



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	BycycleSimulationStrategy	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

- **Dialecto:** 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;
- Storizzazione 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:**

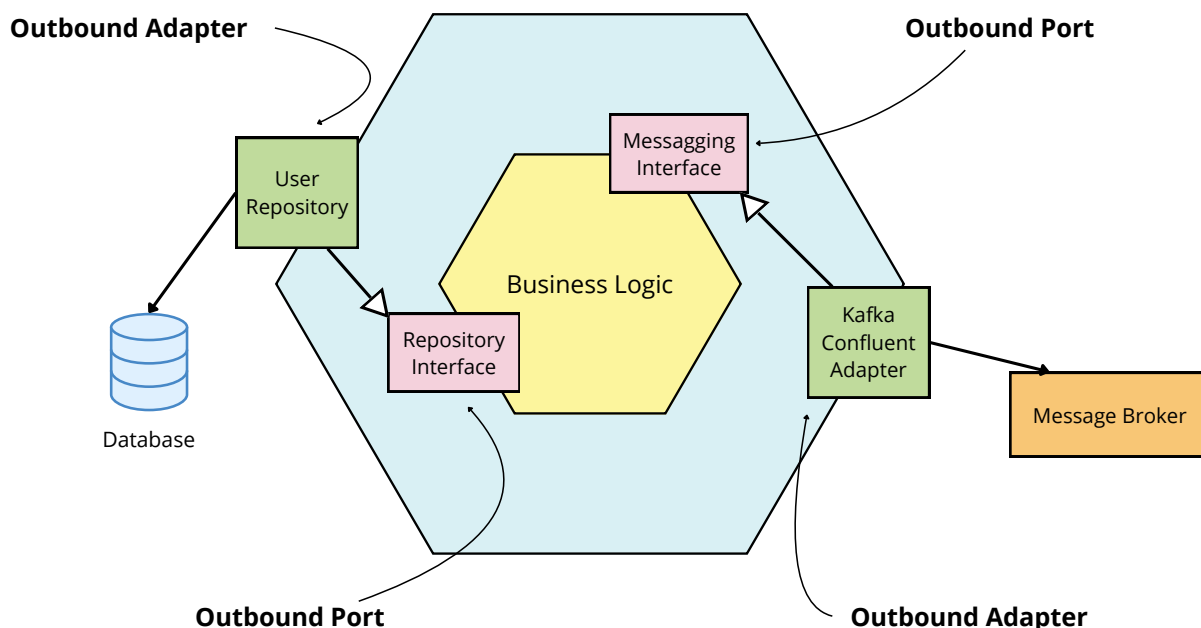


Figure 1: Architettura esagonale del SimulationService

- . **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:

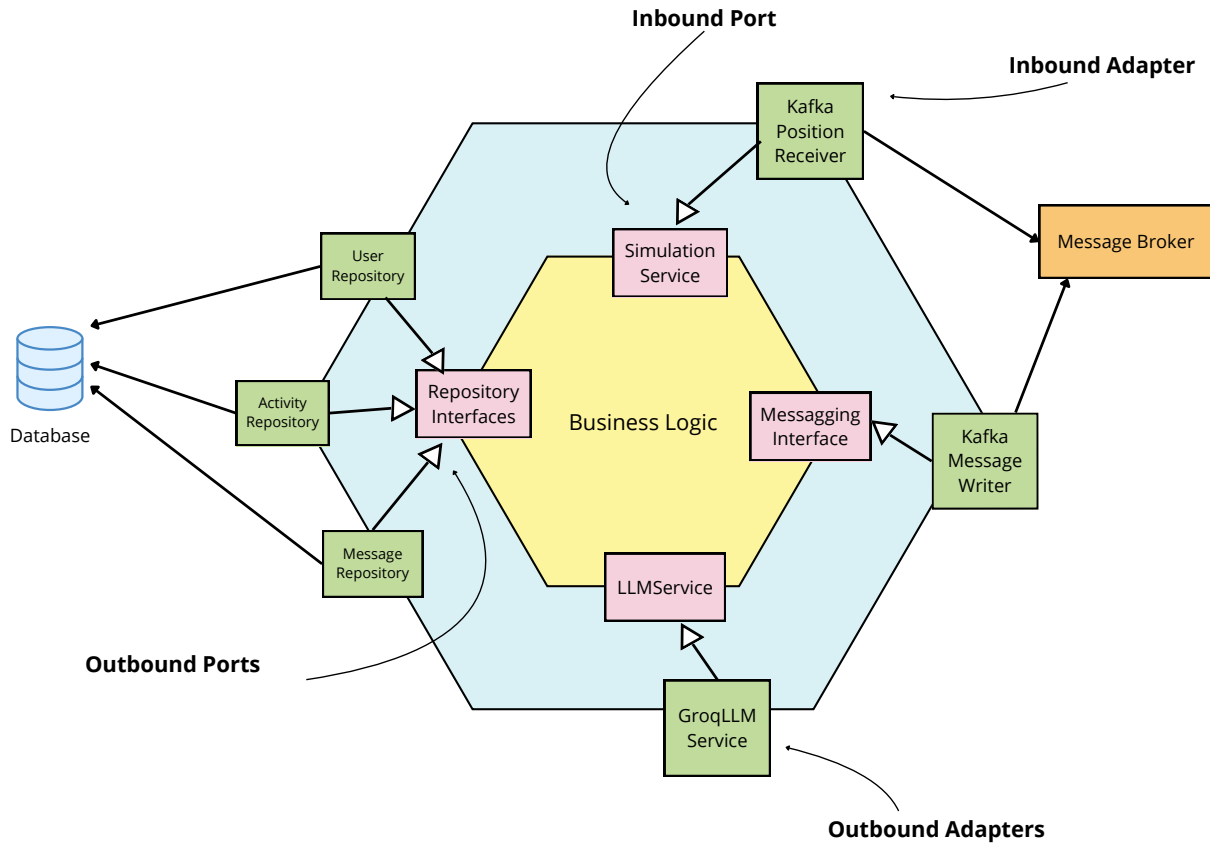


Figure 2: Architettura esagonale del PositionToMessageService

- . **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.

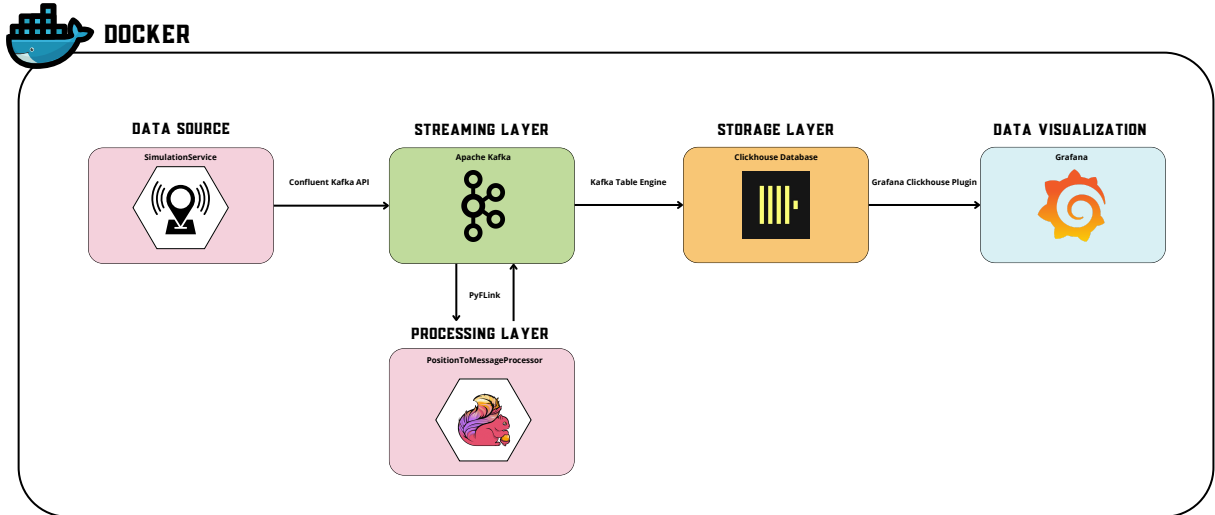


Figure 3: Diagramma dell'architettura di Sistema_G

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 datasources 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	<i>SimulationService</i> 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	<i>PositionToMessageService</i> 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	<i>PositionToMessageService</i> 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

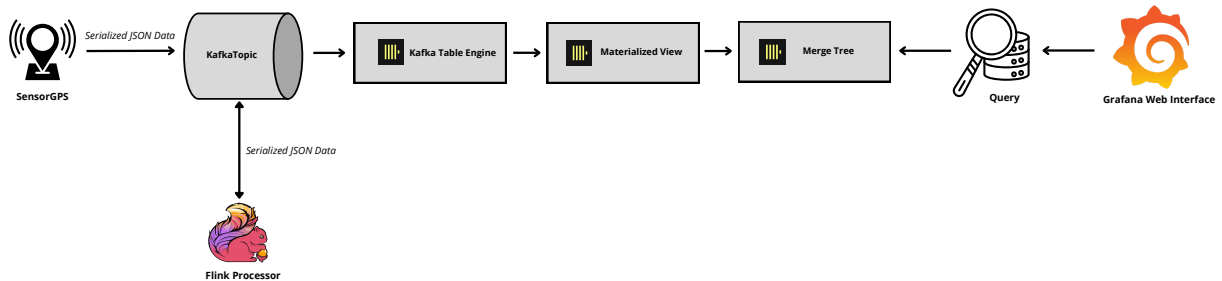


Figure 4: Flusso dei dati nell'architettura

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 esplicita 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 `KafkaG MessageElaborated`.

5. Persistenza e visualizzazione:

- . Clickhouse_G, attraverso il connettore Kafka_G nativo, in particolare sfruttando le tecnologie *Kafka Table Engine* \Rightarrow *Materialized View* \Rightarrow *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 già 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

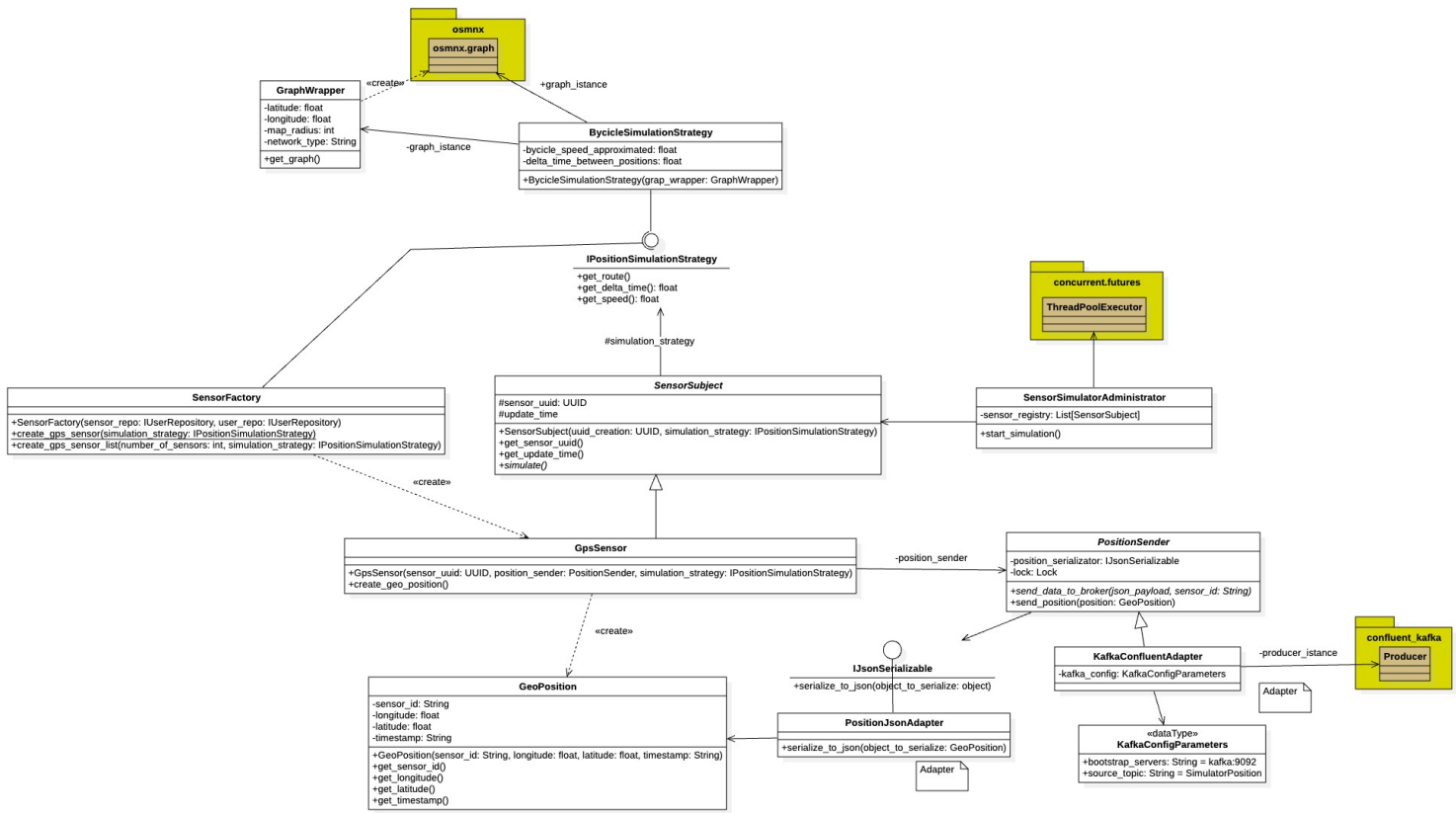


Figure 5: SimulationService Core

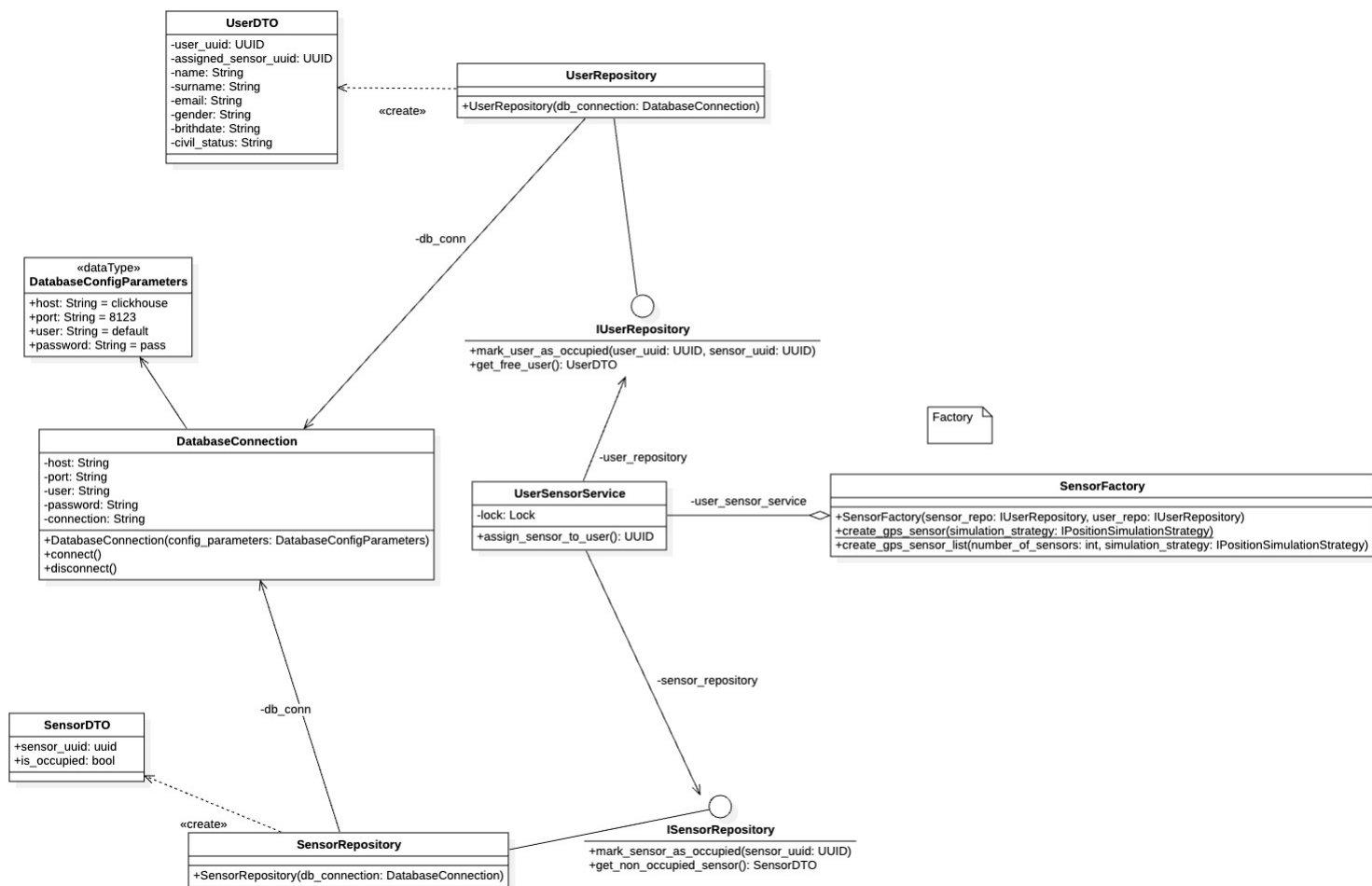


Figure 6: Factory di Sensori

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.

```

1 class IPositionSimulationStrategy(ABC):
2     @abstractmethod
3     def get_route(self):
4         pass
5
6     @abstractmethod
7     def get_delta_time(self) -> float:
8         pass
9
10    @abstractmethod
11    def get_speed(self) -> float:
12        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.

```

1 class BicycleSimulationStrategy(IPositionSimulationStrategy):
2
3     def __init__(self, graph_instance: GraphWrapper):
4         self.__bicycle_speed_approximated = 15
5         self.__delta_time_between_positions = 21
6         self.__graph_instance = graph_instance.get_graph()
7
8     def get_route(self):
9         graph_returned = self.__graph_instance
10        graph_nodes = list(graph_returned.nodes)
11        starting_node = random.choice(graph_nodes)
12        destination_node = random.choice(graph_nodes)
13
14        shortest_route = osmnx.shortest_path(
15            graph_returned,
16            starting_node,
17            destination_node,
18            weight='length'
19        )
20        route_coords = [(graph_returned.nodes[node]["y"], graph_returned.nodes[node]["x"])
21                        for node in shortest_route]
22
23        return route_coords
24
25    def get_delta_time(self) -> float:
26        return self.__delta_time_between_positions
27
28    def get_speed(self) -> float:
29        speed = self.__bicycle_speed_approximated / 3.6

```

Il funzionamento dei metodi della `BicycleSimulationStrategy` 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, `BicycleSimulationStrategy` 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:

```

1 class SensorFactory:
2     def __init__(self, sensor_repo: ISensorRepository, user_repo: IUserRepository):
3         self.__user_sensor_service = UserSensorService(sensor_repo, user_repo)
4
5     def create_gps_sensor(self, position_sender: PositionSender, simulation_strategy:
6         IPositionSimulationStrategy) -> SensorSubject:
7         uuid = self.__user_sensor_service.assign_sensor_to_user()
8         return GpsSensor(uuid, position_sender, simulation_strategy)
9
10    def create_gps_sensor_list(self, position_sender: PositionSender, simulation_strategy:
11        IPositionSimulationStrategy, number_of_sensors: int) -> List[SensorSubject]:
12        sensor_list = [self.create_gps_sensor(position_sender, simulation_strategy) for i in
13            range(number_of_sensors)]
14        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 `KafkaG` per la pubblicazione di dati serializzati, e Adapter dedicati alla gestione della serializzazione e deserializzazione `JSONG` 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 `JSONG` degli oggetti `GeoPosition`:

```

1 class PositionJsonAdapter(IJsonSerializable):
2
3     def serialize_to_json(self, position_instance: GeoPosition):
4
5         return JSON.dumps({
6             'user_uuid': position_instance.get_sensor_id(),
7             'latitude': float(position_instance.get_latitude()),
8             'longitude': float(position_instance.get_longitude()),
9             'received_at': position_instance.get_timestamp(),
10        })

```

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 `JSONG` contenente l'UUID del sensore, la latitudine, la longitudine e il timestamp.

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

```

1 class KafkaConfluentAdapter(PositionSender):
2
3     def __init__(self,
4         kafka_config: KafkaConfigParameters,
5         json_adapter_instance: "PositionJsonAdapter",
6         producer_instance: Producer):
7         super().__init__(json_adapter_instance)
8         self.__kafka_config = kafka_config
9         self.__producer = producer_instance
10
11     def send_data_to_broker(self, json_payload, sensor_id: str):
12         self.__producer.produce(self.__kafka_config.source_topic,
13             key = str(sensor_id),
14             value = json_payload.encode('utf-8'))
15         self.__producer.flush()

```

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

- `__init__(...)`: Costruttore che inizializza l'Adapter, ricevendo la configurazione di `KafkaG`, un Adapter `JSONG` per la serializzazione e un producer `KafkaG` per la gestione dell'invio dei messaggi al broker_G;
- `send_data_to_broker(...)`: Invia il payload `JSONG`, precedentemente serializzato, al topic_G `KafkaG` 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.

```

1 class JsonRowDeserializationAdapter:
2     def __init__(self, row_type_config ):
3         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.

```

1 class JsonRowSerializationAdapter:
2     def __init__(self, row_type_config_message ):
3         self.__row_type_info_message = row_type_config_message
4
5     def get_serialization_schema(self):
6         return JsonRowSerializationSchema.builder()\
7             .with_type_info(self.__row_type_info_message)\
8             .build()
9
10    def get_deserialization_schema(self):
11        return JsonRowDeserializationSchema.builder() \
12            .type_info(self.__row_type_info) \
13            .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 BicycleSimulationStrategy

- **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:**
 - `__bicycle_speed_approximated:` `float` - Velocità approssimativa della bicicletta in km/h;
 - `__delta_time_between_positions:` `float` - Intervallo di tempo in secondi tra posizioni consecutive;
 - `__graph_instance:` `Graph` - Istanza del grafo della rete stradale utilizzata per la generazione del percorso_G.
- **Operazioni:**
 - `__init__(self, graph_instance: 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 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 per l'accesso e la gestione dei dati relativi ai sensori;
 - `__UserRepository:` `IUserRepository` - Repository 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 `JSONG`. 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 `JSONG`.

4.5.2.19 PositionJsonAdapter

- **Descrizione:** Implementa un adattatore che converte oggetti `GeoPosition` in formato `JSONG`, 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 `JSONG` 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_serializer: IJsonSerializable` - Componente che si occupa della serializzazione degli oggetti `GeoPosition` in formato `JSONG`;
 - `_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_instance: IJsonSerializable)` - Costruttore che inizializza il sender con un adattatore `JSONG` 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 `JSONG` 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_instance: "PositionJsonAdapter", producer_instance: 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.

```
1 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:

```
1 {
2   "user_uuid": "UUID",
3   "latitude": "Float64",
4   "longitude": "Float64",
5   "received_at": "String"
6 }
```

4.5.3.5 Schema message elaborated

I messaggi sul topic_G messaggi hanno il seguente formato:

```
1 {
2   "user_id": "UUID",
3   "activity_id": "UUID",
4   "message_id": "UUID",
5   "message_text": "String",
6   "activity_lat": "Float64",
7   "activity_lon": "Float64",
8   "creation_time": "String",
9   "user_lat": "Float64",
10  "user_lon": "Float64"
11 }
```

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 diagrammi delle classi, suddiviso per comodità in 2 parti, rispetto alla classe `FlinkJobManager`

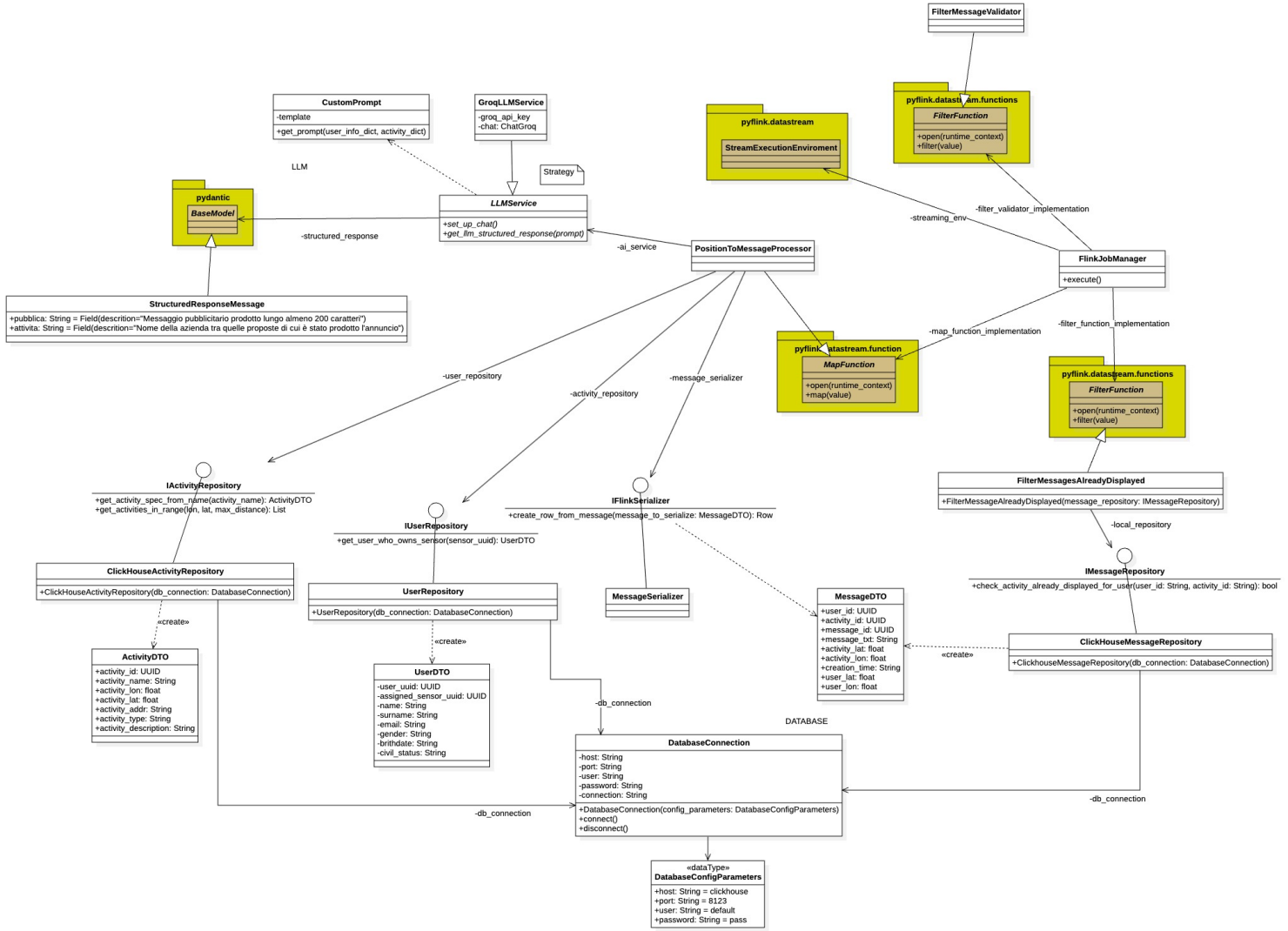


Figure 7: PositionToMessageProcessorService InBound/OutBound Ports con il Broker_G

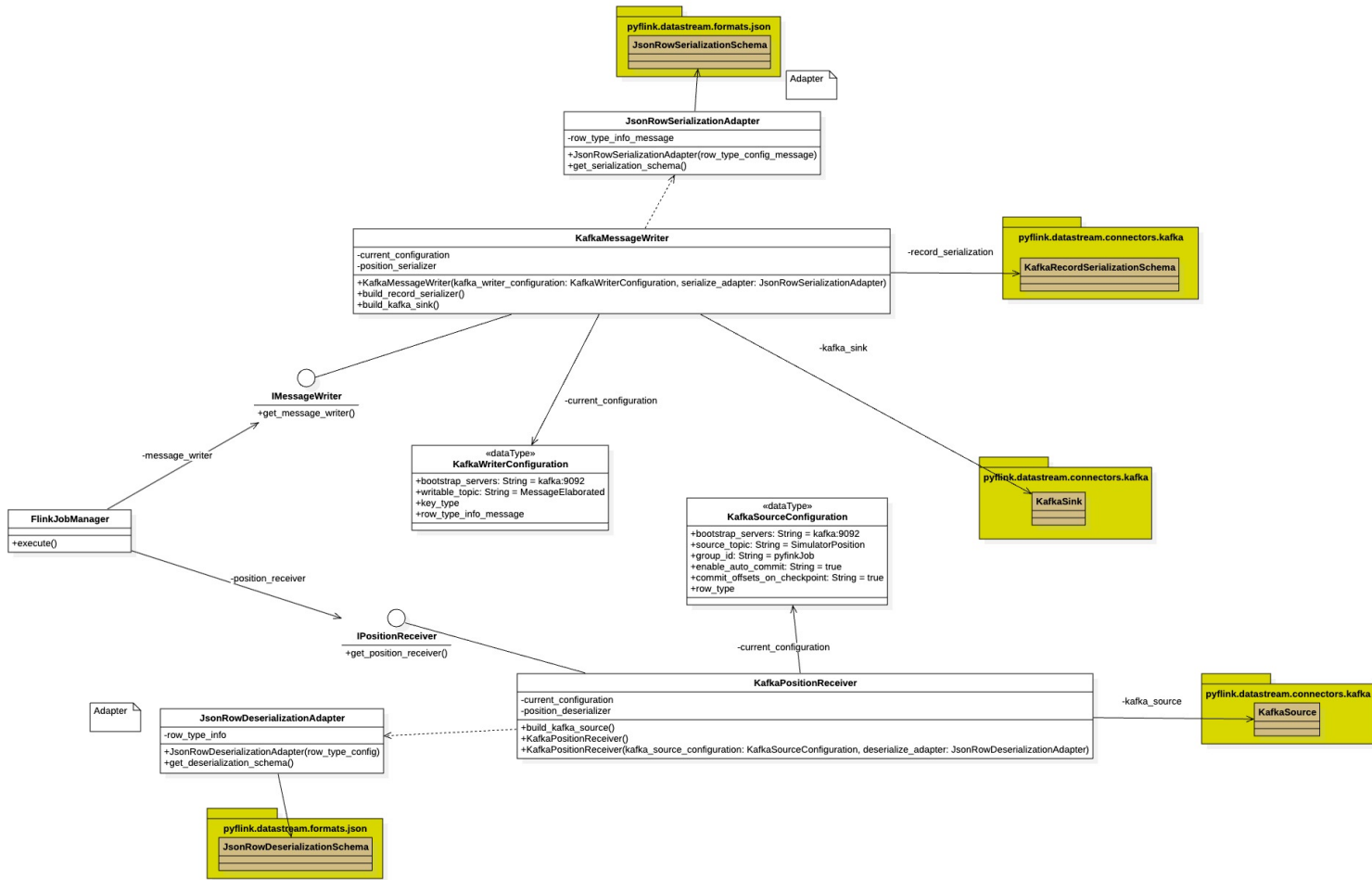


Figure 8: PositionToMessageProcessorService OutBound Ports

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 `KafkaG`:
 1. Riceve una configurazione (`KafkaWriterConfiguration`) e un adapter di serializzazione;
 2. Costruisce un serializzatore di record `KafkaG` configurandolo con il `topicG` di destinazione e gli schemi di serializzazione per chiave e valore;
 3. Crea un sink `KafkaG` 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 `KafkaG`.
- `KafkaPositionReceiver` agisce come adapter per la ricezione di posizioni da `KafkaG`:
 1. Riceve una configurazione (`KafkaSourceConfiguration`) e un adapter di deserializzazione;
 2. Costruisce una sorgente `KafkaG` configurandola con server bootstrap, `topicG`, 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 `KafkaG`.
- Grazie a questi adapter, il core dell'applicazione può interagire con `KafkaG` attraverso un'interfaccia semplificata e coerente, senza essere esposto ai dettagli implementativi della libreria `KafkaG`. Se in futuro fosse necessario sostituire `KafkaG` 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 `LLMG`), incapsularli in classi separate e renderli intercambiabili a runtime;
- Grazie a questo approccio, possiamo estendere facilmente le capacità del sistema_G aggiungendo nuovi servizi `LLMG` 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 `LLMG`, 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 `LLMG`;
 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. Dichiarare il metodo astratto `set_up_chat()` che deve essere implementato per inizializzare la comunicazione con il servizio `LLMG` specifico;
 3. Dichiarare il metodo astratto `get_llm_structured_response(prompt)` che deve essere implementato per ottenere risposte strutturate dal `LLMG`.
- `GroqLLMService` rappresenta una strategia concreta:
 1. Implementa l'interfaccia `LLMService` fornendo un'implementazione specifica per il servizio `GroqG`;

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_instance: StreamExecutionEnvironment, map_function_implementation: MapFunction, filter_validator_implementation: FilterFunction, filter_function_implementation: FilterFunction, position_receiver_instance: IPositionReceiver, message_sender_instance: 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_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 sotto-classi 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 `KafkaG` 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 `KafkaG` configurata, pronta per essere utilizzata nel pipeline `FlinkG`.

4.5.4.13 JsonRowDeserializationAdapter

- **Descrizione:** Implementa un adattatore per la deserializzazione di dati in formato `JSONG`. Incapsula la configurazione e la creazione di uno schema di deserializzazione `JSONG` per l'uso nei flussi di dati di Apache `FlinkG`;
- **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 `JSONG`.
- **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 `JSONG` configurato con le informazioni di tipo fornite. Lo schema creato può essere utilizzato per convertire stringhe `JSONG` in oggetti `FlinkG` 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 `KafkaG` nel pipeline `FlinkG`. Definisce sia i parametri di connessione sia la struttura dei dati da deserializzare;
- **Attributi:**
 - `bootstrap_servers: str` - Indirizzo e porta dei server bootstrap `KafkaG`. Il valore predefinito è "kafka:9092";
 - `source_topic: str` - Nome del topic_G `KafkaG` da cui leggere i messaggi di posizione. Il valore predefinito è "SimulatorPosition";
 - `group_id: str` - Identificatore del gruppo di consumatori `KafkaG`. 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 `FlinkG`. 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 `FlinkG` per filtrare messaggi `KafkaG` 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 `SQLG` injection (come "-", ";", comandi `SQLG` 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 `FlinkG` 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` - `RepositoryG` per accedere alle informazioni sugli utenti;
- `__activity_repository`: `IActivityRepository` - `RepositoryG` per recuperare attività commerciali nelle vicinanze dell'utente;
- `__message_serializer`: `IFlinkSerializable` - Componente per serializzare i messaggi nel formato richiesto da `FlinkG`;
- `prompt_creator`: `CustomPrompt` - Generatore di prompt per l'interazione con il servizio `LLMG`.

• 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 `FlinkG`. Configura il servizio `LLMG` 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 `LLMG` 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 `FlinkG` 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;
 - `attivit `: `str` - Campo che contiene il nome dell'azienda selezionata tra quelle proposte per cui   stato prodotto l'annuncio pubblicitario.
- **Operazioni:**
 - Nessuna operazione esplicita definita, in quanto si tratta di un modello Pydantic che fornisce automaticamente funzionalit  di validazione, serializzazione, deserializzazione e altre funzionalit  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 `FlinkG`. 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 `ClickhouseG`. 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 `ClickhouseG` utilizzata per eseguire query `queryG` sui dati dei messaggi.

- **Operazioni:**

- `__init__(self, db_connection: DatabaseConnection)` - Costruttore che inizializza il repository `ClickhouseG` con una connessione al database `ClickhouseG`;
- `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 `SQLG` 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 `SQLG` 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 `clickhouseG.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 - RepositoryG` 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:

1 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:

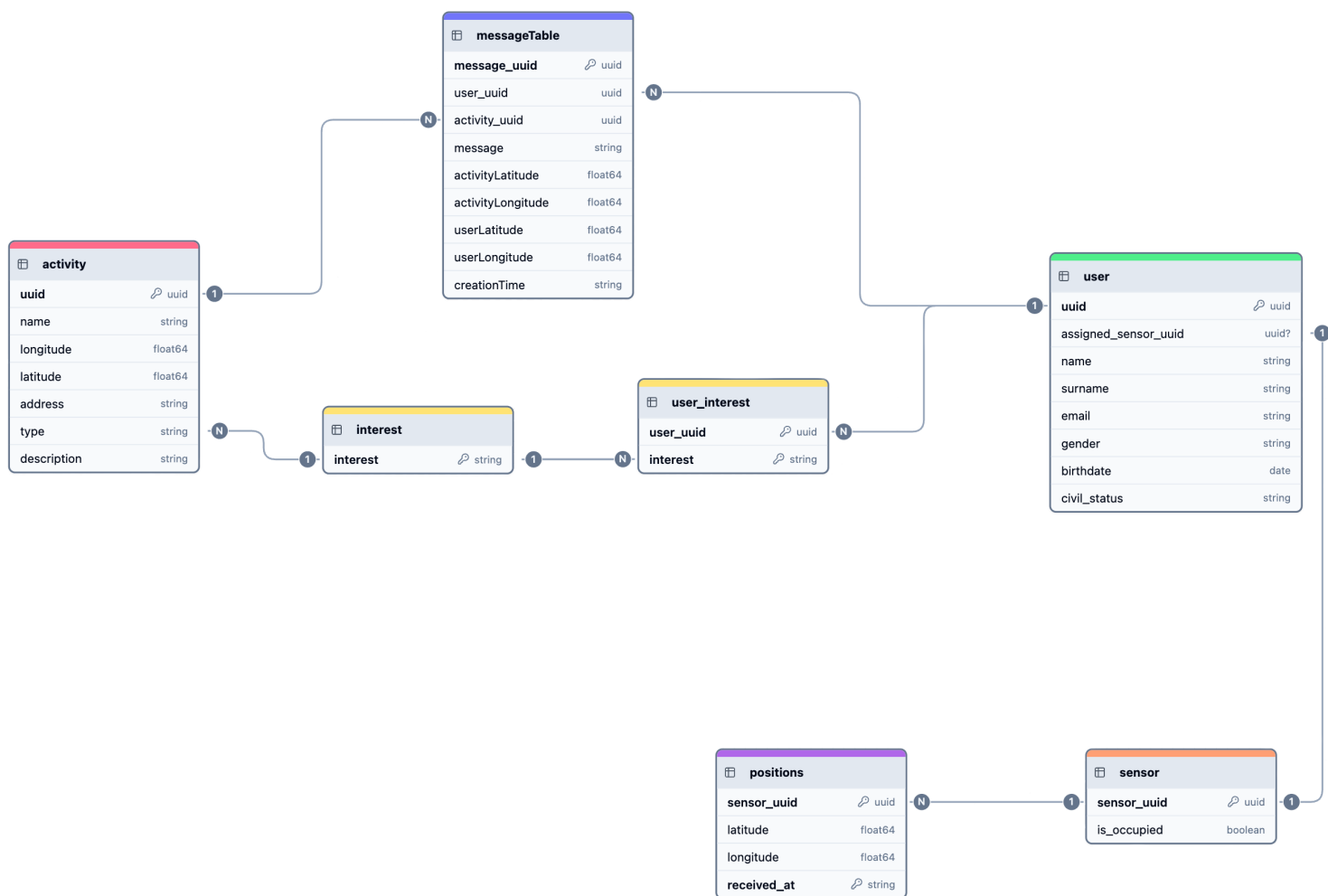


Figure 9: Schema del database_G

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


```

1      CREATE TABLE nearyou.activity(
2          activity_uuid UUID,
3          name String,
4          longitude Float64,
5          latitude Float64,
6          address String,
7          type String,
8          description String,
9          PRIMARY KEY(activity_uuid)
10     ) ENGINE = MergeTree()
11     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.

```

1      CREATE TABLE nearyou.user_interest(
2          user_uuid UUID,
3          interest String,
4          PRIMARY KEY(user_uuid, interest)
5     ) ENGINE = MergeTree()
6     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.

```

1      CREATE TABLE nearyou.messageTableKafka
2      (
3          user_uuid UUID,
4          activity_uuid UUID,
5          message_uuid UUID,
6          message String,
7          activityLatitude Float64,
8          activityLongitude Float64,
9          creationTime String,
10         userLatitude Float64,
11         userLongitude Float64
12     ) ENGINE = Kafka()
13     SETTINGS
14         kafka_broker_list = 'kafka:9092',
15         kafka_topic_list = 'MessageElaborated',
16         kafka_group_name = 'clickhouseConsumerMessage',
17         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.

```

1      CREATE TABLE nearyou.messageTable
2      (
3          user_uuid UUID,
4          activity_uuid UUID,
5          message_uuid UUID,
6          message String,
7          activityLatitude Float64,
8          activityLongitude Float64,
9          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.

```
1 CREATE TABLE nearyou.interest(
2     interest String,
3     PRIMARY KEY(interest)
4 ) ENGINE = MergeTree()
5 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.

```
1 CREATE TABLE nearyou.user(
2     user_uuid UUID,
3     assigned_sensor_uuid UUID NULL,
4     name String,
5     surname String,
6     email String,
7     gender String,
8     birthdate Date DEFAULT toDate(now()),
9     civil_status String,
10    PRIMARY KEY(user_uuid)
11 ) ENGINE = MergeTree()
12 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

```
1 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:

```
1 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

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

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

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

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

```
1 ALTER TABLE positions
2 MODIFY TTL
3     received_at + INTERVAL 7 DAY TO VOLUME 'hot',
4     received_at + INTERVAL 30 DAY TO VOLUME 'cold',
5     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:

```
1 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 come se fosse una tabella normale.

Ecco alcuni esempi:

```

1 CREATE MATERIALIZED VIEW nearyou.mv_positions TO nearyou.positions
2 AS
3 SELECT
4     user_uuid,
5     latitude,
6     longitude,
7     received_at
8 FROM nearyou.positionsKafka;
```

```

1 CREATE MATERIALIZED VIEW nearyou.mv_messageTable TO nearyou.messageTable
2 AS
3 SELECT
4     user_uuid,
5     activity_uuid,
6     message_uuid,
7     message,
8     activityLatitude,
9     activityLongitude,
10    creationTime,
11    userLatitude,
12    userLongitude
13 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:

```

1 WHERE
2     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 nascita;
- . 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.

```

1      SELECT
2          user_uuid,
3          latitude,
4          longitude,
5          received_at
6      FROM (
7          SELECT
8              user_uuid,
9              latitude,
10             longitude,
11             received_at,
12             ROW_NUMBER() OVER (PARTITION BY user_uuid ORDER BY received_at DESC
13                               ) AS row_num
14             FROM "nearyou"."positions"
15         )
16     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.

```

1      SELECT
2          m.userLatitude AS latitude,
3          m.userLongitude AS longitude,
4          m.creationTime AS creazione_time,
5          u.name AS userName,
6          u.surname AS userSurname,
7          a.name AS activityName,
8          m.message AS message
9      FROM (
10         SELECT
11             user_uuid,
12             message_uuid,
13             message,
14             activity_uuid,
15             activityLatitude,
16             activityLongitude,
17             creationTime,
```

```

18         userLatitude,
19         userLongitude,
20         ROW_NUMBER() OVER (PARTITION BY user_uuid ORDER BY toDateTime(
           creationTime) DESC) AS rn
21     FROM nearyou.messageTable
22 ) AS m
23     INNER JOIN nearyou.user u ON m.user_uuid = u.user_uuid
24     INNER JOIN nearyou.activity a ON m.activity_uuid = a.activity_uuid
25     INNER JOIN (
26         SELECT
27             user_uuid,
28             latitude,
29             longitude,
30             received_at,
31             ROW_NUMBER() OVER (PARTITION BY user_uuid ORDER BY received_at DESC
           ) AS rn
32         FROM nearyou.positions
33     ) AS p ON u.assigned_sensor_uuid = p.user_uuid AND p.rn = 1
34     WHERE
35         m.rn = 1
36         AND geoDistance(m.activityLongitude, m.activityLatitude, p.
           longitude, p.latitude) < 300
37     ORDER BY m.creationTime DESC;

```

- **Lista delle attività:** Questa query_G recupera l'elenco completo delle attività (punti di interesse) dal database_G.

```

1     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.

```

1     SELECT
2         a.name AS nome_attivita,
3         a.address AS indirizzo,
4         a.type AS tipologia,
5         a.description as descrizione,
6         COUNT(m.message_uuid) AS numero_messaggi
7     FROM nearyou.activity a
8     INNER JOIN nearyou.messageTable m ON a.activity_uuid = m.activity_uuid
9     GROUP BY a.activity_uuid, a.name, a.type, a.address, a.description
10    HAVING COUNT(m.message_uuid) > 0
11    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.

```

1     SELECT * FROM nearyou.positions
2     WHERE user_uuid = toUUID('${user_id}')
3     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.


```

1      (
2          SELECT *
3          FROM nearyou.positions
4          WHERE user_uuid = toUUID('${user_id}')
5          ORDER BY received_at ASC
6          LIMIT 1
7      )
8      UNION ALL
9      (
10         SELECT *
11         FROM nearyou.positions
12         WHERE user_uuid = toUUID('${user_id}')
13         ORDER BY received_at DESC
14         LIMIT 1
15     );

```

- **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.

```

1      SELECT
2          m.userLatitude,
3          m.userLongitude,
4          m.creationTime,
5          u.name,
6          u.surname,
7          a.name,
8          m.message
9      FROM nearyou.messageTable m
10     INNER JOIN nearyou.user u ON m.user_uuid = u.user_uuid
11     INNER JOIN nearyou.activity a ON m.activity_uuid = a.activity_uuid
12     WHERE u.assigned_sensor_uuid = toUUID('${user_id}')
13     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.

```

1      SELECT
2          u.name,
3          u.surname,
4          u.email,
5          u.gender,
6          u.birthdate,
7          u.civil_status,
8          ARRAY_AGG(ui.interest) AS interests_json
9      FROM
10         nearyou.user u
11     JOIN
12         nearyou.user_interest ui ON u.user_uuid = ui.user_uuid
13     WHERE
14         u.assigned_sensor_uuid = toUUID('${user_id}')
15     GROUP BY
16         u.name, u.surname, u.email, u.gender, u.birthdate, u.civil_status,
17         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:

```
1      SELECT *
2      FROM nearyou.positions
3      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:

```
1      {
2          "id": "extractFields",
3          "options": {
4              "delimiter": ",",
5              "format": "auto",
6              "jsonPaths": [
7                  {
8                      "alias": "interest",
9                      "path": ""
10                 }
11            ],
12            "keepTime": false,
13            "replace": false,
14            "source": "interests_json"
15        }
16    }
```

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

```
1      {
2          "id": "organize",
3          "options": {
4              "excludeByName": {
5                  "interests_json": true
6              },
7              "includeByName": {},
8              "indexByName": {},
9              "renameByName": {}
10         }
11     }
```

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

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

```

6      }
7    }

```

4.5.6.7 Connettore Clickhouse

Grafana_G si integra con Clickhouse_G attraverso un connettore nativo specifico, configurato come data-source all'interno della piattaforma:

```

1  {
2    "name": "ClickHouse",
3    "type": "grafana-clickhouse-datasource",
4    "uid": "ee5wustcp8zr4b",
5    "jsonData": {
6      "defaultDatabase": "nearyou",
7      "port": 9000,
8      "host": "clickhouse",
9      "username": "default",
10     "tlsSkipVerify": false
11   },
12   "secureJsonData": {
13     "password": "pass"
14   }
15 }

```

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:

```

1  {
2    "refresh": "10s"
3  }

```

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 preconfigurate.

- **Provisioning del datasource Clickhouse_G:**

```

1     apiVersion: 1
2     datasources:
3       - name: Clickhouse
4         type: grafana-clickhouse-datasource
5         uid: "ee5wustcp8zr4b"
6         jsonData:
7           defaultDatabase: nearyou
8           port: 9000
9           host: clickhouse
10          username: default
11          tlsSkipVerify: false
12          secureJsonData:
13            password: pass

```

- Provisioning delle dashboard_G:

```

1     apiVersion: 1
2     providers:
3       - name: "Dashboard provider"
4         orgId: 1
5         type: file
6         disableDeletion: false
7         updateIntervalSeconds: 10
8         options:
9           path: /var/lib/grafana/dashboards
10          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 `LLMG` 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 `GroqG` 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`

```

1 class FlinkJobManager:
2     def __init__(self,
3                 streaming_env_instance : StreamExecutionEnvironment,
4                 map_function_implementation : MapFunction,
5                 filter_validator_implementation : FilterFunction,
6                 filter_function_implementation : FilterFunction,
7                 position_receiver_instance: IPositionReceiver,
8                 message_sender_instance: IMessageWriter
9                 ):
10         self.__streaming_env = streaming_env_instance

```

```

11     self.__populated_datastream = self.__streaming_env.from_source(
12         position_receiver_instance.get_position_receiver(),
13         WatermarkStrategy.
14             for_monotonous_timestamps()
15         ,
16         "Positions Source")
17     self.__keyed_stream = self.__populated_datastream.key_by(lambda x: x[0], key_type=
18         Types.STRING())
19     self.__validated_stream = self.__keyed_stream.filter(filter_validator_implementation
20 )
21     self.__mapped_stream = self.__validated_stream.map(map_function_implementation,
22         output_type=KafkaWriterConfiguration().
23             row_type_info_message)
24     self.__filtered_stream = self.__mapped_stream.filter(filter_function_implementation)
25     self.__filtered_stream.sink_to(message_sender_instance.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

```

1 class SensorFactory:
2     def __init__(self, sensor_repository: ISensorRepository, user_repository:
3         IUserRepository):
4         self.__sensor_repository = sensor_repository
5         self.__user_repository = user_repository
6
7     def create_gps_sensor(self, position_sender: PositionSender,
8         simulation_strategy: IPositionSimulationStrategy) -> GpsSensor:
9         # 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 unitari. Utilizzando librerie come unittest.mock, è possibile sostituire facilmente le dipendenze reali con mock:

Listing 3: Test di un componente con Constructor Injection

```

1 def setUp(self):
2     self.mock_sensor_repository = Mock(spec=ISensorRepository)
3     self.mock_user_repository = Mock(spec=IUserRepository)
4     self.mock_position_sender = Mock(spec=PositionSender)
5     self.mock_simulation_strategy = Mock(spec=IPositionSimulationStrategy)
6     self.mock_user_sensor_service = Mock(spec=UserSensorService)
7
8     self.patcher = patch('Models.SensorFactory.UserSensorService',
9         return_value=self.mock_user_sensor_service)
10    self.patcher.start()
11

```

```

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 = BicycleSimulationStrategy(
    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:** `kafkaG` - Nome host assegnato al container_G all'interno della rete Docker_G;

- **container_name:** `kafkaG` - 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 conatterà, 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 `KafkaG` 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 `KafkaG` 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 `kafkaG`:
 - . 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 `kafkaG: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 geospaziali.

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

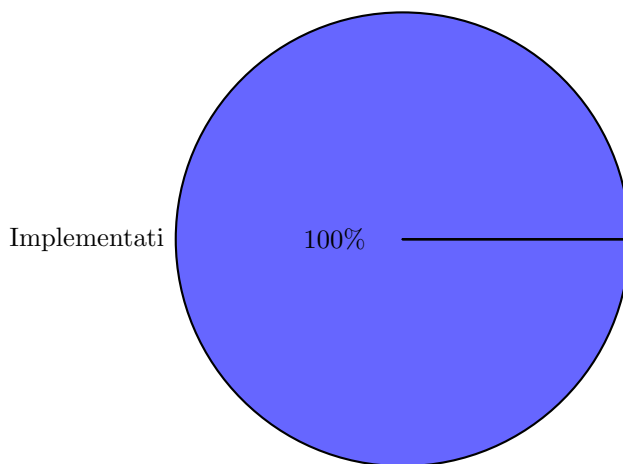


Figure 10: Stato dei requisiti funzionali obbligatori

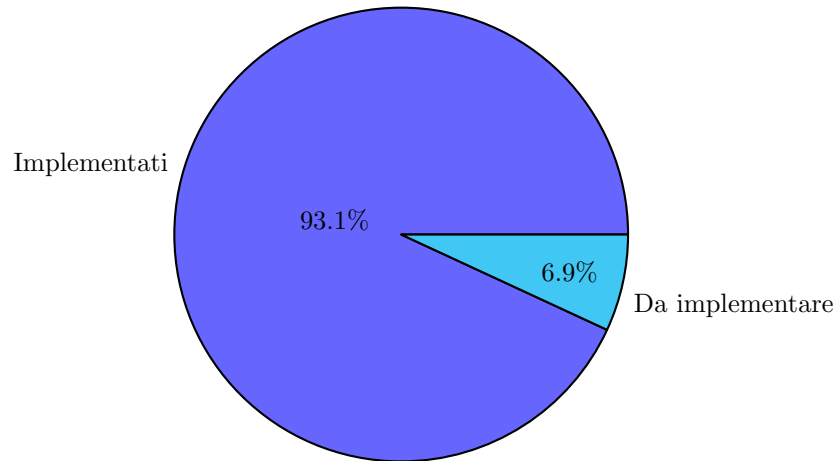


Figure 11: Stato dei requisiti funzionali totali

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.