Specifica Tecnica

NearYou
Smart custom advertising platform

sevenbits.swe.unipd@gmail.com

Registro modifiche

Versione	Data	Autore	Verificatore	Descrizione
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
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

Elenco delle figure

Elenco delle tabelle

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, i design pattern utilizzati. Saranno inclusi diagrammi delle classi, e una spiegazione dettagliata dei design pattern 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_v1.0.0*.
In sintesi, il documento intende essere un punto di riferimento essenziale per tutti gli attori coinvolti nel ciclo di vita del progetto, 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_v1.0.0.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_v1.0.0*

1.3.2 Riferimenti informativi

- *Glossario_v1.0.0*
- *Analisi_dei_Requisiti_v1.0.0*
- 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 Creazionali - SWE 2024-25
  https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Creazionali.pdf
  (Consultato: 2025-02-26).

- Design Pattern 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 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 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;
- **Apache Kafka**: Sistema di messaggistica distribuito per la comunicazione tra componenti;
- **Apache Flink**: Framework di elaborazione dati in tempo reale;
- **ClickHouse**: Database colonnare ad alte prestazioni;
- **Grafana**: Piattaforma di visualizzazione dei dati in tempo reale;
- **Docker**: Sistema 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 l'ampio ecosistema di framework 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 viene impiegato per:

- Sviluppo del simulatore di spostamenti utente;
- Implementazione della logica di elaborazione dati;
- Interazione con i servizi esterni e le API;
- Gestione della persistenza dei dati;
- Applicazione degli algoritmi di selezione dei POI rilevanti.

2.2.1.3 Dipendenze

- **ClickHouse Connect**:
  - **Descrizione**: Libreria client per l'interazione con il database ClickHouse, permettendo operazioni di query e gestione dei dati;
  - **Versione**: 0.6.8;
  - **Documentazione**: https://clickhouse.com/docs/integrations/python (Consultato: 2025-03-02).
- **PyFlink**:
  - **Descrizione**: API Python di Apache Flink 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 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 per l'API Groq, utilizzato per la generazione di contenuti tramite LLM;
  - **Versione**: 0.4.2;
  - **Documentazione**: https://console.groq.com/docs/libraries (Consultato: 2025-03-02).
- **Confluent Kafka**:
  - **Descrizione**: Libreria per l'interazione con Apache Kafka, 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;

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

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

- **pytest**:
  - **Descrizione**: Framework per test 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 relazionali, utilizzato nel contesto di ClickHouse per definire lo schema del database e per interrogare i dati.

2.2.2.1 Specifiche

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

2.2.2.2 Ruolo nel progetto

In NearYou, SQL viene utilizzato per:

- Definizione dello schema del database;
- Interrogazione dei dati per la visualizzazione;
- Creazione di query 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;
- Workflow CI/CD;
- Configurazione dei servizi.

2.2.3.4 JSON

JSON 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 di visualizzazione;
- Comunicazione con i servizi API 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 Kafka e delle loro configurazioni;
- Monitoraggio dello stato dei nodi nel sistema distribuito;
- Coordinamento delle operazioni distribuite tra i componenti;
- Gestione delle elezioni dei leader per le partizioni Kafka.

2.3.2 Apache Kafka

Sistema 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 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 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 è utilizzato per:

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

2.3.4 ClickHouse

Database colonnare progettato per l'analisi OLAP (OnLine Analytical Processing) con prestazioni eccezionali su grandi volumi di dati. Le caratterische principali di ClickHouse sono:

- Architettura colonnare per interrogazioni analitiche efficienti;
- Supporto a funzioni geospaziali per calcoli di distanza;
- Integrazione nativa con Kafka 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 è utilizzato per:

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

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 è utilizzato per:

- Containerizzazione dei diversi componenti del sistema;
- 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 adotta un approccio esagonale (Hexagonal Architecture, noto anche come Ports and Adapters) 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 come Flink, Kafka, ClickHouse e Grafana tramite porte (interface) e adattatori (infrastructure), favorendo una netta suddivisione delle responsabilità.

Figure 1: Architettura logica del prodotto con modello esagonale

3.1 Pattern di architettura esagonale

Il pattern architetturale Ports and Adapters 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, 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 (punti di interesse, utenti, posizioni e messaggi pubblicitari);

- **Porte (Ports)**: Interfacce che definiscono come il dominio interagisce con l'esterno. Si distinguono in:
  . **Inbound ports**: Interfacce che espongono le funzionalità del dominio verso l'esterno, come l'interfaccia `IPositionProcessor` che riceve posizioni da elaborare;
  . **Outbound ports**: Interfacce che definiscono come il dominio può utilizzare servizi esterni, come `IRepository`, `IMessagePublisher` e `ILlmService`.
- **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, come `KafkaConsumerAdapter`;
  . **Outbound adapters**: Adattatori che implementano le interfacce di uscita collegandole a tecnologie specifiche, come `ClickHouseRepository` o `GroqLlmAdapter`.

3.2 Servizi principali e loro componenti

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

- **Generatore di posizioni GPS**:
  . **Core domain**: Implementa la logica di simulazione del movimento di utenti nello spazio;
  . **Outbound ports**: Non dispone di inbound ports poiché non riceve input esterni ma solo genera dati;
  . **Outbound ports**: Include interfacce come `IPositionPublisher` per pubblicare le posizioni generate;
  . **Adapters**: Implementa adattatori come `KafkaPositionPublisherAdapter` che converte le posizioni generate in messaggi Kafka.
- **Elaboratore di posizioni / Generatore di messaggi**:
  . **Core domain**: Contiene la logica di identificazione della prossimità e generazione di messaggi pubblicitari;
  . **Inbound ports**: Include interfacce come `IPositionReceiver` che ricevono eventi di posizione;
  . **Outbound ports**: Comprende interfacce verso repository (`IUserRepository`, `IActivityRepository`), servizi esterni (`ILlmService`) e publisher di messaggi (`IMessagePublisher`);
  . **Adapters**: Implementa adattatori per tecnologie specifiche, come `KafkaPositionReceiverAdapter`, `ClickHouseRepositoryAdapter` 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.

3.3 Dataflow

Il flusso dei dati attraverso l'architettura esagonale segue un percorso 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 generatore crea oggetti `GeoPosition` che rappresentano le coordinate degli utenti;
   . Questi oggetti vengono passati attraverso l'outbound port `IPositionPublisher`;
   . L'adapter `KafkaPositionPublisherAdapter` serializza i dati in formato JSON e li pubblica sul topic Kafka `posizioni`.
2. **Consumo delle posizioni**:
   . L'inbound adapter `KafkaPositionReceiverAdapter` del servizio di elaborazione sottoscrive il topic Kafka `posizioni`;

. I messaggi ricevuti vengono deserializzati e convertiti in oggetti di dominio `UserPosition`;

. Questi oggetti vengono passati all'inbound port `IPositionProcessor` per l'elaborazione.

3. **Elaborazione e generazione di messaggi**:

. Il core domain valuta la prossimità dell'utente rispetto ai punti di interesse, utilizzando l'outbound port `IActivityRepository` per recuperare le attività nelle vicinanze;

. In caso di rilevamento di prossimità, il sistema richiede informazioni sull'utente tramite l'outbound port `IUserRepository`;

. Con i dati dell'utente e del punto di interesse, il core domain richiede un messaggio personalizzato tramite l'outbound port `ILlmService`;

. L'adapter `GroqLlmAdapter` comunica con il servizio LLM esterno e restituisce il messaggio generato;

. Il messaggio personalizzato viene incapsulato in un oggetto `AdvertisingMessage` del dominio.

4. **Pubblicazione del messaggio pubblicitario**:

. L'oggetto `AdvertisingMessage` viene passato all'outbound port `IMessagePublisher`;

. L'adapter `KafkaMessagePublisherAdapter` serializza il messaggio e lo pubblica sul topic Kafka `messaggi`.

5. **Persistenza e visualizzazione**:

. ClickHouse, attraverso il connettore Kafka nativo, consuma e archivia i messaggi dal topic `messaggi`;

. Grafana interroga ClickHouse per recuperare e visualizzare i dati in tempo reale attraverso dashboard interattive.

Il flusso dei dati è progettato per essere asincrono, garantendo la scalabilità e la resilienza del sistema. Ogni componente può funzionare indipendentemente, con Kafka che funge da buffer di messaggi affidabile tra i vari stadi del processo.
Gli adattatori si occupano d'interfacciarsi con l'esterno: il simulatore di posizioni produce eventi JSON su Kafka, successivamente elaborati da Flink 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 esterno crea i messaggi personalizzati che confluiscono nel dominio. La persistenza e la consultazione storica avvengono tramite ClickHouse, mentre Grafana rende immediatamente disponibili tali informazioni agli utenti.
Questo utilizzo di Kafka come backbone di comunicazione sostiene la natura asincrona ed event-driven del sistema, 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.

4 Integrazione Architettura logica e Architettura di sistema

4.1 Descrizione

Le due architetture, la Kappa Architecture e l'architettura esagonale, rappresentano due prospettive differenti dello stesso sistema. Mentre la Kappa Architecture si riferisce all'implementazione concreta del codice e al flusso continuodi dati in tempo reale, l'architettura esagonale evidenzia la separazione logicatra il core di business e le interfacce esterne. Nonostante l'approccio e la terminologia differiscano, i componenti del sistema 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 ClickHouse per la parte di interfaccia utente.

4.2 Mappatura dei componenti

Kappa Architecture	Architettura Esagonale	Ruolo nel Progetto
Generazione di dati	Output Port (Generazione Posizioni)	Pubblica le posizioni generate sul topic Kafka `raw_positions`.
Gestione messaggi	Input Port (Elaborazione Posizioni)	Sistema di topic Kafka usato per lo scambio di informazioni fra i vari componenti.
Elaborazione dei dati	Core Service (Elaborazione Posizioni)	Si occupa della generazione dei messaggi ed esegue la logica di trasformazione e conseguente filtraggio dei dati.
Storage	Output Adapter (Elaborazione Posizioni)	Memorizza i dati elaborati in un database (ClickHouse in questo caso).

Table 2: Mappatura dei componenti tra Kappa Architecture e Architettura Esagonale

5 Architettura del Sistema

5.1 Panoramica architetturale

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

5.2 K-Architecture: Event Streaming Platform

Il cuore dell'architettura del progetto è rappresentato dalla piattaforma di streaming basata su Flink, che implementa il pattern di K-Architecture (Kappa Architecture). Questo pattern rappresenta un'evoluzione della Lambda Architecture, eliminando la necessità del dual-path processing (batch e speed layer) a favore di un unico layer di stream processing.

5.2.1 Motivazioni della scelta architetturale

5.2.1.1 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.

5.2.1.2 Benefici sul disaccoppiamento

Tra i benefici sul disaccoppiamento c'è la possibilità di sviluppare e scalare indipendentemente i componenti del sistema, garantendo una maggiore flessibilità.

5.2.1.3 Ottimizzazione del codebase

La semplificazione della pipeline di dati permette di ridurre la complessità del codice e di semplificare la manutenzione.

5.2.1.4 Gestione della consistenza dei dati

La K-Architecture garantisce la coerenza dei dati in tempo reale, evitando problemi di sincronizzazione e di versionamento.

5.2.1.5 Tecniche di riduzione della latenza

La riduzione della latenza è garantita dalla gestione dei dati in tempo reale, senza la necessità di processi batch.

5.2.2 Componenti principali

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

- Generazione di dati;
- Gestione messaggi;
- Elaborazione dei dati;
- Storage;
- Visualizzazione.

5.2.2.1 Generazione di dati
Responsabile della produzione dei dati di posizione degli utenti in formato JSON, pubblicati su Kafka. Per garantire dati più realistici, il simulatore introduce un parametro velocità che varia fra le tipologie di simulazione.

5.2.2.2 Gestione messaggi
Basato su Apache Kafka, gestisce la comunicazione asincrona tra i microservizi, garantendo la scalabilità e la resilienza del sistema grazie alla gestione dei topic e delle partizioni con le chiavi e la replicazione sul database ClickHouse.

5.2.2.3 Elaborazione dei dati
Implementato con Apache Flink, elabora i dati valutando la prossimità dei punti d'interesse e interagendo con l'LLM per generare annunci personalizzati.

5.2.2.4 Storage
Supportato dal database ClickHouse, memorizza i dati in tabelle colonnari ad alte prestazioni, consentendo query analitiche rapide grazie all'ottimizzazione per letture intensive.

5.2.2.5 Visualizzazione
Basato su Grafana, costituisce una soluzione di visualizzazione dei dati su una mappa e facilita l'estensione di tale interfaccia sfruttando i dati già presenti. Sfrutta inoltre il connettore nativo di ClickHouse, permettendo l'integrazione e l'analisi in tempo reale delle informazioni.

5.2.3 Flusso di dati end-to-end

1. Il simulatore genera dati di posizione e li invia a Kafka, includendo informazioni sul timestamp e sull'ID utente.
2. Flink legge i topic, rileva la prossimità dei punti d'interesse e genera un messaggio tramite l'LLM.
3. L'LLM elabora i dettagli dell'evento, producendo un messaggio personalizzato, poi memorizzato in ClickHouse.
4. Grafana interroga ClickHouse e consente la visualizzazione in tempo reale tramite dashboard di attività, utenti, relative posizioni e annunci generati.

5.2.4 Interfacce tra componenti

Le comunicazioni tra componenti avvengono su Kafka, con dati strutturati in formato JSON. Per quanto riguarda l'integrazione fra Kafka e ClickHouse è presente un connettore nativo che permette inoltre di creare delle materialized view per la storicizzazione dei dati. Per Grafana è sempre presente un connettore nativo per ClickHouse che permette di interrogare il database e visualizzare i dati in tempo reale.

5.3 Implementazione tecnica dei componenti principali

5.3.0.1 Apache Kafka

5.3.0.1.1 Topic e partitioning

In questo progetto si utilizzano due topic:

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

5.3.0.1.2 Producer e Consumer

5.3.0.1.3 Message keys

Le chiavi dei messaggi (key) determinano la partizione Kafka 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.

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

5.3.0.1.4 Integrazione con Flink keyed stream

All'interno del job Flink, 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.

5.3.0.1.5 Schema topic simulator position

I dati inviati dal producer sul topic `posizioni` seguono questa struttura JSON:

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

5.3.0.1.6 Schema message elaborated

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

```
{
    "user_uuid": "UUID",
    "activity_uuid": "UUID",
    "message_uuid": "UUID",
    "message": "String",
    "activityLatitude": "Float64",
    "activityLongitude": "Float64",
    "creationTime": "String",
    "userLatitude": "Float64",
    "userLongitude": "Float64"
}
```

5.3.0.1.7 Kafka poisoning

- **Descrizione del problema**
  Il sistema di stream processing Kafka risulta potenzialmente vulnerabile ad un attaccante che inserisca dati falsi o malformati al fine di alterare il comportamento del sistema, 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 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 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 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 TLS per la comunicazione sensori-sistema
    - **Descrizione**:
      Il protocollo 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 utilizza una chiave di cifratura asimmetrica certificata per stabilire la comunicazione iniziale per poi utilizzare cifratura simmetrica per il resto della sessione. Apache Kafka dispone inoltre della possibilità di applicare 2-way TLS per introdurre un'ulteriore autenticazione del client.
    - **Requisiti implementazione**:
      Il protocollo TLS è già implementato all'interno di Apache Kafka è pertanto semplicemente necessario abilitarlo ed inserire i certificati richiesti. È possibile adottare sia certificati interni che certificati garantiti da una Certification Authority.
  - Autenticazione sensori mediante SASL
    - **Descrizione**:
      Il protocollo 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 SASL è già implementato all'interno di Apache Kafka è pertanto semplicemente necessario abilitarlo ed inserire i certificati richiesti. È possibile adottare sia certificati interni che certificati garantiti da una Certification Authority.
  - 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. Ogni regola definisce per un client o gruppo di client se questi sia autorizzato o meno a produrre o consumare elementi di un topic.
    - **Requisiti implementazione**:
      Apache Kafka dispone di un sistema integrato di gestione dei permessi ed è quindi semplicemente necessario definire un file di configurazione che elenchi le policies che si intende adottare.

- **Conclusioni**
  La validazione dei dati a livello di codice è stata adottata per mitigare il rischio di *Kafka poisoning*, in quanto non erano richieste misure di sicurezza avanzate dal proponente. 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. 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.

5.3.0.2 Apache Flink

5.3.0.2.1 Stream processing jobs

Nell'architettura NearYou, Flink gestisce un job di streaming strutturato secondo il paradigma DataStream API, 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 del processing dei dati. Questo manager riceve i dati di posizione dagli utenti tramite Kafka, 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;
- Raggruppamento (key-by) per identificatore utente;
- Applicazione di funzioni di mapping per la generazione dei messaggi;
- Filtraggio dei messaggi già visualizzati;
- Pubblicazione dei risultati su un topic Kafka 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. L'utilizzo della DataStream API permette inoltre di definire operazioni di trasformazione in modo dichiarativo, aumentando la leggibilità del codice e facilitando la manutenzione.

5.3.0.2.2 Elaborazione dati e pattern di progettazione

Il cuore dell'elaborazione dati in Flink 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 di utenti per recuperare informazioni demografiche e preferenze;
- Repository di attività per individuare punti di interesse nelle vicinanze;
- Servizio LLM 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.
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.

5.3.0.2.3 Integrazione con componenti esterni

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

- **Integrazione con Kafka**: Attraverso i connettori KafkaPositionReceiver e KafkaMessageWriter, Flink legge le posizioni degli utenti dal topic "SimulatorPosition" e pubblica i messaggi elaborati sul topic "MessageElaborated". La DataStream API fornisce connettori nativi per Kafka che semplificano questa integrazione, garantendo la consistenza dei tipi di dati e la corretta gestione delle configurazioni.
- **Interazione con ClickHouse**: Il rilevamento di prossimità ai punti di interesse nel raggio di generazione non avviene direttamente in Flink, bensì delegata a ClickHouse attraverso query geospaziali ottimizzate che sfruttano la funzione nativa geoDistance. Questo approccio sfrutta le capacità di calcolo geospaziale già presenti nel database, ottimizzando così le performance del sistema.
- **Comunicazione con servizi LLM**: Flink orchestra l'interazione con il servizio Groq 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.

5.3.0.2.4 Serializzazione e deserializzazione dei messaggi

La gestione della serializzazione e deserializzazione è fondamentale nell'architettura Flink per garantire l'efficiente trasferimento dei dati tra i componenti del sistema streaming. Il sistema implementa serializzatori personalizzati che garantiscono coerenza e integrità dei dati durante l'elaborazione.
Per i messaggi di posizione, viene utilizzato un deserializzatore JSON conforme allo schema del topic `posizioni`:

```
    {
        "id": "550e8400-e29b-41d4-a716-446655440000",
        "latitude": 45.4095,
        "longitude": 11.8767,
        "received_at": "2025-03-15T14:22:36"
    }
```

Analogamente, per i messaggi elaborati viene implementato un serializzatore che converte gli oggetti di dominio nel formato JSON atteso dal topic `messaggi`:

```
    {
        "user_uuid": "550e8400-e29b-41d4-a716-446655440000",
        "activity_uuid": "a3b8d425-2b60-4ad7-becc-bedf2ef860a1",
        "message_uuid": "7dac4d54-fa3f-42b2-9383-165228e9d1bc",
        "message": "Ciao! Ti trovi a soli 50 metri da Caff\‘e Pedrocchi. Vieni a provare il
            loro caff\‘e speciale, perfetto per una pausa rinfrescante!",
        "activityLatitude": 45.4084,
        "activityLongitude": 11.8762,
        "creationTime": "2025-03-15T14:22:40",
        "userLatitude": 45.4095,
        "userLongitude": 11.8767
    }
```

5.3.0.3 Simulatore posizioni

Il simulatore di posizioni è un componente fondamentale dell'architettura che simula i dati GPS degli utenti, consentendo di testare e dimostrare il funzionamento del sistema senza richiedere dispositivi fisici reali.

5.3.0.3.1 Strategie di movimento

Il simulatore implementa il pattern Strategy per definire diverse modalità di movimento. L'interfaccia `IPositionSimulationStrategy` definisce il contratto comune per tutte le strategie di simulazione:

```
class IPositionSimulationStrategy(ABC):
    @abstractmethod
    def simulate_position_live_update(self, sensor_istance):
        pass
```

La strategia principale implementata, `BycicleSimulationStrategy`, simula il movimento di una bicicletta su un grafo stradale reale utilizzando la libreria OSMnx. Questa strategia:

- Seleziona casualmente nodi nel grafo stradale per creare percorsi realistici;
- Utilizza una velocità parametrizzabile (default: 10-20 km/h);
- Applica interpolazione lineare tra i punti del percorso;
- Consente configurazione temporale degli aggiornamenti di posizione.

5.3.0.3.2 Generazione dati JSON

La serializzazione dei dati avviene attraverso il pattern Adapter, implementato dalla classe `PositionJsonAdapter` che converte gli oggetti `GeoPosition` in formato JSON:

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

Questo formato è compatibile con le aspettative del topic Kafka `SimulatorPosition` e della tabella ClickHouse corrispondente. L'adapter implementa l'interfaccia `IJsonSerializable`, garantendo uniformità nella serializzazione di diversi tipi di dati.

5.3.0.3.3 Configurazione del simulatore

Il simulatore è altamente configurabile attraverso diversi parametri:

- **Delta tempo**: Intervallo temporale tra aggiornamenti di posizione;
- **Range di velocità**: Valori minimi e massimi per la velocità di spostamento;
- **Area geografica**: Delimitazione dell'area in cui generare percorsi;
- **Numero di sensori**: Quantità di sensori simulati simultaneamente.

La configurazione sfrutta la libreria OSMnx per caricare grafi stradali reali. Il sistema utilizza un grafo precaricato dell'area di interesse per ottimizzare le performance di avvio:

```
graph_istance = GraphWrapper().get_graph()
strategy_simulation = BycicleSimulationStrategy()
sensor_istance = sensor_factory.create_sensor(sensor_uuid, strategy_simulation)
```

5.3.0.3.4 Tipologie di dati generati

Il simulatore produce dati con le seguenti caratteristiche:

- **Continuità spaziale**: I punti generati seguono percorsi realistici lungo le strade del grafo OSM;
- **Variabilità temporale**: Gli aggiornamenti rispettano intervalli di tempo configurabili;
- **Coerenza di velocità**: La distanza tra punti consecutivi è proporzionale alla velocità simulata.

La classe `GeoPosition` rappresenta la struttura dati fondamentale per le posizioni:

```
    class GeoPosition:
        def __init__(self, sensor_id, latitude, longitude, timestamp):
            self.__sensor_id = sensor_id
            self.__latitude = latitude
            self.__longitude = longitude
            self.__timestamp = timestamp
```

5.3.0.4 ClickHouse

ClickHouse è un sistema di gestione database colonnare open-source progettato specificamente per carichi analitici. Nel contesto di NearYou, è stato adottato per la sua capacità di processare ed analizzare in tempo reale grandi volumi di dati geospaziali mantenendo latenze minime e offrendo funzionalità avanzate di partizione e gestione del ciclo di vita dei dati.

5.3.0.4.1 Architettura MergeTree

Il cuore di ClickHouse è il motore MergeTree, fondamentale per le prestazioni del sistema NearYou. A differenza dei database tradizionali, MergeTree è ottimizzato per l'inserimento di grandi volumi di dati e query analitiche complesse grazie alle seguenti caratteristiche:

- **Archiviazione colonnare**: I dati sono organizzati per colonne anziché per righe, permettendo:
  . Compressione più efficiente (10-100x rispetto ai database row-oriented);
  . I/O ridotto quando le query selezionano solo un sottoinsieme di colonne;
  . Elaborazione vettoriale che sfrutta le istruzioni SIMD del processore.
- **Organizzazione in parti (parts)**: I dati vengono inseriti in parti separate che vengono poi fuse in background:
  . Le nuove scritture avvengono in parti temporanee separate;
  . Un processo di merge periodico unisce parti piccole in parti più grandi;
  . Questo processo consente inserimenti massivi senza bloccare le query di lettura.
- **Indici sparsi**: Ogni parte contiene un indice sparso per le colonne di ordinamento:
  . L'indice divide i dati in granuli (blocchi) di 8192 righe;
  . Per ogni granulo, vengono memorizzati i valori min/max delle chiavi di ordinamento;
  . Durante l'esecuzione delle query, interi granuli possono essere saltati.
- **Partizionamento primario**: I dati vengono suddivisi fisicamente in base a una chiave di partizione:

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

  . Ogni partizione è memorizzata come directory separata;
  . Le query possono escludere rapidamente intere partizioni non pertinenti;
  . Facilita operazioni di manutenzione come eliminazione o spostamento di dati storici.
- **Ordinamento primario**: I dati all'interno di ogni partizione sono fisicamente ordinati:

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

  . Accelera ricerche per range su queste colonne;
  . Migliora la compressione raggruppando valori simili;
  . Definisce l'ordine di costruzione degli indici sparsi.

Nel contesto di NearYou, questa architettura consente di gestire efficacemente:

- Migliaia di aggiornamenti di posizione al secondo;

- Query analitiche complesse come il calcolo di distanze geospaziali;
- Aggregazioni per dashboard in tempo reale;
- Interrogazioni su dati storici con latenze contenute.

Il ciclo di vita di un dato in MergeTree segue questi passi:

1. Inserimento in memoria come parte temporanea;
2. Scrittura su disco quando la parte raggiunge una certa dimensione;
3. Fusione periodica con altre parti per ottimizzare lo storage;
4. Eventuale eliminazione tramite TTL o operazioni manuali.

5.3.0.4.2 Schema del database

Lo schema del database comprende le seguenti tabelle principali:

- **positions**: Archivia le posizioni GPS degli utenti con timestamp.

```
CREATE TABLE positions (
    sensor_id UUID,
    latitude Float64,
    longitude Float64,
    received_at DateTime64(3),
    _timestamp DateTime DEFAULT now()
) ENGINE = MergeTree()
PARTITION BY toYYYYMMDD(received_at)
ORDER BY (sensor_id, received_at);
```

  Questa tabella memorizza i dati di posizione degli utenti ricevuti dal simulatore, permettendo di tracciare gli spostamenti nel tempo.

- **activities**: Contiene i punti di interesse con coordinate geografiche e metadati.

```
CREATE TABLE activities (
    activity_id UUID,
    name String,
    category String,
    description String,
    latitude Float64,
    longitude Float64,
    created_at DateTime,
    updated_at DateTime DEFAULT now()
) ENGINE = MergeTree()
ORDER BY activity_id;
```

  Questa tabella archivia le informazioni sui punti di interesse come negozi, ristoranti e attrazioni turistiche che possono generare messaggi pubblicitari personalizzati.

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

```
CREATE TABLE users (
    user_id UUID,
    name String,
    age UInt8,
    gender String,
    interests Array(String),
    preferences Array(String),
    created_at DateTime DEFAULT now()
) ENGINE = MergeTree()
```

```
        ORDER BY user_id;
```

Questa tabella contiene i dati demografici e le preferenze degli utenti utilizzati per personalizzare i messaggi pubblicitari.

- **messages**: Memorizza i messaggi pubblicitari generati.

```
        CREATE TABLE messages (
            message_id UUID DEFAULT generateUUIDv4(),
            user_id UUID,
            activity_id UUID,
            content String,
            user_lat Float64,
            user_lon Float64,
            activity_lat Float64,
            activity_lon Float64,
            created_at DateTime,
            _timestamp DateTime DEFAULT now()
        ) ENGINE = MergeTree()
        PARTITION BY toYYYYMMDD(created_at)
        ORDER BY (user_id, created_at);
```

Questa tabella archivia tutti i messaggi pubblicitari generati dal sistema, collegando utenti e attività.

- **kafka_messages**: Tabella di ingestion per i messaggi da Kafka.

```
        CREATE TABLE kafka_messages (
            message String
        ) ENGINE = Kafka(
            'kafka:9092',
            'messaggi',
            'clickhouse-consumer-group',
            'JSONEachRow'
        );
```

Questa tabella speciale con engine Kafka consente di consumare direttamente gli eventi dal topic Kafka "messaggi" senza bisogno di un consumer dedicato.

- **kafka_positions**: Tabella di ingestion per le posizioni da Kafka.

```
        CREATE TABLE kafka_positions (
            message String
        ) ENGINE = Kafka(
            'kafka:9092',
            'posizioni',
            'clickhouse-position-group',
            'JSONEachRow'
        );
```

Simile alla precedente ma dedicata al consumo di dati di posizione dal topic "posizioni".

5.3.0.4.3 Indici e partizioni

ClickHouse utilizza diversi tipi di indici per ottimizzare l'accesso ai dati:

- **Indici primari**: Basati sulle colonne di ordinamento, accelerano la ricerca per utente e timestamp.

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

- **Indici spaziali impliciti**: Funzioni geospaziali ottimizzate per interrogazioni di prossimità.

```
        WHERE geoDistance((latitude, longitude), (user_lat, user_lon)) <= 500
```

La partizione dei dati è implementata su base temporale. In questo caso le tabelle positions e messages sono partizionate per giorno:

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

Questa strategia di partizionamento:

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

5.3.0.4.4 Time-To-Live (TTL)

Il meccanismo TTL in ClickHouse rappresenta una funzionalità cruciale per NearYou, automatizzando la gestione del ciclo di vita dei dati senza richiedere procedure manuali o script esterni. ClickHouse implementa TTL come parte integrante del processo di merging di MergeTree:

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

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

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

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

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

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

Il TTL viene applicato durante le operazioni di merge:

1. Durante il merge, 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.

Nel nostro progetto usiamo solo il TTL a livello di tabella per eliminare automaticamente i dati più vecchi di 30 giorni: Ad esempio nella tabella delle posizioni:

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

Questa strategia bilancia efficacemente le esigenze di performance con l'ottimizzazione dei costi di storage, mantenendo solo i dati necessari e nella forma appropriata.

5.3.0.4.5 Materialized Views

Le materialized view rappresentano un elemento fondamentale nell'architettura di NearYou, consentendo di usare i dati provenienti da Kafka come normali tabelle ClickHouse offrendo inoltre ottime performance per le query in tempo reale. Ecco alcuni esempi:

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

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

Queste view consentono:

- Trasformazione automatica dei dati JSON in formato tabellare;
- Inserimento automatico nelle tabelle di destinazione;
- Sincronizzazione in tempo reale tra Kafka e le tabelle ClickHouse.

5.3.0.4.6 Integrazione con Kafka

L'integrazione tra Kafka e ClickHouse avviene tramite tabelle speciali con engine Kafka che fungono da consumer nativo:

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

```
CREATE TABLE positionsKafka
(
    sensor_id UUID,
    latitude Float64,
    longitude Float64,
    received_at DateTime64(3)
) ENGINE = Kafka()
SETTINGS
    kafka_broker_list = 'kafka:9092',
    kafka_topic_list = 'SimulatorPosition',
    kafka_group_name = 'clickhouseConsumerPosition',
    kafka_format = 'JSONEachRow';
```

Caratteristiche del connettore:

- Consumo asincrono dei messaggi da Kafka;
- Supporto per diverse codifiche (JSON utilizzato nel progetto);
- Gestione automatica degli offset;
- Integrazione con il sistema di materialized view per la trasformazione dati.

Questo approccio offre diversi vantaggi:

- Elimina la necessità di consumer Kafka dedicati;
- Garantisce trasformazione e inserimento dei dati in tempo reale;
- Offre resilienza: i messaggi rimangono disponibili in Kafka in caso di problemi;
- Permette di definire pipeline ETL dichiarative.

5.3.0.4.7 Funzionalità geospaziali

NearYou sfrutta intensivamente le funzioni geospaziali di ClickHouse per il calcolo delle distanze e la rilevazione di prossimità: **geoDistance**: Calcola la distanza in metri tra due punti geografici
Esempio di implementazione per la rilevazione di prossimità che è stata usata per i messaggi nella mappa generale:

```
        SELECT
            m.userLatitude AS latitude,
            m.userLongitude AS longitude,
            m.creationTime AS creazione_time,
            u.name AS userName,
            u.surname AS userSurname,
            a.name AS activityName,
            m.message AS message
        FROM (
            SELECT
            user_uuid,
            message_uuid,
            message,
            activity_uuid,
            activityLatitude,
            activityLongitude,
            creationTime,
            userLatitude,
            userLongitude,
            ROW_NUMBER() OVER (PARTITION BY user_uuid
                   ORDER BY toDateTime(creationTime) DESC) AS rn
            FROM nearyou.messageTable
        ) AS m
        INNER JOIN nearyou.user u
            ON m.user_uuid = u.user_uuid
```

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

Questa combinazione di archiviazione colonnare efficiente, indici ottimizzati, TTL automatico e strategie di query avanzate permette a ClickHouse di supportare l'intera pipeline di dati di NearYou con prestazioni eccellenti sia per l'inserimento che per l'analisi dei dati di posizione e la generazione di messaggi pubblicitari contestuali.

5.3.0.5 Grafana

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

5.3.0.5.1 Dashboard

La dashboard principale di Grafana in NearYou fornisce un'interfaccia centralizzata per monitorare l'intero sistema. È composta da due elementi fondamentali:

- **Mappa geospaziale**: Un pannello interattivo che visualizza su una mappa OpenStreetMap:
  . Marker verdi che rappresentano le posizioni attuali degli utenti;
  . Marker arancioni che mostrano i messaggi pubblicitari generati;
  . Marker rossi 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;
  . Conteggio totale dei messaggi generati.

La query che alimenta la leaderboard delle attività è la seguente:

```
    SELECT
        a.nome AS nome_attivita,
        a.tipologia,
        a.indirizzo,
        COUNT(m.id) AS numero_messaggi
    FROM nearyou.attivita a
    LEFT JOIN nearyou.messageTable m ON a.id = m.attivitaID
    GROUP BY a.id, a.nome, a.tipologia, a.indirizzo
    HAVING COUNT(m.id) > 0
    ORDER BY numero_messaggi DESC
```

5.3.0.5.2 Dashboard specifica

La dashboard specifica è dedicata all'analisi dettagliata di un singolo utente, identificato tramite un parametro `user_id`. Questa dashboard permette di:

- Visualizzare l'intero percorso storico dell'utente sulla mappa, con punti che rappresentano le posizioni registrate nel tempo;
- Evidenziare con marker specifici:
  . La prima e l'ultima posizione registrata (verde scuro e chiaro);
  . Le attività commerciali vicine al percorso (rosso);
  . I messaggi pubblicitari generati lungo il percorso (arancione).
- Analizzare i dati temporali delle posizioni e dei messaggi correlati.

5.3.0.5.3 Querying ClickHouse

L'integrazione tra Grafana e ClickHouse è realizzata tramite query SQL ottimizzate per le performance. Di seguito sono riportate le principali query utilizzate nelle dashboard:

- **Query per posizioni attuali degli utenti** (utilizzata nella mappa principale):

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

Questa query utilizza una window function (ROW_NUMBER) per selezionare solo l'ultima posizione rilevata per ogni utente.

- **Query per messaggi recenti con informazioni contestuali** (utilizzata nella mappa principale):

```
SELECT
m.userLatitude AS latitude,
m.userLongitude AS longitude,
m.creationTime AS creazione_time,
u.name AS userName,
u.surname AS userSurname,
a.name AS activityName,
m.message AS message
FROM (
SELECT
   user_uuid,
   message_uuid,
   message,
   activity_uuid,
   activityLatitude,
   activityLongitude,
   creationTime,
   userLatitude,
   userLongitude,
   ROW_NUMBER() OVER (PARTITION BY user_uuid ORDER BY toDateTime(creationTime) DESC) AS
       rn
```

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

Questa query complessa combina messaggi, dati utente e attività, utilizzando la funzione geospaziale geoDistance per filtrare i messaggi entro 300 metri dalla posizione attuale dell'utente.

- **Query per la tabella dei punti di interesse** (utilizzata nella dashboard principale):

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

Questa query genera la classifica dei punti di interesse in base al numero di messaggi pubblicitari generati.

- **Query per lo storico delle posizioni di un utente specifico** (dashboard specifica):

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

- **Query per identificare gli estremi del percorso utente** (dashboard specifica):

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

Questa query utilizza UNION ALL per combinare il primo e l'ultimo punto del percorso dell'utente, permettendo di evidenziarli con marker speciali.

- **Query per i messaggi generati per un utente specifico** (dashboard specifica):

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

- **Query per i dati anagrafici e interessi dell'utente** (dashboard specifica):

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

Questa query utilizza ARRAY_AGG per raccogliere tutti gli interessi dell'utente in un array JSON, che viene poi espanso nella dashboard utilizzando trasformazioni.

5.3.0.5.4 Connettore ClickHouse e visualizzazione in tempo reale

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

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

Il connettore nativo offre vantaggi significativi rispetto alle alternative generiche:

- **Porta con se tutte le funzionalità di ClickHouse**:
  . Supporto per query SQL 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 configuare il provisioning automatico.

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

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

Questa impostazione assicura che i dati visualizzati nelle dashboard 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.

5.3.0.5.5 Provisioning automatico

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

- **Provisioning del datasource ClickHouse**:

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

- **Provisioning delle dashboard**:

```
apiVersion: 1
providers:
  - name: "Dashboard provider"
    orgId: 1
    type: file
    disableDeletion: false
    updateIntervalSeconds: 10
    options:
```

```
            path: /var/lib/grafana/dashboards
            foldersFromFilesStructure: true
```

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

5.4 Design Pattern e Best Practices

5.4.1 Design pattern applicati

5.4.1.1 Strategy Pattern

5.4.1.1.1 Motivazioni e studio del design pattern

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

5.4.1.1.2 Implementazione del design pattern

Il pattern Strategy è implementato attraverso la definizione di un'interfaccia che delinea il comportamento comune a tutte le strategie. Ogni strategia concreta realizza questa interfaccia, fornendo un'implementazione specifica per una determinata funzionalità. Questo design consente la selezione dinamica della strategia più appropriata in base alle esigenze operative, migliorando la modularità e la manutenibilità del codice complessivo.

5.4.1.1.3 Utilizzo

L'integrazione del pattern Strategy disaccoppia la logica specifica delle funzionalità dal codice client che le invoca, semplificando l'estensibilità del sistema. 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.

5.4.1.1.4 Integrazione del pattern

Il pattern Strategy si compone di tre elementi principali: un contesto che utilizza la strategia, un'interfaccia che definisce il contratto comune e le classi concrete che implementano le varie strategie. Nel nostro sistema, questo pattern è implementato in diversi modi:

- Per la simulazione della posizione, il nostro sistema utilizza l'interfaccia IPositionSimulationStrategy, un contratto che definisce tre metodi astratti: get_route(), get_delta_time() e get_speed(). Questa interfaccia, basata sul pattern Strategy, permette di implementare diverse logiche di simulazione, garantendo flessibilità ed estendibilità per vari scenari di utilizzo.

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

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

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

La strategia concreta BycicleSimulationStrategy implementa l'interfaccia IPositionSimulationStrategy, fornendo un algoritmo specifico per simulare il movimento di una bicicletta. BycicleSimulationStrategy utilizza la libreria OSMnx per generare percorsi realistici su mappe stradali, calcolando il percorso più breve tra due nodi selezionati casualmente dal grafo e restituendo le relative coordinate geografiche. Definisce inoltre parametri specifici per la simulazione ciclistica, come una velocità media di circa 15 km/h e un delta temporale tra le posizioni simulate.
Attualmente, questa è la nostra unica implementazione, ma grazie all'adozione del pattern Strategy, il sistema è facilmente estendibile con altre strategie di simulazione.

- Per gestire l'interazione con modelli linguistici di grandi dimensioni (LLM), il nostro sistema utilizza la classe astratta LLMService, un contratto che definisce i metodi __init__(self, structured_response: BaseModel), set_up_chat(self) e get_llm_structured_response(self, prompt). Questa classe astratta, basata sul pattern Strategy, permette di integrare diversi servizi di IA.

```
class LLMService(ABC):
    def __init__(self, structured_response: BaseModel):
        self.__llm_structured_response = structured_response

    @abstractmethod
    def set_up_chat(self):
        pass

    @abstractmethod
    def get_llm_structured_response(self, prompt):
        pass
```

La strategia concreta GroqLLMService implementa la classe astratta LLMService, fornendo un meccanismo specifico per comunicare con il modello Groq. GroqLLMService si avvale della libreria langchain-groq per stabilire la connessione e inviare richieste al servizio Groq, gestendo l'autenticazione tramite una chiave API. Il metodo set_up_chat() configura l'interazione con il modello "Gemma2-9b-it" mentre il metodo get_llm_structured_response(self, prompt) prende un prompt come input e restituisce una risposta strutturata, definita al momento dell'inizializzazione della classe.
Attualmente, questa è la nostra unica implementazione, ma grazie all'adozione del pattern Strategy, il sistema è facilmente estendibile con altre strategie per diversi fornitori di LLM.

In conclusione, questo approccio modulare, basato sul pattern Strategy, ci consente di estendere facilmente il sistema con nuove strategie senza dover modificare il codice esistente. Tale progettazione rispetta i principi SOLID, in particolare il principio Open/Closed, e contribuisce significativamente al miglioramento della manutenibilità complessiva dell'applicazione.

5.4.1.2 Factory Pattern

5.4.1.2.1 Motivazioni e studio del design pattern

Il pattern Factory è stato introdotto nel nostro sistema 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 di istanziazione, si incapsula la logica potenzialmente complessa di creazione degli oggetti e si facilita l'introduzione di nuove varianti o tipologie di oggetti.

5.4.1.2.2 Implementazione del design pattern

Il pattern Factory viene implementato attraverso una classe dedicata, denominata "Factory", che contiene metodi specifici per la creazione dei diversi tipi di oggetti che essa è responsabile di produrre. Questa classe Factory spesso dipende da astrazioni (come interfacce o classi astratte) per poter creare le istanze concrete. I metodi di creazione all'interno della Factory si occupano di gestire la logica necessaria per istanziare e configurare correttamente gli oggetti richiesti, potenzialmente gestendo dipendenze o configurazioni specifiche per ciascun tipo di oggetto. La Factory può anche svolgere un ruolo nell'assicurare che gli oggetti creati rispettino determinati contratti o abbiano uno stato iniziale valido.

5.4.1.2.3 Utilizzo

L'integrazione del pattern Factory semplifica il processo 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, in quanto le dipendenze di creazione sono centralizzate e possono essere facilmente sostituite o testate isolatamente.

5.4.1.2.4 Integrazione del pattern

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

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

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

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

Il funzionamento dei metodi della SensorFactory è il seguente:

- __init__(self, sensor_repo, user_repo): Costruttore che inizializza la factory, ricevendo repository per sensori e utenti per la gestione dell'assegnazione degli ID unici tramite UserSensorService;
- create_gps_sensor(self, position_sender, simulation_strategy) -> 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(self, position_sender, simulation_strategy, number_of_sensors) -> 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.

5.4.1.3 Adapter Pattern

5.4.1.3.1 Motivazioni e studio del design pattern

Nel contesto della nostra architettura esagonale, 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 tramite serializzazione JSON oppure la comunicazione con il repository di ClickHouse). 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. Questo pattern consente quindi di adattare interfacce incompatibili e promuove il riutilizzo del codice.

5.4.1.3.2 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.

5.4.1.3.3 Integrazione del pattern

L'Adapter funge da collegamento tra l'architettura esagonale e le librerie esterne. L'architettura esagonale interagisce esclusivamente con `GeoPosition` e altre entità di dominio, senza preoccuparsi del formato dei messaggi o delle dipendenze verso Kafka. Se in futuro fosse necessario sostituire il broker di messaggistica o cambiare il formato di serializzazione, basterebbe quindi aggiornare l'Adapter corrispondente, senza intaccare la logica di business interna.
Di seguito viene mostrata l'implementazione concreta dell'Adapter, che trasforma gli oggetti `GeoPosition` in stringhe JSON compatibili con il topic di Kafka:

```
class PositionJsonAdapter(IJsonSerializable):
def serialize_to_json(self, position_istance: GeoPosition):

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

Il componente di pubblicazione `KafkaPositionPublisher`, utilizza l'Adapter per serializzare i dati prima dell'invio a Kafka:

```
class KafkaConfluentAdapter(PositionSender):

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

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

5.4.2 Best practices architetturali

5.4.2.1 PEP8 - Stile di codifica Python

Il progetto aderisce alle linee guida PEP8, lo standard di stile di codifica Python, 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 (punteggio 7.7/10.0).

5.4.2.2 Principi SOLID

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

- **Single Responsibility Principle**: Ogni classe ha una singola responsabilità ben definita. Ad esempio, la classe `KafkaMessageWriter` è responsabile esclusivamente per la pubblicazione di messaggi sul broker Kafka, mentre `MessageSerializer` si occupa solo della serializzazione dei dati;
- **Open/Closed Principle**: Le classi sono estensibili senza modificare il codice esistente. Questo è evidente nell'implementazione delle strategie di simulazione, dove nuovi comportamenti possono essere aggiunti implementando l'interfaccia `IPositionSimulationStrategy`;
- **Liskov Substitution Principle**: Le classi derivate possono sostituire le classi base senza alterare il comportamento. Le varie implementazioni di strategie di simulazione possono essere utilizzate in modo intercambiabile;
- **Interface Segregation Principle**: Le interfacce sono specifiche e mirate. Ad esempio, `IJsonSerializable` definisce solo i metodi necessari per la serializzazione JSON;
- **Dependency Inversion Principle**: I componenti dipendono da astrazioni, non da implementazioni concrete. Questo è visibile nell'iniezione delle dipendenze nei costruttori.

5.5 Diagramma delle Classi

In questa sezione viene fornita una descrizione dettagliata dei diagrammi delle classi del sistema, evidenziando le componenti principali, i loro ruoli, attributi, operazioni e le relazioni tra di esse. I diagrammi rappresentano i due componenti fondamentali dell'architettura: il modulo di simulazione delle posizioni e il processore Flink.

5.5.1 Modulo di Simulazione

Il modulo di simulazione è responsabile della generazione di dati di posizione simulati, consentendo di testare l'intero sistema senza richiedere l'utilizzo di dispositivi GPS reali.

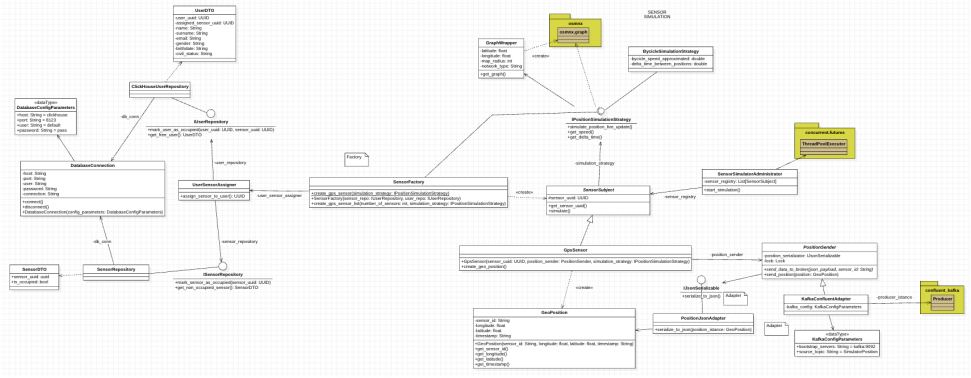

Figure 2: Diagramma delle classi del modulo di simulazione

5.5.1.1 SensorFactory

- **Descrizione**: Implementa il pattern Factory per la creazione di sensori GPS configurati. Si occupa di istanziare oggetti di tipo GpsSensor con le configurazioni appropriate, facilitando la gestione delle dipendenze.
- **Attributi**:
  - `__user_sensor_service: UserSensorService` - Servizio per l'assegnazione di sensori agli utenti, gestisce la corrispondenza tra UUID dei sensori e profili utente.
- **Operazioni**:
  - `__init__(self, sensor_repo: ISensorRepository, user_repo: IUserRepository)` - Costruttore che inizializza la factory con i repository necessari per sensori e utenti.

- `create_gps_sensor(self, position_sender: PositionSender, simulation_strategy: IPositionSimu -> SensorSubject` - Crea un singolo sensore GPS con il sender e la strategia forniti, generando un UUID unico.
- `create_gps_sensor_list(self, position_sender: PositionSender, simulation_strategy: IPositionSimulationStrategy, number_of_sensors: int) -> List[SensorSubject]` - Crea una lista di sensori GPS con la configurazione specificata, utile per simulazioni multiple.

5.5.1.2 GpsSensor

- **Descrizione**: Implementa un sensore GPS che simula il movimento utilizzando la strategia fornita. Rappresenta un dispositivo fisico che trasmette dati di posizione.
- **Attributi**:
  - `__sensor_uuid: str` - Identificatore univoco del sensore utilizzato per tracciare le posizioni.
  - `__position_sender: PositionSender` - Componente responsabile dell'invio delle posizioni al sistema di messaggistica.
  - `__simulation_strategy: IPositionSimulationStrategy` - Strategia utilizzata per simulare lo spostamento del sensore.
- **Operazioni**:
  - `__init__(self, sensor_id: str, position_sender: PositionSender, simulation_strategy: IPositionSimulationStrategy)` - Inizializza il sensore con un ID, un sender e una strategia di simulazione.
  - `simulate(self)` - Metodo principale che avvia la simulazione del movimento utilizzando la strategia definita, generando nuove posizioni a intervalli regolari.
  - `create_geo_position(self, latitude: float, longitude: float, timestamp: str) -> GeoPosition` - Crea un oggetto posizione con le coordinate specificate e il timestamp corrente.

5.5.1.3 SensorSubject

- **Descrizione**: Interfaccia che definisce il contratto per gli oggetti di tipo sensore, implementa il pattern Observer per notificare i cambiamenti di posizione.
- **Operazioni**:
  - `attach(self, observer: Observer)` - Registra un observer per ricevere notifiche dal sensore.
  - `detach(self, observer: Observer)` - Rimuove un observer registrato.
  - `notify(self)` - Notifica tutti gli observer registrati di un cambiamento di stato.
  - `simulate(self)` - Metodo astratto che deve essere implementato dalle classi concrete per avviare la simulazione.

5.5.1.4 PositionSender

- **Descrizione**: Classe astratta che definisce l'interfaccia per l'invio delle posizioni attraverso vari canali di comunicazione.
- **Attributi**:
  - `__position_serializator: IJsonSerializable` - Adattatore per la serializzazione in formato JSON.
  - `_lock: threading.Lock` - Lock per garantire thread safety durante l'invio di dati.
- **Operazioni**:
  - `__init__(self, json_adapter_istance: IJsonSerializable)` - Costruttore che inizializza il sender con l'istanza del serializzatore.

- `send_data_to_broker(self, json_payload, sensor_id: str)` - Metodo astratto da implementare nelle sottoclassi per inviare dati al broker.
- `send_position(self, position: GeoPosition)` - Metodo che serializza e invia una posizione utilizzando l'adapter fornito.

5.5.1.5 KafkaConfluentAdapter

- **Descrizione**: Implementazione concreta di PositionSender che utilizza Confluent Kafka per pubblicare i dati di posizione.
- **Attributi**:
  - `__kafka_config: KafkaConfigParameters` - Configurazione per la connessione a Kafka, include host, porta e topic.
  - `__producer: Producer` - Producer Kafka utilizzato per l'invio dei messaggi.
- **Operazioni**:
  - `__init__(self, kafka_config: KafkaConfigParameters, json_adapter_istance: PositionJsonAdapte producer_istance: Producer)` - Costruttore che inizializza l'adapter con la configurazione Kafka e il producer.
  - `send_data_to_broker(self, json_payload, sensor_id: str)` - Implementazione concreta che invia dati a Kafka utilizzando il producer configurato.

5.5.1.6 IPositionSimulationStrategy

- **Descrizione**: Interfaccia che definisce le strategie di simulazione del movimento, implementando il pattern Strategy.
- **Operazioni**:
  - `get_route(self)` - Ottiene il percorso da simulare, ritornando una sequenza di coordinate geografiche.
  - `get_delta_time(self) -> float` - Ottiene l'intervallo di tempo in secondi tra aggiornamenti consecutivi di posizione.
  - `get_speed(self) -> float` - Ottiene la velocità di simulazione in km/h.

5.5.1.7 BycicleSimulationStrategy

- **Descrizione**: Implementazione concreta della strategia che simula il movimento di una bicicletta, generando percorsi realistici.
- **Attributi**:
  - `__osmnx_graph: GraphWrapper` - Grafo stradale di OpenStreetMap utilizzato per la generazione di percorsi realistici.
- **Operazioni**:
  - `__init__(self, map_graph: GraphWrapper)` - Costruttore che inizializza la strategia con un grafo stradale.
  - `get_route(self)` - Genera un percorso realistico utilizzando il grafo stradale, selezionando punti di partenza e arrivo casuali.
  - `get_delta_time(self) -> float` - Restituisce l'intervallo temporale predefinito di 2 secondi tra aggiornamenti.
  - `get_speed(self) -> float` - Restituisce la velocità media simulata di 15 km/h tipica per una bicicletta.

5.5.1.8 SensorSimulationAdministrator

- **Descrizione**: Coordina la simulazione di più sensori in parallelo, gestendo l'esecuzione concorrente.
- **Attributi**:
  - `__sensor_registry: List[SensorSubject]` - Registro contenente tutti i sensori da simulare.
  - `__executor: ThreadPoolExecutor` - Pool di thread per l'esecuzione parallela delle simulazioni.
- **Operazioni**:
  - `__init__(self, list_of_sensors: List[SensorSubject])` - Costruttore che inizializza l'amministratore con una lista di sensori.
  - `start_simulation(self)` - Avvia la simulazione per tutti i sensori registrati utilizzando un thread pool.
  - `stop_simulation(self)` - Arresta la simulazione in corso per tutti i sensori.

5.5.1.9 GeoPosition

- **Descrizione**: Classe che rappresenta una posizione geografica con coordinate e timestamp.
- **Attributi**:
  - `__sensor_id: str` - Identificatore del sensore che ha generato la posizione.
  - `__latitude: float` - Latitudine della posizione in gradi decimali.
  - `__longitude: float` - Longitudine della posizione in gradi decimali.
  - `__timestamp: str` - Timestamp ISO 8601 che indica quando è stata rilevata la posizione.
- **Operazioni**:
  - `get_sensor_id(self) -> str` - Restituisce l'ID del sensore associato.
  - `get_latitude(self) -> float` - Restituisce la latitudine.
  - `get_longitude(self) -> float` - Restituisce la longitudine.
  - `get_timestamp(self) -> str` - Restituisce il timestamp della rilevazione.

5.5.2 Processore Flink

Il processore Flink è responsabile dell'elaborazione dei dati di posizione in tempo reale per generare messaggi pubblicitari personalizzati.

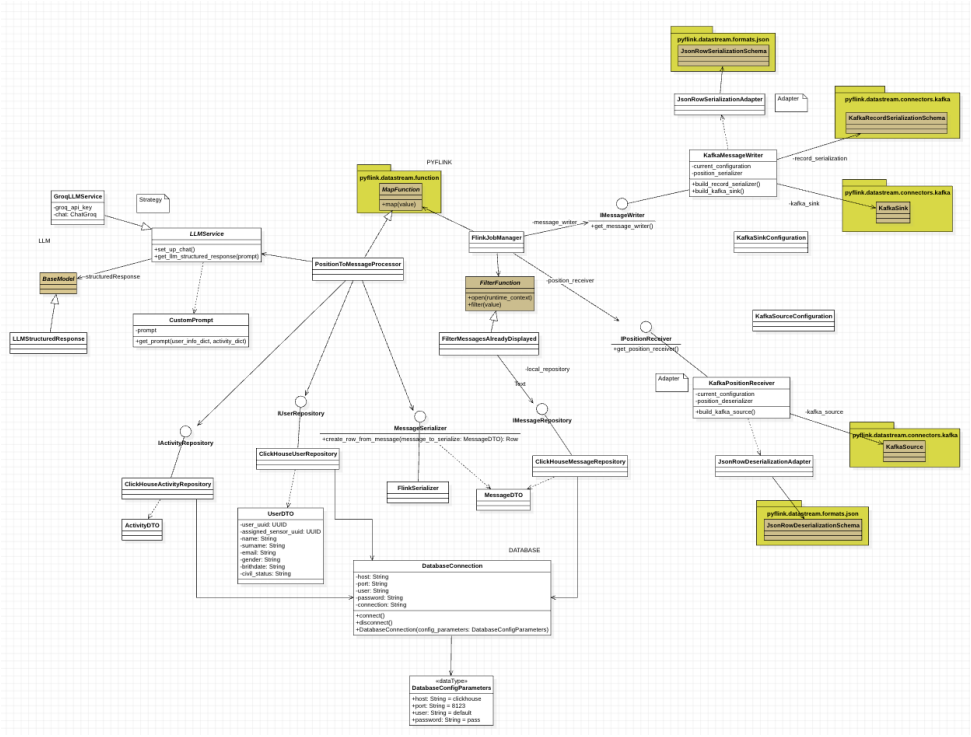

Figure 3: Diagramma delle classi del processore Flink

5.5.2.1 FlinkJobManager

- **Descrizione**: Coordina l'esecuzione del job Flink, definendo il flusso di elaborazione dei dati e orchestrando le diverse trasformazioni.
- **Attributi**:
  - `__streaming_env: StreamExecutionEnvironment` - Ambiente di esecuzione Flink per la definizione e l'esecuzione del job.
  - `__populated_datastream` - Stream di dati iniziale proveniente dalla source Kafka.
  - `keyed_stream` - Stream con chiave applicata, utilizzato per raggruppare dati relativi allo stesso sensore/utente.
  - `__mapped_stream` - Stream risultante dopo l'applicazione della map function.
  - `__filtered_stream` - Stream finale dopo l'applicazione della funzione di filtro.
- **Operazioni**:
  - `__init__(self, streaming_env_istance: StreamExecutionEnvironment, map_function_implementation: MapFunction, filter_function_implementation: FilterFunction, position_receiver_istance: IPositionReceiver, message_sender_istance: IMessageWriter)` - Costruttore che configura il job con l'ambiente e le funzioni di trasformazione.
  - `execute(self)` - Esegue il job Flink, avviando l'elaborazione e mantenendola attiva.

5.5.2.2 PositionToMessageProcessor

- **Descrizione**: Implementa la MapFunction di Flink per trasformare le posizioni in messaggi pubblicitari personalizzati.
- **Attributi**:
  - `ai_service: LLMService` - Servizio LLM per generare messaggi pubblicitari contestuali.
  - `__user_repository: IUserRepository` - Repository per accedere ai dati utente, come preferenze e informazioni demografiche.

  - `__activity_repository: IActivityRepository` - Repository per accedere ai dati delle attività commerciali nelle vicinanze.
  - `__message_serializer: IFlinkSerializable` - Serializer per i messaggi generati che saranno inviati a Kafka.

- **Operazioni**:
  - `__init__(self, ai_chatbot_service: LLMService, user_repository: IUserRepository, activity_repository: IActivityRepository, message_serializer: IFlinkSerializable)` - Costruttore che inizializza il processore con i servizi necessari.
  - `open(self, runtime_context)` - Inizializza il servizio LLM e altre risorse necessarie durante l'avvio del task.
  - `map(self, value)` - Trasforma un record di posizione in un messaggio pubblicitario, identificando POI nelle vicinanze e generando contenuti personalizzati.

5.5.2.3 FilterMessageAlreadyDisplayed

- **Descrizione**: Implementa FilterFunction per evitare duplicazioni di messaggi per lo stesso utente entro un certo periodo di tempo.
- **Attributi**:
  - `__message_repository: IMessageRepository` - Repository per verificare l'esistenza di messaggi precedenti.
- **Operazioni**:
  - `__init__(self, message_repository: IMessageRepository)` - Costruttore che inizializza il filtro con il repository dei messaggi.
  - `filter(self, value)` - Filtra i messaggi che sono già stati visualizzati dall'utente per la stessa attività entro un periodo configurabile.

5.5.2.4 KafkaPositionReceiver

- **Descrizione**: Adattatore per ricevere posizioni da Kafka, implementa IPositionReceiver.
- **Attributi**:
  - `__current_configuration: KafkaSourceConfiguration` - Configurazione della source Kafka, inclusi topic, gruppo di consumer e server.
  - `__position_deserializer` - Deserializzatore per convertire i dati JSON in oggetti utilizzabili.
  - `__kafka_source: KafkaSource` - Source Kafka configurata per la lettura dei dati.
- **Operazioni**:
  - `__init__(self, kafka_source_configuration: KafkaSourceConfiguration, deserialize_adapter: JsonRowDeserializationAdapter)` - Costruttore che inizializza il receiver con la configurazione e il deserializzatore.
  - `build_kafka_source(self) -> KafkaSource` - Costruisce e configura la source Kafka con le impostazioni appropriate.
  - `get_position_receiver(self)` - Restituisce la source Kafka configurata per l'utilizzo nel job Flink.

5.5.2.5 KafkaMessageWriter

- **Descrizione**: Adattatore per scrivere messaggi su Kafka, implementa IMessageWriter.
- **Attributi**:
  - `__current_configuration: KafkaWriterConfiguration` - Configurazione del writer Kafka, inclusi topic e server.
  - `__position_serializer` - Serializzatore per convertire gli oggetti in formato JSON.
  - `__record_serializer` - Serializzatore per i record Kafka, gestisce la trasformazione e la definizione di schema.
  - `__kafka_sink: KafkaSink` - Sink Kafka configurata per la scrittura dei dati.
- **Operazioni**:
  - `__init__(self, kafka_writer_configuration: KafkaWriterConfiguration, serialize_adapter: JsonRowSerializationAdapter)` - Costruttore che inizializza il writer con la configurazione e il serializzatore.
  - `build_record_serializer(self)` - Costruisce il serializzatore di record per Kafka.
  - `build_kafka_sink(self)` - Costruisce la sink Kafka con le impostazioni appropriate.
  - `get_message_writer(self)` - Restituisce la sink Kafka configurata per l'utilizzo nel job Flink.

5.5.2.6 LLMService

- **Descrizione**: Classe astratta che definisce l'interfaccia per i servizi di language model utilizzati nella generazione di messaggi personalizzati.
- **Attributi**:
  - `_llm_structured_response: BaseModel` - Modello Pydantic per la risposta strutturata proveniente dal LLM.
- **Operazioni**:
  - `__init__(self, structured_response: BaseModel)` - Costruttore che inizializza il servizio con il modello di risposta strutturata.
  - `set_up_chat(self)` - Metodo astratto per inizializzare la connessione con il servizio LLM.
  - `get_llm_structured_response(self, prompt)` - Metodo astratto per ottenere una risposta strutturata dall'LLM.

5.5.2.7 GroqLLMService

- **Descrizione**: Implementazione concreta di LLMService che utilizza il provider Groq per la generazione di testi.
- **Attributi**:
  - `__groq_api_key: str` - Chiave API per l'autenticazione con il servizio Groq.
  - `__chat` - Istanza del client Groq per l'interazione con il servizio.
- **Operazioni**:
  - `__init__(self, structured_response: BaseModel)` - Costruttore che inizializza il servizio Groq.
  - `set_up_chat(self)` - Configura il client Groq con rate limiting e gestione degli errori.
  - `get_llm_structured_response(self, prompt)` - Invia un prompt al servizio Groq e converte la risposta in un formato strutturato.

5.5.2.8 MessageDTO

- **Descrizione**: Data Transfer Object per i messaggi pubblicitari generati.
- **Attributi**:
  - `user_uuid: str` - Identificatore univoco dell'utente destinatario.
  - `activity_uuid: str` - Identificatore univoco dell'attività commerciale associata.
  - `message_uuid: str` - Identificatore univoco del messaggio.
  - `message: str` - Contenuto testuale del messaggio pubblicitario.
  - `activityLatitude: float` - Latitudine dell'attività commerciale.
  - `activityLongitude: float` - Longitudine dell'attività commerciale.
  - `creationTime: str` - Timestamp di creazione del messaggio.
  - `userLatitude: float` - Latitudine dell'utente al momento della generazione.
  - `userLongitude: float` - Longitudine dell'utente al momento della generazione.

5.5.2.9 ActivityDTO

- **Descrizione**: Data Transfer Object per le attività commerciali.
- **Attributi**:
  - `activity_id: str` - Identificatore univoco dell'attività.
  - `name: str` - Nome dell'attività commerciale.
  - `category: str` - Categoria o tipologia dell'attività.
  - `description: str` - Descrizione dettagliata dell'attività.
  - `latitude: float` - Latitudine dell'attività.
  - `longitude: float` - Longitudine dell'attività.

5.5.2.10 UserDTO

- **Descrizione**: Data Transfer Object per gli utenti del sistema.
- **Attributi**:
  - `user_id: str` - Identificatore univoco dell'utente.
  - `name: str` - Nome dell'utente.
  - `age: int` - Età dell'utente.
  - `gender: str` - Genere dell'utente.
  - `interests: List[str]` - Lista degli interessi dell'utente.
  - `preferences: List[str]` - Lista delle preferenze dell'utente.

5.5.3 Relazioni tra componenti

Le relazioni tra i componenti riflettono l'architettura modulare del sistema, basata su interfacce che seguono i principi SOLID:

5.5.3.1 Modulo di Simulazione

- `SensorFactory` crea istanze di `GpsSensor` che implementano l'interfaccia `SensorSubject`.
- `GpsSensor` utilizza `IPositionSimulationStrategy` (implementata da `BycicleSimulationStrategy`) per determinare il proprio movimento.
- `GpsSensor` invia dati attraverso `PositionSender` (implementato da `KafkaConfluentAdapter`) che utilizza `IJsonSerializable` per la serializzazione.
- `SensorSimulationAdministrator` coordina una collezione di sensori che implementano `SensorSubject`.

5.5.3.2 Processore Flink

- `FlinkJobManager` orchestra l'intero flusso di elaborazione, ricevendo dati tramite `KafkaPositionReceiver` (implementa `IPositionReceiver`).
- `FlinkJobManager` applica `PositionToMessageProcessor` per trasformare i dati (implementa `MapFunction`) e `FilterMessageAlreadyDisplayed` per filtrare messaggi duplicati (implementa `FilterFunction`).
- `FlinkJobManager` invia i risultati tramite `KafkaMessageWriter` (implementa `IMessageWriter`).
- `PositionToMessageProcessor` interagisce con `LLMService` (implementato da `GroqLLMService`) per generare messaggi, e con repository di dati per accedere a informazioni su utenti e attività.

Questa architettura modulare consente un alto grado di testabilità e manutenibilità, permettendo la sostituzione di componenti con implementazioni alternative senza modificare il flusso di elaborazione complessivo.

6 Architettura di deployment

Da capitolato 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 seguendo questa architettura, è stato configurato un ambiente Docker 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 del progetto oltre che nella sezione 2.3.

6.1 Panoramica dell'infrastruttura

6.1.1 Ambiente Docker

Descrizione dell'ambiente Docker utilizzato, incluse le immagini e le configurazioni specifiche.

6.1.2 Componenti principali

In questa sezione vengono descritti i componenti principali dell'architettura, evidenziandone i ruoli e le responsabilità.

6.1.3 Dipendenze tra componenti

Le interazioni tra i vari componenti avvengono attraverso Kafka, 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 definito e li invia al broker Kafka.
- **Gestione messaggi:**
  - **Container kafka:**
    . Esegue Apache Kafka per la gestione del flusso di dati in tempo reale;
    . Gestisce i topic dedicati per i diversi tipi di messaggi (posizioni, POI, messaggi pubblicitari);
    . Accessibile agli altri container tramite l'indirizzo kafka:9092.
  - **Componenti di supporto:**
    . **Container zookeeper:**
      - Esegue il servizio di coordinamento per Kafka;
      - Gestisce lo stato distribuito del sistema;

- Accessibile dagli altri container attraverso l'indirizzo zookeeper:2181.

. **Container kafka-ui:**
- Fornisce un'interfaccia web per il monitoraggio e la gestione di Kafka;
- Espone la porta 8080 per accedere alla dashboard 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 per la generazione di messaggi pubblicitari personalizzati.
- **Storage:**
  - **Container clickhouse:**
    . Esegue ClickHouse come database 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 tramite l'indirizzo clickhouse:8123 e 9000.
- **Visualizzazione:**
  - **Container grafana:**
    . Esegue Grafana come piattaforma di visualizzazione e monitoraggio;
    . Offre dashboard interattive per l'analisi dei dati di posizione e messaggi pubblicitari;
    . Espone la porta 3000 all'esterno per permettere l'accesso alle dashboard;
    . Consente l'integrazione con vari datasource per la visualizzazione dei dati.

6.2 Vantaggi dell'architettura containerizzata

Questa struttura containerizzata permette una distribuzione modulare e scalabile del sistema, semplificando la gestione e la manutenzione dei componenti e consentendo una rapida scalabilità in risposta alle esigenze emergenti. Grazie all'uso di Docker, 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;
- **Portabilità della configurazione**: Grazie al compose di Docker si possono definire delle cartelle o dei file di configurazione per ogni container, rendendo il sistema facilmente replicabile;
- **Scalabilità orizzontale**: I container 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 riceve solo le risorse necessarie per il suo funzionamento;

6.3 Comunicazione tra container

La comunicazione tra i vari container avviene principalmente attraverso il networking interno di Docker, con Kafka che agisce come backbone di messaggistica centrale del sistema. 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 e copiati subito dopo su una tabella clickhouse, garantendo la storicizzazione del dato;

6.4 Orchestrazione e gestione

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

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

6.5 Evoluzione futura

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

- **Migrazione verso Kubernetes**: L'attuale configurazione Docker è pronta per essere eventualmente trasferita su un orchestratore come Kubernetes per una gestione più avanzata dei container;
- **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 in tempo reale.

7 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_v1.0.0*.

7.1 Riepilogo dei requisiti

Durante la fase di analisi sono stati individuati 33 requisiti funzionali (RF01-RF33), di cui:

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

I requisiti funzionali riguardano principalmente:

- La visualizzazione della Dashboard e dei Marker sulla mappa;
- La gestione e visualizzazione dei punti di interesse;
- La visualizzazione degli annunci pubblicitari generati;
- L'interazione con il servizio LLM per la generazione di contenuti personalizzati;
- La trasmissione e gestione dei dati geoposizionali.

7.2 Tabella dei requisiti funzionali

Id. Requisito	Importanza	Descrizione	Stato
RF01	Obbligatorio	L'utente privilegiato deve poter visualizzare la Dashboard composta da una mappa interattiva con i vari Marker su di essa.	Implementato
RF02	Obbligatorio	L'utente privilegiato deve poter visualizzare dei Marker che rappresentano i vari Percorsi effettuati in tempo reale dagli utenti presenti nel Sistema	Implementato
RF03	Obbligatorio	L'utente privilegiato deve poter visualizzare un Marker che rappresenta un Percorso effettuato in tempo reale da un utente presente nel Sistema	Implementato
RF04	Obbligatorio	L'utente privilegiato deve poter visualizzare tutti i punti di interesse riconosciuti dal Sistema.	Implementato
RF05	Obbligatorio	L'utente privilegiato deve poter visualizzare un Marker che rappresenta un punto di interesse riconosciuto dal Sistema.	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.	Implementato
RF08	Obbligatorio	L'utente privilegiato deve poter visualizzare una Dashboard relativa ad un singolo utente quando seleziona un Marker utente nella Dashboard principale.	Implementato
RF09	Obbligatorio	L'utente privilegiato deve poter visualizzare dei Marker che rappresentano lo storico delle posizioni dell'utente a cui è riferita la Dashboard di singolo utente.	Implementato
RF10	Obbligatorio	L'utente privilegiato deve poter visualizzare un Marker che rappresenta la posizione dell'utente in un determinato istante nella Dashboard di singolo utente.	Implementato
RF11	Obbligatorio	L'utente privilegiato deve poter visualizzare, nella Dashboard di singolo utente, tutti i punti di interesse riconosciuti dal Sistema.	Implementato
RF12	Obbligatorio	L'utente privilegiato deve poter visualizzare, nella Dashboard di singolo utente, un Marker che rappresenta un punto di interesse riconosciuto dal Sistema.	Implementato
RF13	Obbligatorio	L'utente privilegiato deve poter visualizzare lo storico degli annunci pubblicitari generati per l'utente a cui è riferita la Dashboard singolo utente.	Implementato
RF14	Obbligatorio	L'utente privilegiato deve poter visualizzare un singolo annuncio pubblicitario tramite un Marker nella Dashboard di singolo utente.	Implementato
RF15	Obbligatorio	L'utente privilegiato deve poter visualizzare un pannello apposito contenente le informazioni dell'utente, a cui è riferita la Dashboard di singolo utente, in forma tabellare.	Implementato
RF16	Obbligatorio	L'utente privilegiato deve poter visualizzare nel pannello 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 riguardante una singola posizione di un utente nella rispettiva Dashboard	Implementato

RF18	Obbligatorio	L'utente privilegiato quando visualizza i dettagli del Marker, riguardante una singola posizione di un utente nella rispettiva Dashboard, deve poter vedere la latitudine, la longitudine e l'istante di rilevamento del Marker	Implementato
RF19	Opzionale	L'utente privilegiato deve poter visualizzare l'area di influenza di un punto di interesse selezionato.	Implementato
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 ordinati per la quantità di messaggi inviati nel mese.	Implementato
RF24	Obbligatorio	L'utente privilegiato deve poter visualizzare nella tabella dei PoI un singolo PoI, rappresentato da una riga della tabella.	Implementato
RF25	Obbligatorio	L'utente privilegiato deve poter visualizzare in ogni riga della tabella dei PoI 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.	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.	Implementato
RF29	Obbligatorio	Il sensore deve essere in grado di trasmettere il proprio id, la sua latitudine e longitudine al Sistema.	Implementato
RF30	Obbligatorio	Il servizio LLM deve essere in grado di ricevere i dati dell'utente e del punto di interesse inviati dal Sistema.	In implementazione
RF31	Obbligatorio	Il servizio LLM deve essere in grado di ricevere le informazioni dell'utente inviate dal Sistema, quali: il nome, il cognome, l'email, il genere, la data di nascita, lo stato civile e i suoi interessi.	In implementazione
RF32	Obbligatorio	Il servizio LLM deve essere in grado di ricevere le informazioni del punto di interesse inviate dal Sistema, quali: il nome, l'indirizzo, la tipologia, la descrizione e la distanza del PoI dall'utente.	In implementazione
RF33	Obbligatorio	Il servizio LLM deve essere in grado di trasmettere un messaggio custom, rappresentante il contenuto dell'annuncio per un utente, al Sistema.	Da implementare

7.3 Stato di implementazione

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

Tipo di requisito	**Totale**	**Implementati**	**In implementazione**	**Da implementare**
Obbligatori	31	26	3	2
Desiderabili	0	0	0	0
Opzionali	2	1	0	1
Totale	33	27	3	3

Table 4: Stato di implementazione dei requisiti funzionali

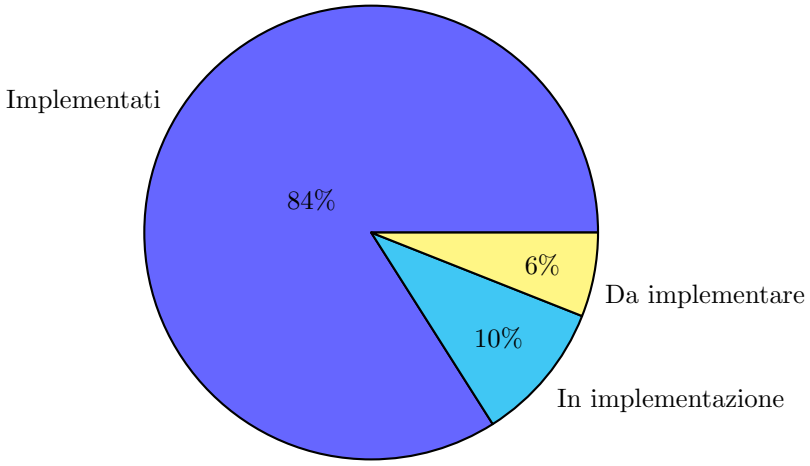

Figure 4: Stato dei requisiti funzionali obbligatori

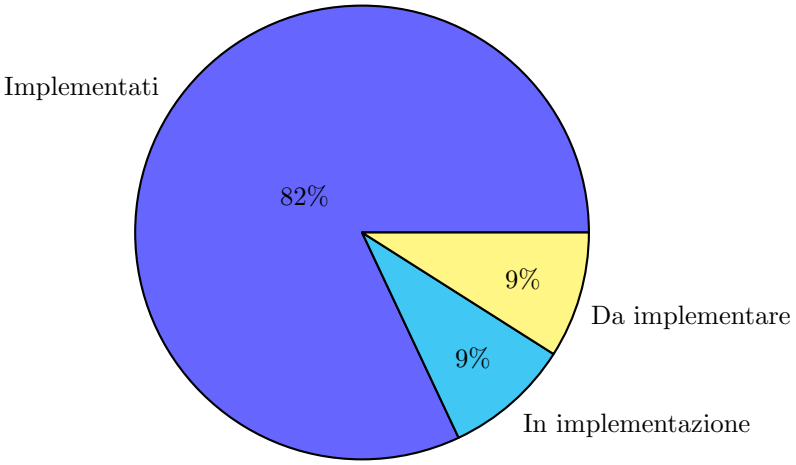

Figure 5: Stato dei requisiti funzionali totali

7.4 Conclusioni

L'analisi dello stato dei requisiti funzionali indica un buon progresso nell'implementazione delle funzionalità richieste. Con l'82% dei requisiti funzionali già implementati e il 9% in fase di implementazione, il progetto è sulla buona strada per soddisfare tutti i requisiti obbligatori entro la consegna finale.
I requisiti attualmente in fase di implementazione riguardano principalmente l'integrazione con il servizio LLM, mentre quelli ancora da implementare comprendono uno dei requisiti opzionali e alcuni aspetti avanzati del sistema. La stabilità raggiunta nei requisiti dopo la revisione tecnica fornisce una base solida per il completamento dell'implementazione, mentre il sistema di test definito garantirà la qualità del prodotto finale.