione dello stato del progetto$_G$ a stakeholder$_G$ tecnici e non.
- **Integrazione con il processo$_G$ di sviluppo**:
  - La CI è progettata per integrarsi perfettamente con il flusso di lavoro Git-flow adottato dal team, fornendo feedback$_G$ immediato sulle pull-request$_G$;
  - I risultati dei test$_G$ diventano parte della documentazione$_G$ delle pull-request$_G$, facilitando il processo$_G$ di code review e la decisione sull'approvazione delle modifiche;
  - L'automazione della generazione e del commit dei report riduce il carico manuale sul team, consentendo agli sviluppatori di concentrarsi sulla risoluzione dei problemi piuttosto che sulla loro documentazione$_G$.
- **Sicurezza e gestione delle credenziali**:
  - La pipeline utilizza i segreti di Github$_G$ per gestire in modo sicuro token sensibili come quello di Coveralls, garantendo che le credenziali non vengano esposte;
  - L'utilizzo di un'identità bot per i commit generati automaticamente (`github-actions[bot]`) permette di distinguere chiaramente tra modifiche manuali e automatiche nella cronologia del repository$_G$.
- **Estensibilità e manutenzione**:
  - La struttura modulare del workflow facilita l'aggiunta di nuovi strumenti di analisi o la modifica di quelli esistenti senza richiedere una riprogettazione completa;
  - Lo script `createCharts.py` è progettato per elaborare i report generati e produrre visualizzazioni significative, con la possibilità di estendere facilmente l'analisi a nuove metriche;
  - La generazione automatica di badge tramite `readmeBadges.py` può essere facilmente adattata per includere nuovi indicatori di qualità man mano che evolvono le esigenze del progetto$_G$.

5.3 Vantaggi dell'architettura containerizzata

Questa struttura containerizzata 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:

- **Isolamento**: Ogni componente opera nel proprio ambiente isolato, riducendo le interferenze tra servizi;
- **Portabilità**: L'applicazione può essere eseguita su qualsiasi piattaforma che supporti Docker$_G$;
- **Portabilità della configurazione**: Grazie al compose di Docker$_G$ si possono definire delle cartelle o dei file di configurazione per ogni container$_G$, rendendo il sistema$_G$ facilmente replicabile;
- **Scalabilità orizzontale**: I container$_G$ possono essere facilmente replicati per gestire carichi maggiori;
- **Gestione dichiarativa**: La configurazione dell'intero ambiente è definita nel file docker-compose.yml;
- **Efficienza delle risorse**: Ogni container$_G$ riceve solo le risorse necessarie per il suo funzionamento.

5.4 Comunicazione tra container

La comunicazione tra i vari container$_G$ avviene principalmente attraverso il networking interno di Docker$_G$, con Kafka$_G$ che agisce come backbone di messaggistica centrale del sistema$_G$. Questa architettura event-driven garantisce:

- **Disaccoppiamento**: I componenti possono evolvere indipendentemente, purché mantengano l'interfaccia di comunicazione;
- **Persistenza dei messaggi**: I dati vengono memorizzati in Kafka$_G$ e copiati subito dopo su una tabella Clickhouse$_G$, garantendo la storicizzazione del dato.

5.5 Orchestrazione e gestione

Per la gestione dei container$_G$ in ambiente di produzione, sono state implementate le seguenti strategie:

- **Health checks**: Ogni container$_G$ è configurato con controlli di integrità che verificano periodicamente il corretto funzionamento del servizio;
- **Gerarchia di avvio dei container$_G$**: I container$_G$ vengono avviati in un ordine specifico per garantire che le dipendenze siano soddisfatte prima di avviare i servizi che ne fanno uso.

5.6 Evoluzione futura

L'architettura di deployment containerizzato con la sua separazione degli ambienti offre facile implementazione di future evoluzioni del sistema$_G$, tra cui:

- **Migrazione verso Kubernetes**: L'attuale configurazione Docker$_G$ è pronta per essere eventualmente trasferita su un orchestratore come Kubernetes per una gestione più avanzata dei container$_G$;
- **Implementazione di auto-scaling**: Aggiunta di meccanismi per scalare automaticamente i servizi in base al carico;
- **Monitoraggio**: C'è la possibilità di integrare strumenti di monitoraggio avanzati per analizzare le performance e il comportamento del sistema$_G$ in tempo reale.

6 Stato dei requisiti funzionali

La presente sezione fornisce una visione d'insieme dello stato di avanzamento dei requisiti funzionali identificati durante la fase di analisi. I requisiti funzionali sono stati classificati in base alla loro importanza (obbligatori, desiderabili e opzionali) come definito nel documento *Analisi_dei_Requisiti_v2.0.0*.

6.1 Riepilogo dei requisiti

Durante la fase di analisi sono stati individuati 29 requisiti funzionali (RF01-RF29), di cui:

- 27 requisiti obbligatori;
- 0 requisiti desiderabili;
- 2 requisiti opzionali.

I requisiti funzionali riguardano principalmente:

- La visualizzazione della Dashboard$_G$ e dei marker$_G$ sulla mappa;
- La gestione e visualizzazione dei punti di interesse;
- La gestione e visualizzazione degli utenti;
- La visualizzazione degli annunci pubblicitari generati;
- La trasmissione e gestione dei dati geoposizionali.

6.2 Tabella dei requisiti funzionali

Id. Requisito$_G$	Importanza	Descrizione	Stato
RF01	Obbligatorio	L'utente privilegiato deve poter visualizzare la Dashboard$_G$ composta da una mappa interattiva con i vari Marker$_G$ su di essa.	Implementato
RF02	Obbligatorio	L'utente privilegiato deve poter visualizzare dei Marker$_G$ che rappresentano i vari Percorsi effettuati in tempo reale dagli utenti presenti nel Sistema$_G$	Implementato
RF03	Obbligatorio	L'utente privilegiato deve poter visualizzare un Marker$_G$ che rappresenta un Percorso$_G$ effettuato in tempo reale da un utente presente nel Sistema$_G$	Implementato
RF04	Obbligatorio	L'utente privilegiato deve poter visualizzare tutti i punti di interesse riconosciuti dal Sistema$_G$.	Implementato
RF05	Obbligatorio	L'utente privilegiato deve poter visualizzare un Marker$_G$ che rappresenta un punto di interesse riconosciuto dal Sistema$_G$.	Implementato
RF06	Obbligatorio	L'utente privilegiato deve poter visualizzare gli annunci pubblicitari provenienti da un determinato punto di interesse.	Implementato

RF07	Obbligatorio	L'utente privilegiato deve poter visualizzare un singolo annuncio pubblicitario tramite un Marker$_G$.	Implementato
RF08	Obbligatorio	L'utente privilegiato deve poter visualizzare una Dashboard$_G$ relativa ad un singolo utente quando seleziona un Marker$_G$ utente nella Dashboard$_G$ principale.	Implementato
RF09	Obbligatorio	L'utente privilegiato deve poter visualizzare dei Marker$_G$ che rappresentano lo storico delle posizioni dell'utente a cui è riferita la Dashboard$_G$ di singolo utente.	Implementato
RF10	Obbligatorio	L'utente privilegiato deve poter visualizzare un Marker$_G$ che rappresenta la posizione dell'utente in un determinato istante nella Dashboard$_G$ di singolo utente.	Implementato
RF11	Obbligatorio	L'utente privilegiato deve poter visualizzare, nella Dashboard$_G$ di singolo utente, tutti i punti di interesse riconosciuti dal Sistema$_G$.	Implementato
RF12	Obbligatorio	L'utente privilegiato deve poter visualizzare, nella Dashboard$_G$ di singolo utente, un Marker$_G$ che rappresenta un punto di interesse riconosciuto dal Sistema$_G$.	Implementato
RF13	Obbligatorio	L'utente privilegiato deve poter visualizzare lo storico degli annunci pubblicitari generati per l'utente a cui è riferita la Dashboard$_G$ singolo utente.	Implementato
RF14	Obbligatorio	L'utente privilegiato deve poter visualizzare un singolo annuncio pubblicitario tramite un Marker$_G$ nella Dashboard$_G$ di singolo utente.	Implementato
RF15	Obbligatorio	L'utente privilegiato deve poter visualizzare un pannello$_G$ apposito contenente le informazioni dell'utente, a cui è riferita la Dashboard$_G$ di singolo utente, in forma tabellare.	Implementato
RF16	Obbligatorio	L'utente privilegiato deve poter visualizzare nel pannello$_G$ apposito di visualizzazione informazioni dell'utente: il nome, il cognome, l'email, il genere, la data di nascita e lo stato civile.	Implementato
RF17	Obbligatorio	L'utente privilegiato deve poter visualizzare i dettagli del Marker$_G$ riguardante una singola posizione di un utente nella rispettiva Dashboard$_G$	Implementato

RF18	Obbligatorio	L'utente privilegiato quando visualizza i dettagli del Marker$_G$, riguardante una singola posizione di un utente nella rispettiva Dashboard$_G$, deve poter vedere la latitudine, la longitudine e l'istante di rilevamento del Marker$_G$	Implementato
RF19	Opzionale	L'utente privilegiato deve poter visualizzare l'area di influenza di un punto di interesse selezionato.	Da implementare
RF20	Obbligatorio	L'utente privilegiato deve poter visualizzare le informazioni dettagliate di un punto di interesse quando selezionato.	Implementato
RF21	Obbligatorio	L'utente privilegiato quando visualizza le informazioni dettagliate di un punto di interesse deve poter visualizzare la latitudine, la longitudine, il nome, la tipologia e la descrizione del punto di interesse.	Implementato
RF22	Opzionale	L'utente deve poter visualizzare l'annuncio pubblicitario proveniente dal punto di interesse situato nell'area che sta attraversando.	Da implementare
RF23	Obbligatorio	L'utente privilegiato deve poter visualizzare una tabella contenente le informazioni dei singoli PoI$_G$ ordinati per la quantità di messaggi inviati nel mese.	Implementato
RF24	Obbligatorio	L'utente privilegiato deve poter visualizzare nella tabella dei PoI$_G$ un singolo PoI$_G$, rappresentato da una riga della tabella.	Implementato
RF25	Obbligatorio	L'utente privilegiato deve poter visualizzare in ogni riga della tabella dei PoI$_G$ il nome, l'indirizzo, la tipologia (di che ambito si occupa), la descrizione e il numero di messaggi inviati durante il mese di un singolo PoI$_G$.	Implementato
RF26	Obbligatorio	L'utente privilegiato deve poter visualizzare i dettagli di un annuncio generato.	Implementato
RF27	Obbligatorio	L'utente privilegiato quando visualizza i dettagli di un annuncio deve poter visualizzare la latitudine, la longitudine, l'istante di creazione, il nome dell'utente coinvolto, il nome del punto di interesse coinvolto e il contenuto dell'annuncio.	Implementato

RF28	Obbligatorio	Il sensore deve essere in grado di trasmettere i dati rilevati in tempo reale al Sistema$_G$.	Implementato
RF29	Obbligatorio	Il sensore deve essere in grado di trasmettere il proprio id, la sua latitudine e longitudine al Sistema$_G$.	Implementato

6.3 Stato di implementazione

Lo stato di implementazione dei requisiti funzionali è rappresentato nella seguente tabella:

Tipo di requisito$_G$	**Totale**	**Implementati**	**In implementazione**	**Da implementare**
Obbligatori	27	27	0	0
Desiderabili	0	0	0	0
Opzionali	2	0	0	2
Totale	29	27	0	2

Table 4: Stato di implementazione dei requisiti funzionali

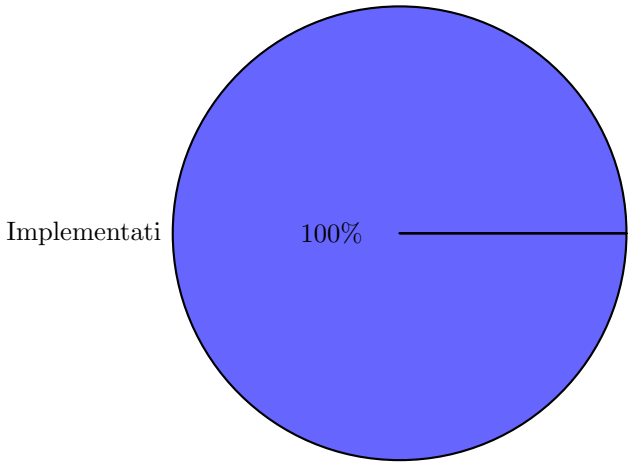

Figure 10: Stato dei requisiti funzionali obbligatori

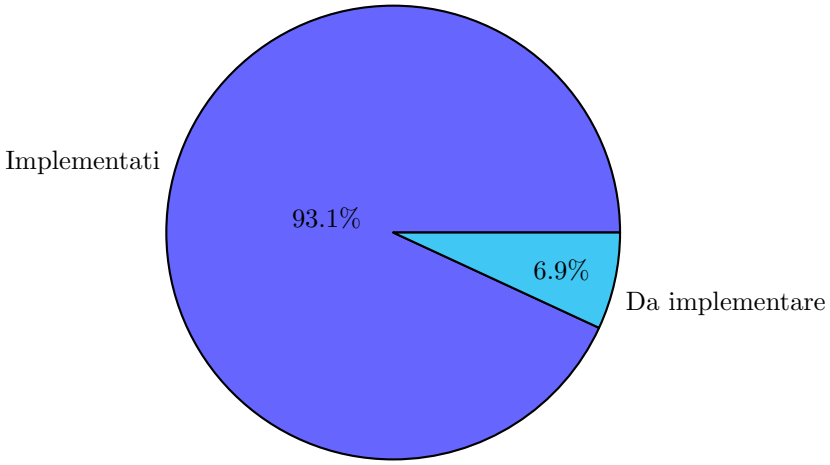

Figure 11: Stato dei requisiti funzionali totali

6.4 Riepilogo e Conclusioni

L'analisi aggiornata dello stato dei requisiti funzionali indica un completamento quasi totale delle funzionalità richieste. Con il 93.1% dei requisiti funzionali totali implementati, il progetto$_G$ ha raggiunto un traguardo significativo. Per quanto riguarda i requisiti obbligatori, l'implementazione è completa al 100%.
Attualmente, nessun requisito$_G$ è in fase di implementazione. I soli requisiti ancora da implementare sono i 2 requisiti opzionali. Il completamento di tutti i requisiti obbligatori fornisce una base estremamente solida per la consegna finale, mentre il sistema$_G$ di test$_G$ definito continuerà a garantire la qualità del prodotto finale in vista dell'implementazione dei requisiti opzionali rimanenti.