Specifica Tecnica

2025-08-08

Responsabile	Alessandro
Redattori	Alessandro Di Pasquale
	Nicolò Bovo
	Elia Leonetti
Verificatori	Massimo Chioru
	Romeo Calearo
	Manuel Cinnirella
	Giovanni Battista Matteazzi

AlphaCode
Università Degli Studi di Padova
Versione 0.2.0

Registro delle modifiche

Vers.	Data	Descrizione	Autore	Verificatore
0.2.0	2025-08-13	Aggiunta classi, pattern e test	Nicolò Bovo	Massimo Chioru
0.1.0	2025-04-13	Stesura iniziale del glossario	Alessandro Di Pasquale	Manuel Cinnirella

Indice

I. Introduzione 8
I - 1. Scopo del documento 8
I - 2. Scopo del progetto 8
I - 3. Riferimenti 8
I - 3.1. Riferimenti informativi 8
I - 3.2. Riferimenti normativi 8
II. Tecnologie 9
II - 1. Infrastuttura del sistema 9
II - 1.1. Docker 9
II - 1.2. Docker Compose per l'Orchestrazione 9
II - 2. Linguaggi di sviluppo 10
II - 2.1. Python 10
II - 2.1.1. Utilizzo nel progetto: 11
II - 2.1.2. Versione: 11
II - 2.1.3. Librerie e framework: 11
II - 2.1.4. Test 12
II - 2.2. JavaScript 12
II - 2.2.1. Utilizzo nel progetto 13
II - 2.2.2. Versione 13
II - 2.2.3. Librerie e framework 13
II - 2.2.4. Test 14
II - 2.3. Data Broker 14
II - 2.3.1. Apache Kafka 14
II - 2.3.2. Configurazione protezione comunicazioni 14
Schema dei messaggi di posizionamento 15
II - 2.4. Stream Processor 16
II - 2.4.1. Bytewax 16
II - 2.4.2. Creazione comunicazioni personalizzate 16
II - 2.4.3. Architettura LLM 16
II - 2.4.4. LangChain Python Integration 17
II - 2.5. Sistema di Persistenza (Database) 17
II - 2.5.1. PostgreSQL con PostGIS 17
II - 2.5.2. Estensione PostGIS 17
II - 2.5.3. ClickHouse 18
II - 2.5.4. Architettura Database: schema PostgreSQL (Negozi e Offerte) 18
II - 2.5.5. Architettura Database: Schema ClickHouse (Eventi e Analytics) 19
II - 2.5.6. Diagramma Relazionale 21
II - 2.5.7. Decisioni Architetturali 21
II - 2.6. Interfaccia utente 22
II - 2.6.1. Dashboard Utente Real-Time 22

II - 2.7. Sistema di Monitoraggio Admin - Grafana 23

III. Architettura del sistema 23

III - 1. Lambda Architecture 24

III - 1.1. Strati Principali 25

III - 1.1.1. Batch Layer 25

III - 1.1.2. Speed Layer 25

III - 1.1.3. Serving Layer 25

III - 1.2. Componenti Tecnologici 26

III - 1.3. Flusso dei Dati 26

III - 1.4. Consistenza e Fusione 27

III - 1.5. Riepilogo 27

III - 2. Diagramma delle classi 27

IV. Documentazione delle Classi - Sistema NearYou 27

IV - 1. Gestione Configurazione 27

IV - 1.1. Struttura delle classi 28

IV - 1.1.1. ConfigurationManager 28

Attributi 28

Costruttori 28

Metodi 28

IV - 2. Sistema di Cache 29

IV - 2.1. Struttura delle classi 29

IV - 2.1.1. MemoryCache 29

Attributi 29

Costruttori 29

Metodi 29

IV - 2.1.2. RedisCache 30

Attributi 30

Costruttori 30

Metodi 30

IV - 3. Modelli Dati 30

IV - 3.1. Struttura delle classi 30

IV - 3.1.1. OfferType 30

Attributi 31

Costruttori 31

Metodi 31

IV - 3.1.2. OfferValidatorProtocol 31

Attributi 31

Costruttori 31

Metodi 31

IV - 3.1.3. OfferValidator 31

Attributi 31

Costruttori 31

Metodi 31

IV - 3.1.4. Offer 32

Attributi 32
Costruttori 33
Metodi 33
IV - 3.1.5. OfferBuilder 33
Attributi 33
Costruttori 33
Metodi 33
IV - 3.1.6. OfferFactory 34
Attributi 34
Costruttori 34
Metodi 34
IV - 3.1.7. UserVisit 35
Attributi 35
Costruttori 36
Metodi 36
IV - 4. Servizi Business Logic 36
IV - 4.1. Struttura delle classi 36
IV - 4.1.1. OfferGenerationStrategy 36
Attributi 36
Costruttori 36
Metodi 36
IV - 4.1.2. StandardOfferStrategy 37
Attributi 37
Costruttori 37
Metodi 37
IV - 4.1.3. AggressiveOfferStrategy 37
Attributi 37
Costruttori 37
Metodi 37
IV - 4.1.4. ConservativeOfferStrategy 38
Attributi 38
Costruttori 38
Metodi 38
IV - 4.1.5. OfferStrategyFactory 38
Attributi 38
Costruttori 39
Metodi 39
IV - 4.1.6. OffersService 39
Attributi 39
Costruttori 39
Metodi 39
IV - 5. Data Pipeline 39
IV - 5.1. Struttura delle classi 40
IV - 5.1.1. Observer 40
Attributi 40

Costruttori 40
Metodi 40
IV - 5.1.2. Subject 40
Attributi 40
Costruttori 40
Metodi 40
IV - 5.1.3. MetricsObserver 41
Attributi 41
Costruttori 41
Metodi 41
IV - 5.1.4. PerformanceObserver 41
Attributi 41
Costruttori 41
Metodi 42
IV - 5.1.5. DatabaseConnections 42
Attributi 42
Costruttori 43
Metodi 43
IV - 6. Utilità 43
IV - 6.1. Struttura delle classi 43
IV - 6.1.1. Utilità Database 43
IV - 6.1.2. Configurazione Logging 43
IV - 6.1.3. Metriche FastAPI 43
IV - 6.1.4. Operatori Pipeline 43
IV - 7. Design Patterns 44
IV - 7.1. Introduzione 44
IV - 7.2. Singleton Pattern 44
IV - 7.2.1. Panoramica 44
IV - 7.2.2. ConfigurationManager 44
Implementazione 44
Caratteristiche Avanzate 45
Vantaggi Architetturali 45
IV - 7.2.3. CacheManager 45
Implementazione 45
Integrazione con Factory Pattern 46
IV - 7.2.4. DatabaseConnections 46
Implementazione con Observer Pattern 46
IV - 7.3. Factory Pattern 47
IV - 7.3.1. Panoramica 47
IV - 7.3.2. CacheFactory 47
Implementazione Completa 47
Caratteristiche Avanzate 47
IV - 7.3.3. OfferStrategyFactory 48
Implementazione 48
Estensibilità 48

IV - 7.4. Strategy Pattern 48
IV - 7.4.1. Panoramica 48
IV - 7.4.2. Architettura delle Strategie 49
Interfaccia Base 49
IV - 7.4.3. Implementazioni Concrete 49
StandardOfferStrategy 49
AggressiveOfferStrategy 50
IV - 7.4.4. Utilizzo Runtime 50
Switching Dinamico 50
IV - 7.5. Observer Pattern 51
IV - 7.5.1. Panoramica 51
IV - 7.5.2. Architettura Observer 51
Subject Base 51
IV - 7.5.3. Implementazioni Observer 51
MetricsObserver 51
PerformanceObserver 52
IV - 7.5.4. DatabaseConnections come Subject 53
Implementazione Integrata 53
IV - 8. Tabella dei Requisiti - Stato di Implementazione 54

I. Introduzione

I - 1. Scopo del documento

Questo documento di Specifiche Tecniche ha l'obiettivo di illustrare in modo approfondito le decisioni tecnologiche e le soluzioni tecniche adottate dal team di sviluppo per la realizzazione del progetto **NearYou - Smart Custom Advertising Platform**, sviluppato nell'ambito del capitolato 4 proposto da **SyncLab S.r.l.**.

All'interno del documento vengono fornite descrizioni dettagliate delle tecnologie impiegate, delle decisioni architetturali e delle scelte implementative, insieme ai pattern$_G$ di progettazione utilizzati per costruire l'ecosistema software che costituisce la piattaforma **NearYou**.

Il documento include anche la mappatura dei requisiti funzionali che sono stati soddisfatti durante il processo di sviluppo del prodotto, accompagnata da rappresentazioni grafiche che dimostrano il livello di copertura raggiunto per ciascuno di essi.

I - 2. Scopo del progetto

La piattaforma **NearYou Smart Custom Advertising Platform** rappresenta una soluzione innovativa che impiega l'Intelligenza Artificiale Generativa$_G$ per sviluppare contenuti pubblicitari altamente personalizzati e mirati per ogni singolo utente. Il sistema utilizza diverse tipologie di informazioni, tra cui i dati di geolocalizzazione$_G$ trasmessi in tempo reale$_G$, le caratteristiche demografiche degli utenti e i profili comportamentali acquisiti, con l'obiettivo di ottimizzare l'esperienza dell'utente finale e massimizzare simultaneamente il ritorno sull'investimento e l'efficacia delle strategie di marketing digitale.

Le principali caratteristiche della piattaforma includono:

- **Tracking geospaziale in tempo reale** attraverso elaborazione di stream$_G$ di dati GPS$_G$
- **Generazione automatica di messaggi personalizzati** mediante LLM$_G$
- **Sistema di notifiche di prossimità** basato sul calcolo della distanza
- **Dashboard$_G$ interattiva** con visualizzazione cartografica per monitoraggio utenti
- **Analytics comportamentali avanzate** per analisi

I - 3. Riferimenti

I - 3.1. Riferimenti informativi

- Glossario (v2.0.0)
- Capitolato C4 fornito dall'azienda

I - 3.2. Riferimenti normativi

- Norme di Progetto (v2.0.0)
- Capitolato C4 fornito dall'azienda

II. Tecnologie

II - 1. Infrastuttura del sistema

II - 1.1. Docker

Docker$_G$ costituisce la tecnologia di containerizzazione$_G$ adottata per l'orchestrazione dell'intera infrastruttura applicativa di NearYou. La piattaforma viene utilizzata per incapsulare ogni microservizio$_G$ in container$_G$ isolati, garantendo consistenza ambientale tra sviluppo, testing e produzione. Nel progetto NearYou, Docker facilita:

- **Isolamento dei servizi**: Ogni componente (Kafka$_G$, ClickHouse$_G$, PostgreSQL$_G$, Redis$_G$) opera in container dedicati
- **Gestione delle dipendenze**: Eliminazione dei conflitti tra librerie e versioni diverse
- **Deployment$_G$ semplificato**: Avvio dell'intera stack con un singolo comando *docker-compose up*
- **Scalabilità orizzontale$_G$:** Possibilità di replicare servizi aumentando il numero di container
- **Ambiente riproducibile:** Configurazione identica su qualsiasi macchina di sviluppo

Architettura$_G$ Container: Il sistema utilizza un approccio multi-stage con Dockerfile principale (deployment/docker/Dockerfile) per la base Python, Dockerfile OSRM specializzato per il routing, configurazioni dedicate tramite file .env, healthcheck integrati e restart policies automatiche.

II - 1.2. Docker Compose per l'Orchestrazione

La replicabilità del sistema viene resa possibile grazie a **Docker Compose**$_G$, che permette di definire e gestire applicazioni multi-container attraverso file di configurazione YAML$_G$. NearYou adotta un approccio modulare con un file principale *docker-compose.yml* nella root che coordina l'intera infrastruttura tramite la direttiva *include*:

- ./deployment/docker/docker-compose.yml - servizi core dell'applicazione
- ./monitoring/docker-compose.monitoring.yml - servizi di monitoraggio

Questa architettura modulare facilita la manutenzione e permette l'attivazione selettiva di sottosistemi specifici. L'avvio dell'intera stack avviene con il singolo comando `docker-compose up -d` che coordina l'inizializzazione dei servizi presenti.

Servizi Core del Sistema:

1. **osrm-milano**: Servizio di routing ottimizzato per l'area di Milano
   - Immagine: ghcr.io/project-osrm/osrm-backend:v5.27.1
2. **kafka**: Message broker$_G$ per lo streaming$_G$ di dati GPS in tempo reale
   - Immagine: bitnami/kafka:3.4
3. **zookeeper**: Coordinatore per Kafka
   - Immagine: bitnami/zookeeper:latest
4. **clickhouse**: Database$_G$ analitico per l'archiviazione degli utenti e dei relativi eventi
   - Immagine: clickhouse/clickhouse-server:latest

5. **postgres-postgis**: Database relazionale con estensioni geospaziali per negozi e offerte
   - Immagine: postgis/postgis:15-3.3
6. **message-generator**: Microservizio per la generazione di messaggi personalizzati via LLM
   - Immagine: Build custom basata su python:3.10-slim
7. **dashboard-user**: Interfaccia web per utenti finali
   - Immagine: Build custom basata su python:3.10-slim
8. **producer/consumer**: Pipeline$_G$ di elaborazione dati tramite Bytewax
   - Immagine: Build custom basata su python:3.10-slim
9. **airflow-webserver/scheduler/worker**: Orchestratore per processi ETL$_G$ di negozi e offerte
   - Immagine: apache/airflow:2.5.0
10. **redis-cache**: Sistema di caching$_G$ per ottimizzazione performance
    - Immagine: redis:alpine
11. **grafana**: Dashboard di visualizzazione e analytics
    - Immagine: grafana/grafana:latest

Servizi di Monitoraggio:

1. **prometheus**: Raccolta metriche applicative
   - Immagine: prom/prometheus:v2.45.0
2. **loki**: Aggregazione log centralizzata
   - Immagine: grafana/loki:2.9.1
3. **promtail**: Agente per raccolta log
   - Immagine: grafana/promtail:2.9.1
4. **node-exporter** : Monitoraggio risorse sistema
   - Immagine: prom/node-exporter:v1.6.1
5. **redis-exporter**: Esportatore metriche Redis
   - Immagine: oliver006/redis_exporter

L'utilizzo del **Makefile** semplifica le operazioni di sviluppo con comandi come make build e make run_dev, rendendo il sistema accessibile anche a sviluppatori meno esperti con Docker.

II - 2. Linguaggi di sviluppo

II - 2.1. Python

Python$_G$ è un linguaggio di programmazione interpretato, multiparadigma e ad alto livello che supporta sia la programmazione orientata agli oggetti che quella procedurale. La sua sintassi chiara e la vasta libreria standard lo rendono ideale per applicazioni di data processing, machine learning e sviluppo web.

II - 2.1.1. Utilizzo nel progetto:

Python costituisce il linguaggio principale del progetto NearYou, utilizzato per:

- **Data Pipeline**: Producer per la generazione di dati GPS simulati e consumer per l'elaborazione in tempo reale tramite Bytewax
- **ETL Processes**: Script Airflow$_G$ per l'estrazione e caricamento dati negozi da Overpass API$_G$
- **Simulazione utenti**: Generazione di profili utente realistici
- **Integrazione LLM**: Interfacciamento con modelli linguistici per la creazione di messaggi personalizzati

II - 2.1.2. Versione:

La versione di Python utilizzata per lo sviluppo è la **3.10**, come specificato nel Dockerfile base (python:3.10-slim).

II - 2.1.3. Librerie e framework:

Per la gestione delle dipendenze è stato utilizzato **pip** con file requirements.txt modulari. Per una visione dettagliata di tutte le librerie utilizzate, è possibile consultare i file presenti nella cartella requirements/ del progetto.

La seguente lista rappresenta le dipendenze più rilevanti:

Stream Processing:

1. **Bytewax**$_G$
   - Documentazione: https://docs.bytewax.io
   - Versione: 0.19.0
   - Descrizione: Framework$_G$ per stream processing in tempo reale dei dati GPS, permettendo operazioni stateful su flussi di dati in maniera reattiva e scalabile

Database e Storage :

1. **ClickHouse-driver**
   - Documentazione: https://clickhouse-driver.readthedocs.io
   - Versione: 0.2.5
   - Descrizione: Driver per la connessione al database analitico ClickHouse per l'archiviazione di eventi utente
2. **Psycopg2-binary**
   - Documentazione: https://www.psycopg.org/docs
   - Versione: 2.9.6
   - Descrizione: Adapter PostgreSQL per operazioni geospaziali con PostGIS$_G$, utilizzato per gestire negozi e offerte

Web Framework e API:

1. **FastAPI**$_G$
   - Documentazione: https://fastapi.tiangolo.com
   - Versione: 0.103.1
   - Descrizione: Framework moderno per API$_G$ REST$_G$
2. **Uvicorn**

- Documentazione: https://www.uvicorn.org
- Versione: 0.23.2
- Descrizione: Server ASGI$_G$ per applicazioni asincrone Python, utilizzato per servire le API FastAPI

Machine Learning e LLM :

1. **LangChain**$_G$
   - Documentazione: https://docs.langchain.com
   - Versione: 0.0.286
   - Descrizione: Framework per applicazioni basate su modelli linguistici, utilizzato per la generazione di messaggi personalizzati tramite LLM
2. **OpenAI**
   - Documentazione: https://platform.openai.com/docs
   - Versione: 0.28.1
   - Descrizione: SDK$_G$ per integrazione con modelli linguistici GPT e provider compatibili

Utilities:

1. **Faker**
   - Documentazione: https://faker.readthedocs.io
   - Versione: 18.13.0
   - Descrizione: Generazione di dati di test realistici per profili utente e simulazioni
2. **Pydantic**$_G$
   - Documentazione: https://docs.pydantic.dev
   - Versione: 2.4.2
   - Descrizione: Validazione dati e serializzazione modelli per garantire type safety nelle API
3. **Python-dotenv**
   - Documentazione: https://pypi.org/project/python-dotenv
   - Versione: 1.0.0
   - Descrizione: Modulo per caricare variabili d'ambiente da file .env in modo sicuro e configurabile

II - 2.1.4. Test

Per effettuare i test e le analisi statiche del codice sono state utilizzate le seguenti librerie:

- **Pytest** per i test di unità e integrazione
- **Black** per la formattazione automatica del codice
- **Flake8** per l'analisi statica del codice
- **MyPy** per il type checking

II - 2.2. JavaScript

JavaScript$_G$ è un linguaggio di programmazione interpretato, dinamico e multi-paradigma che consente lo sviluppo di applicazioni web interattive e real-time, eseguito nativamente dai browser moderni senza necessità di compilazione.

II - 2.2.1. Utilizzo nel progetto

Nel nostro specifico caso, viene adottato per la creazione dell'interfaccia utente interattiva della dashboard (denominata frontend_user) che si occupa di visualizzare in tempo reale i dati di localizzazione degli utenti, garantendone la sincronizzazione tramite comunicazione WebSocket$_G$ con il backend$_G$, e di gestire l'interazione con le mappe geospaziali per fornire un'esperienza utente fluida e reattiva durante la navigazione e la ricezione di offerte personalizzate.

II - 2.2.2. Versione

Per garantire compatibilità con i browser moderni è stato adottato **JavaScript ES6+$_G$ (ECMAScript 2015+)**. La scelta di utilizzare JavaScript vanilla elimina la necessità di transpilazione e bundling, garantendo performance ottimali e riducendo la complessità del deployment. Le funzionalità ES6+ utilizzate includono arrow functions, template literals, destructuring assignment e async/await per la gestione asincrona delle comunicazioni.

II - 2.2.3. Librerie e framework

Per la gestione dell'interfaccia utente e delle funzionalità geospaziali non è stato utilizzato alcun sistema di build automation, privilegiando l'inclusione diretta tramite CDN$_G$ per garantire velocità di caricamento e semplicità di manutenzione. Per avere una visione nel dettaglio di tutte le librerie utilizzate all'interno del nostro sistema, è possibile visionare il file index_user.html presente all'interno della cartella services/dashboard/frontend_user del nostro progetto. La seguente lista rappresenta le dipendenze più rilevanti presenti all'interno del progetto e non vuole essere un mero elenco di tutte le dipendenze e librerie presenti all'interno del nostro sistema frontend$_G$.

1. **Leaflet$_G$**
   - Documentazione: https://leafletjs.com/reference.html
   - Versione: Latest CDN
   - Descrizione: Framework open-source per la creazione di mappe interattive che permette, nel nostro caso, la visualizzazione in tempo reale delle posizioni degli utenti, dei percorsi di navigazione e dei punti di interesse con supporto per marker personalizzati e popup informativi.
2. **Font Awesome**
   - Documentazione: https://fontawesome.com/docs
   - Versione: 6.4.0
   - Descrizione: Libreria di icone vettoriali scalabili utilizzata per fornire elementi grafici consistenti e accessibili all'interno dell'interfaccia utente della dashboard.
3. **WebSocket API (Nativa)**
   - Documentazione: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket
   - Versione: Standard HTML5
   - Descrizione: API$_G$ nativa del browser per la comunicazione bidirezionale real-time con il backend, utilizzata per ricevere aggiornamenti di posizione e notifiche personalizzate senza necessità di polling$_G$.
4. **Fetch API (Nativa)**
   - Documentazione: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

- Versione: Standard HTML5
- Descrizione: API nativa per effettuare richieste HTTP$_G$ asincrone, utilizzata per l'autenticazione JWT$_G$, il recupero dati utente e l'interazione con gli endpoint$_G$ REST del backend.

5. **Intersection Observer API (Nativa)**
   - Documentazione: https://developer.mozilla.org/en-US/docs/Web/API/Intersection_Observer_API
   - Versione: Standard HTML5
   - Descrizione: API nativa utilizzata per implementare il lazy loading delle notifiche e ottimizzare le performance di rendering della dashboard.

II - 2.2.4. Test

Per effettuare i test e la validazione del codice JavaScript vengono utilizzati i seguenti strumenti integrati nella pipeline di sviluppo definita nel Makefile:

- **Browser DevTools** per il debugging real-time
- **ESLint** per l'analisi statica del codice (configurazione inclusa nel workflow di sviluppo)
- **Manual Testing** per la validazione dell'esperienza utente e delle funzionalità interattive

II - 2.3. Data Broker

Il sistema di messaggistica costituisce un componente cruciale all'interno della nostra architettura poiché gestisce la ricezione dei flussi di posizionamento degli utenti e li distribuisce ai moduli consumatori garantendo efficienza e scalabilità orizzontale. Nella nostra implementazione, il message broker acquisisce le coordinate di movimento dal generatore di simulazione e le trasmette al componente di elaborazione stream basato su Bytewax.

II - 2.3.1. Apache Kafka

Apache Kafka rappresenta una piattaforma distribuita di event streaming$_G$. Concepito per supportare scalabilità massiva, resilienza ai guasti e throughput$_G$ elevato, viene impiegato per orchestrare i flussi informativi real-time. Specificatamente nella nostra soluzione, Kafka orchestra i canali di trasmissione dati tra il motore di simulazione percorsi utente **(src/data_pipeline/producer.py)** e l'engine di stream processing **(src/data_pipeline/bytewax_flow.py)**.

Nonostante ciò, non l'intero ventaglio di capacità che Apache Kafka mette a disposizione degli sviluppatori è stato implementato nel nostro ecosistema, quali la duplicazione delle informazioni su cluster$_G$ multipli. Considerando il carattere prototipale dell'applicazione, si è optato per deployare una singola istanza Kafka con protezione SSL$_G$ per assicurare sicurezza nelle trasmissioni, tuttavia questo approccio non impedisce l'adozione di architetture clusterizzate, le quali possono generare benefici significativi sulla robustezza del sistema.

II - 2.3.2. Configurazione protezione comunicazioni

L'ecosistema implementa trasmissioni protette mediante SSL/TLS$_G$ con credenziali configurate nel modulo **src/configg.py**

```
# Configurazione credenziali SSL per Kafka
SSL_CAFILE = "/workspace/certs/ca.crt"
SSL_CERTFILE = "/workspace/certs/client_cert.pem"
SSL_KEYFILE = "/workspace/certs/client_key.pem"
```

L'architettura prevede validazione client-side obbligatoria e cifratura end-to-end per preservare integrità e riservatezza delle informazioni di posizionamento degli utilizzatori.

Schema dei messaggi di posizionamento

Ogni informazione di coordinate emessa dal generatore di simulazione è strutturata come oggetto JSON$_G$ contenente le seguenti proprietà:

```
{
    "user_id": 1,
    "latitude": 45.464664,
    "longitude": 9.188540,
    "timestamp": "2025-01-27T14:30:45.123456+00:00",
    "age": 28,
    "profession": "Ingegnere",
    "interests": "tecnologia, viaggi, cucina"
}
```

- **user_id**: Codice identificativo univoco dell'utilizzatore che ha generato l'informazione di posizionamento (si rimanda alla documentazione delle entità utente in ClickHouse per approfondimenti riguardo l'oggetto User).
- **latitude**: Coordinata latitudinale della localizzazione attuale dell'utilizzatore espressa in notazione decimale secondo lo standard WGS84$_G$.
- **longitude**: Coordinata longitudinale della localizzazione attuale dell'utilizzatore espressa in notazione decimale secondo lo standard WGS84.
- **timestamp**: Marcatura temporale dell'informazione di posizionamento in notazione ISO 8601$_G$ con fuso orario UTC$_G$. Questo attributo risulta essenziale per l'amministrazione della persistenza informativa all'interno di ClickHouse e per la creazione degli annunci targettizzati, poiché consente di prevenire sovrapposizioni tra molteplici informazioni di coordinate emesse dal medesimo utilizzatore.
- **age**: Anagrafica dell'utilizzatore impiegata per la customizzazione delle proposte commerciali e del targeting promozionale.
- **profession**: Occupazione lavorativa dell'utilizzatore sfruttata dal motore di generazione messaggi LLM per produrre contenuti contestualmente pertinenti.

- **interests**: Elenco di passioni dell'utilizzatore delimitate da virgola, adoperate per l'associazione con le tipologie dei punti vendita e la personalizzazione delle comunicazioni commerciali.

Questi payload$_G$ vengono elaborati dal consumer Bytewax che si occupa dell'enrichment con dettagli sui punti di interesse circostanti e della creazione di comunicazioni personalizzate tramite integrazione con il servizio LLM.

II - 2.4. Stream Processor

L'engine di elaborazione rappresenta il nucleo operativo dell'intera soluzione sviluppata dal team. Esso gestisce l'acquisizione dei flussi di coordinate, li arricchisce con metadati necessari alla formulazione della richiesta da inoltrare al modello linguistico e garantisce la persistenza di queste informazioni all'interno del sistema di storage.

II - 2.4.1. Bytewax

Bytewax costituisce un framework di processing distribuito che consente di eseguire trasformazioni definite stateful$_G$ su flussi di informazioni in ingresso, bounded o unbounded che siano. È architettato per operazioni continue con latenze$_G$ e tempi di risposta estremamente ridotti. Nella nostra implementazione Bytewax viene impiegato per elaborare i payload di posizionamento real-time provenienti dai simulatori, assicurandone la memorizzazione all'interno del database e, partendo da questi payload, recuperare il maggior numero di metadati possibili allo scopo di generare la comunicazione più appropriata da recapitare all'utilizzatore finale.

Il dataflow$_G$ implementato orchestra le seguenti operazioni:

- **Parsing**: Deserializzazione messaggi Kafka in oggetti Python strutturati
- **Enrichment**: Arricchimento con informazioni geospaziali sui negozi circostanti via PostGIS
- **AI Integration**: Chiamata al servizio di generazione messaggi personalizzati
- **Persistence**: Memorizzazione eventi processati in ClickHouse per analytics

II - 2.4.2. Creazione comunicazioni personalizzate

Le specifiche richiedono l'impiego di modelli linguistici per la creazione delle comunicazioni sfruttando come input le preferenze dell'utilizzatore finale, la tipologia commerciale e le proposte del punto vendita più prossimo alla localizzazione del sensore. È importante sottolineare che l'importanza maggiore dei parametri risiede nei campi testuali liberi (preferenze dell'utilizzatore e proposte del punto vendita) poiché i modelli linguistici sono specializzati nell'interpretazione di queste tipologie di input.

II - 2.4.3. Architettura LLM

Il sistema implementa una logica di generazione messaggi. Questa implementazione garantisce:

- **Scalabilità indipendente**: Il servizio può essere scalato autonomamente in base al carico
- **Fault tolerance**: Errori nella generazione non compromettono il resto del sistema

- **Provider agnostic**: Supporto per multipli fornitori LLM tramite configurazione

II - 2.4.4. LangChain Python Integration

La nostra soluzione adopera LangChain nella sua implementazione Python, una libreria che semplifica l'integrazione di modelli linguistici con applicazioni Python. Fornisce un ecosistema di strumenti per operare con LLM, inclusa la costruzione di prompt templates$_G$ e la gestione delle risposte tramite il modello stesso.

LangChain supporta numerosi provider di modelli linguistici: uno tra questi, da noi implementato, è **Gemma2-9B-IT**$_G$ tramite Groq$_G$. Questo modello è stato selezionato per una ragione specifica: consentire al team di sviluppare il progetto utilizzando API gratuite ad alta velocità, sfruttando l'infrastruttura Groq per inferenza$_G$ ottimizzata.

Il provider può essere facilmente sostituito per permettere l'utilizzo di altri modelli grazie alla modularità del sistema implementato:

```
# Configurazione multi-provider flessibile
if PROVIDER == "openai":
    model_name = "gpt-4o-mini"
elif PROVIDER == "groq":
    model_name = "gemma2-9b-it"
else:
    model_name = "gpt-3.5-turbo"  # Default fallback
```

La generazione dei messaggi integra inoltre un sistema di cache Redis per ottimizzare le performance e ridurre i costi di inferenza, memorizzando risposte precedenti per combinazioni simili di parametri utente e negozio.

II - 2.5. Sistema di Persistenza (Database)

II - 2.5.1. PostgreSQL con PostGIS

Per la gestione delle informazioni relazionali e geospaziali è stato selezionato PostgreSQL, un RDBMS$_G$ che garantisce robustezza e notevole versatilità per l'espansione tramite moduli ed estensioni. Nel nostro ecosistema, durante l'inizializzazione viene eseguito automaticamente lo script **deployment/scripts/init_postgres.sh** che costruisce lo schema del database (entità, associazioni, vincoli) secondo i requisiti del progetto e inserisce i metadati necessari a validare il funzionamento del nostro ecosistema.

II - 2.5.2. Estensione PostGIS

Per l'elaborazione e la memorizzazione di informazioni geografiche si utilizza l'estensione PostGIS, la quale arricchisce PostgreSQL con il supporto per tipologie, operatori e indici spaziali. Specificatamente l'immagine Docker impiegata è **postgis/postgis:15-3.3**. Oltre a PostgreSQL questa include già la libreria PostGIS e le relative dipendenze. Questa configurazione consente, nella nostra implementazione, di:

- Memorizzare le coordinate geografiche (latitudine e longitudine) dei punti vendita e delle localizzazioni trasmesse real-time da ogni utilizzatore attivo.

- Eseguire interrogazioni geospaziali all'interno del database per identificare i potenziali punti vendita in relazione ad una specifica localizzazione ed entro un determinato raggio di prossimità.

II - 2.5.3. ClickHouse

Per la gestione di grandi volumi di dati analitici e telemetria utente è stato implementato ClickHouse, un DBMS$_G$ colonnare ottimizzato per query analitiche OLAP$_G$. Il setup automatizzato tramite **deployment/scripts/init_clickhouse.sh** configura:

- **Aggregazione eventi**: Memorizzazione efficiente di milioni di eventi di posizionamento utente
- **Analytics real-time**: Supporto per dashboard Grafana con query sub-secondo
- **Data retention**: Partizionamento automatico per gestione lifecycle dati

II - 2.5.4. Architettura Database: schema PostgreSQL (Negozi e Offerte)

```
-- Tabella principale negozi
CREATE TABLE shops (
    shop_id SERIAL PRIMARY KEY,
    shop_name VARCHAR(255),
    address TEXT,
    category VARCHAR(100),
    geom GEOMETRY(Point, 4326),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Tabella offerte commerciali
CREATE TABLE offers (
    offer_id SERIAL PRIMARY KEY,
    shop_id INTEGER NOT NULL REFERENCES shops(shop_id) ON DELETE CASCADE,
    discount_percent INTEGER NOT NULL CHECK (discount_percent > 0 AND discount_percent <= 50),
    description TEXT NOT NULL,
    offer_type VARCHAR(20) DEFAULT 'percentage',
    valid_from DATE DEFAULT CURRENT_DATE,
    valid_until DATE NOT NULL,
    is_active BOOLEAN DEFAULT true,
    max_uses INTEGER DEFAULT NULL,
    current_uses INTEGER DEFAULT 0,
    min_age INTEGER DEFAULT NULL,
    max_age INTEGER DEFAULT NULL,
    target_categories TEXT[],
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

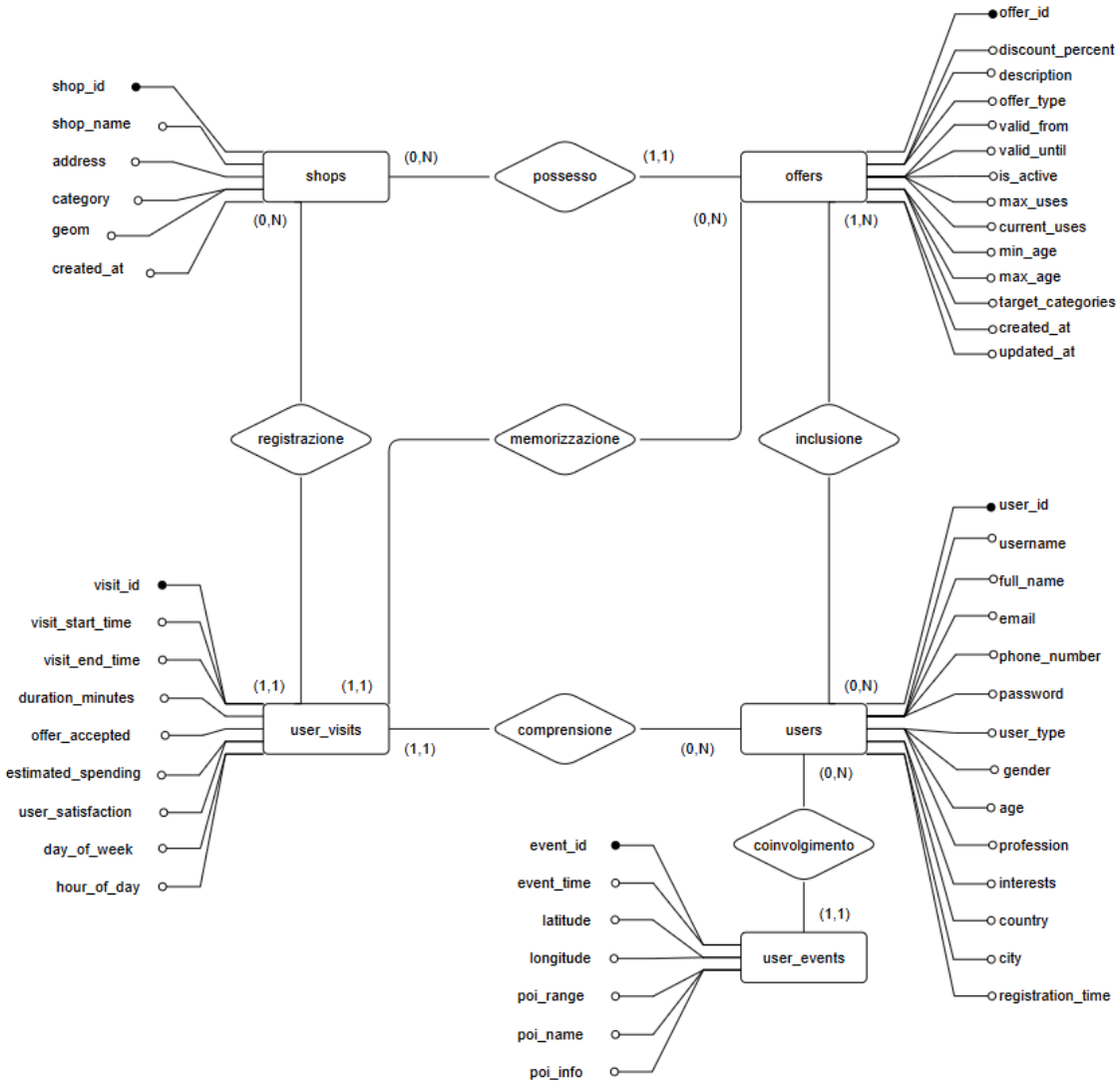
II - 2.5.5. Architettura Database: Schema ClickHouse (Eventi e Analytics)

```
-- Tabella profili utente
CREATE TABLE users (
    user_id UInt64,
    username String,
    full_name String,
    email String,
    phone_number String,
    password String,
    user_type String,
    gender String,
    age UInt32,
    profession String,
    interests String,
    country String,
    city String,
    registration_time DateTime
) ENGINE = MergeTree()
ORDER BY user_id;

-- Tabella eventi posizione
CREATE TABLE user_events (
    event_id UInt64,
    event_time DateTime,
    user_id UInt64,
    latitude Float64,
    longitude Float64,
    poi_range Float64,
    poi_name String,
    poi_info String
) ENGINE = MergeTree()
ORDER BY event_id;
```

```
-- Tabella visite simulate
CREATE TABLE user_visits (
    visit_id UInt64,
    user_id UInt64,
    shop_id UInt64,
    offer_id UInt64 DEFAULT 0,
    visit_start_time DateTime,
    visit_end_time DateTime DEFAULT toDateTime(0),
    duration_minutes UInt32 DEFAULT 0,
    offer_accepted Boolean DEFAULT false,
    estimated_spending Float32 DEFAULT 0.0,
    user_satisfaction UInt8 DEFAULT 5,
    day_of_week UInt8 DEFAULT toDayOfWeek(visit_start_time),
    hour_of_day UInt8 DEFAULT toHour(visit_start_time),
    weather_condition String DEFAULT '',
    user_age UInt8 DEFAULT 0,
    user_profession String DEFAULT '',
    user_interests String DEFAULT '',
    shop_name String DEFAULT '',
    shop_category String DEFAULT '',
    created_at DateTime DEFAULT now()
) ENGINE = MergeTree()
PARTITION BY toYYYYMM(visit_start_time)
ORDER BY (user_id, visit_start_time, shop_id);
```

II - 2.5.6. Diagramma Relazionale



II - 2.5.7. Decisioni Architetturali

Alcune scelte progettuali, apparentemente ridondanti, sono state implementate per soddisfare esigenze specifiche, in particolare per strumenti di monitoraggio come Grafana e ottimizzazioni performance.

1. **Strategie di Chiavi Primarie**:
   - **Chiavi surrogate vs naturali**: La tabella shops utilizza una chiave surrogata (shop_id SERIAL) per garantire stabilità dei riferimenti anche in caso di modifiche ai dati geografici. Tuttavia, per le query geospaziali, la colonna geom (Point geometry) funge da identificatore naturale tramite indici spaziali PostGIS.
   - **Chiavi composte temporali**: La tabella user_events adotta un approccio ibrido con event_id come chiave primaria e un indice composto su (user_id, event_time) per ottimizzare le query di time-series[G]. Questo garantisce l'univocità di ogni evento registrato per utente ed evita conflitti temporali.
2. **Denormalizzazione Strategica**:

- **Snapshot dati utente**: La tabella user_visits include campi denormalizzati (user_age, user_profession, shop_name, shop_category) per evitare JOIN$_G$ costosi durante l'aggregazione analitica. Questo trade-off tra spazio di storage e performance è essenziale per dashboard real-time Grafana.
- **Snapshot dati utente**: La tabella user_visits include campi denormalizzati (user_age, user_profession, shop_name, shop_category) per evitare JOIN costosi durante l'aggregazione analitica. Questo trade-off tra spazio di storage e performance è essenziale per dashboard real-time Grafana.

3. **Ottimizzazioni Geospaziali**
   - **Coordinate vs ID geografici**: Le tabelle mantengono sia coordinate geografiche (latitude, longitude) che riferimenti a entità (shop_id) per supportare due pattern di query: ricerca geospaziale diretta tramite PostGIS e join relazionali per analytics business. Questo duplicato migliora significantly le performance delle query di prossimità.
   - **Partizionamento temporale**: ClickHouse partiziona automaticamente user_visits per mese (PARTITION BY toYYYYMM(visit_start_time)) garantendo performance costanti anche con crescita esponenziale dei dati, facilitando la visualizzazione time-series in Grafana.

Queste decisioni bilanciano performance, manutenibilità e integrazione con strumenti di business intelligence, mantenendo la flessibilità per future evoluzioni architetturali.

II - 2.6. Interfaccia utente

Il sistema NearYou fornisce due interfacce distinte per soddisfare esigenze differenti: un'interfaccia utente real-time per l'esperienza finale e un sistema di monitoraggio per l'amministrazione e l'analisi dei dati.

II - 2.6.1. Dashboard Utente Real-Time

L'interfaccia utente principale (services/dashboard/frontend_user/) fornisce un'esperienza interattiva real-time che permette agli utilizzatori di:

- **Visualizzazione posizione live**: Tracking$_G$ in tempo reale della posizione utente con aggiornamenti via WebSocket ogni 3 secondi, mostrando percorsi di navigazione e negozi nelle vicinanze.
- **Mappa interattiva**: Implementazione con Leaflet che visualizza marker dinamici per negozi categorizzati, con popup informativi contenenti offerte personalizzate e dettagli punti vendita.
- **Notifiche personalizzate**: Ricezione real-time di messaggi generati dall'LLM quando l'utente si trova in prossimità di negozi con offerte attive, con sistema di cache per ottimizzare le performance.
- **Gestione profilo**: Visualizzazione statistiche personali (distanza percorsa, negozi visitati, notifiche ricevute) e cronologia delle promozioni ricevute con lazy loading$_G$.

L'interfaccia utilizza comunicazione WebSocket bidirezionale per garantire aggiornamenti istantanei senza polling, implementando pattern di reconnection automatica e gestione errori per massima affidabilità.

II - 2.7. Sistema di Monitoraggio Admin - Grafana

Grafana$_G$ non costituisce un sistema «reattivo», ovvero non reagisce direttamente agli eventi, bensì acquisisce i dati tramite query periodiche indipendentemente dalle modifiche nel database. Per questa ragione non è tecnicamente appropriato definirla «interfaccia real-time», tuttavia le interrogazioni vengono eseguite a intervalli molto ravvicinati (5-15 secondi) simulando quindi con elevata precisione il comportamento dell'interfaccia desiderata.

Le funzionalità principali di Grafana nel nostro ecosistema sono:

- **Monitoraggio analytics**: Grafana raccoglie continuamente i dati degli utilizzatori registrati nel sistema, ovvero identificativo utente, eventi di posizionamento associati, latitudine, longitudine e metadati comportamentali.
- **Visualizzazione dati geospaziali**: I dati di posizionamento acquisiti vengono mostrati in dashboard di tipo geomap interattive, dove le posizioni degli utenti sono rappresentate da layer di tipo route e i punti vendita con relativi messaggi tramite layer di tipo marker con colorazione per categoria.
- **Analytics Comportamentali Dettagliate**: Storico Visite Simulate: Il sistema traccia automaticamente le visite simulate degli utenti presso i negozi tramite la tabella user_visits in ClickHouse. Tra le visualizzazioni implementate vi sono:
  1. **Heatmap$_G$ Geografica**: Mappa interattiva che mostra concentrazione visite per zona di Milano, con intensità colore basata su revenue generato
  2. **Timeline Comportamenti**: Grafici temporali che evidenziano picchi di attività per fascia oraria e giorno della settimana
  3. **Funnel$_G$ Conversione**: Visualizzazione del percorso utente da posizionamento → prossimità → messaggio → visita → acquisto
  4. **Segmentazione Utenti**: Analisi comportamentale per età, professione e interessi, mostrando pattern di preferenza per categoria negozi
- **Metriche di Business Specifiche**: Dashboard dedicate per monitorare KPI$_G$ come conversion rate delle offerte, revenue per categoria, pattern temporali di utilizzo e performance del sistema di raccomandazioni LLM.

III. Architettura del sistema

NearYou ha adottato la **Lambda Architecture**$_G$ dopo una valutazione mirata dell'alternativa Kappa$_G$. L'analisi ha mostrato che mantenere tutto (inclusi i dati dei negozi / POI$_G$) nel flusso real-time avrebbe introdotto costi senza reale beneficio: i POI cambiano lentamente e non richiedono propagazione sub-secondo.

Motivazioni della scelta (valutazione Kappa → Lambda):

- Frequenza aggiornamento POI: bassa (variazioni giornaliere / settimanali), quindi lo stream li «sovra-tratta».
- Impatto sulla latenza critica: concorrenza inutile con eventi dinamici (posizioni utente, visite simulate, notifiche).
- Complessità operativa: più invalidazioni cache (profili ↔ POI) e maggiore rumore nelle metriche di throughput.

- Requisiti analitici: necessità di ricalcoli completi (feature, segmenti, scoring) meglio serviti da job batch controllati.
- Manutenibilità: separare calcolo intensivo (batch) e reattivo (speed) facilita tuning e scaling indipendenti.

Struttura Lambda applicata:

- Batch Layer$_G$: aggiorna periodicamente POI, offerte, feature e pulisce / consolida lo storico (Airflow ⟶ PostGIS / ClickHouse).
- Speed Layer$_G$: gestisce solo ciò che varia rapidamente (eventi posizione, simulazioni visita, generazione messaggi personalizzati via LLM).
- Serving Layer$_G$: espone una vista coerente fondendo «storico consolidato» + «delta recenti» con logiche di watermark$_G$ e priorità ai dati più freschi.

Benefici concreti:

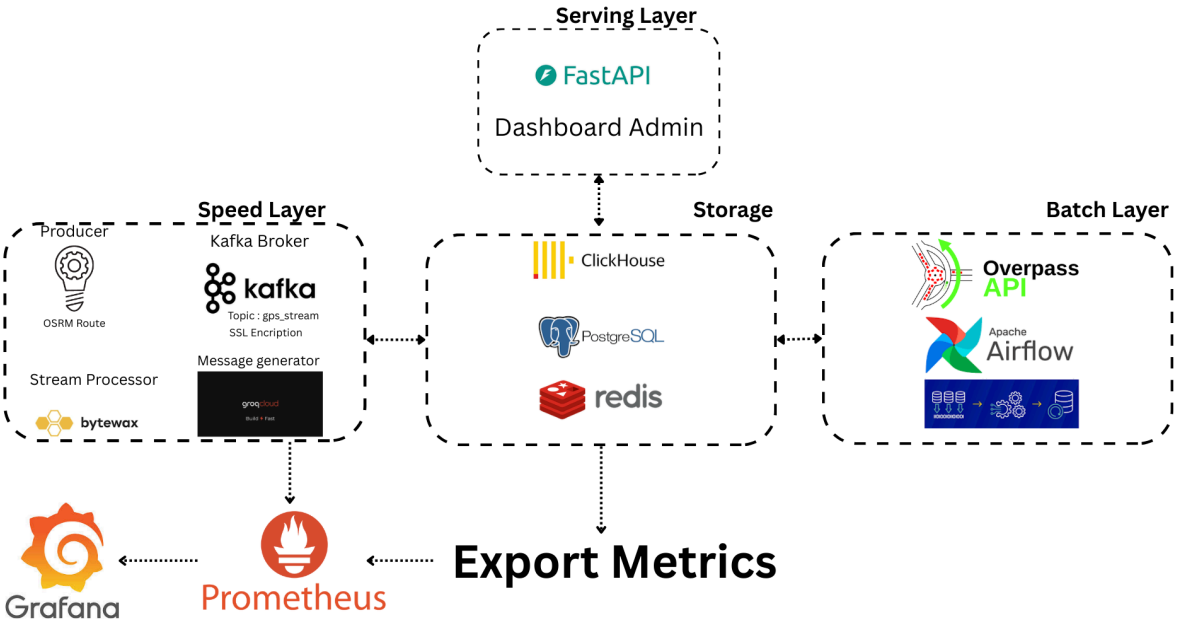
- Latenza sub-secondo preservata per notifiche contestuali.
- Riduzione dei costi (meno messaggi, meno cache invalidata, meno I/O superfluo).
- Ricalcoli batch controllati per qualità storica e correzione drift.
- Evoluzione agile: nuove feature introdotte prima nel batch, poi eventualmente ottimizzate nello speed se necessario.

In sintesi, la decisione nasce dall'allineamento tra natura dei dati (POI stabili vs movimento utenti dinamico) e obiettivi operativo-funzionali: il modello Lambda evita di «simulare dinamicità» dove non serve.

III - 1. Lambda Architecture

La Lambda Architecture consente di combinare:

- Calcolo incrementale a bassa latenza (speed) per eventi di posizione e generazione notifiche
- Ricalcoli completi e arricchimenti intensivi (batch) per dataset storici, feature e modelli
- Un layer di serving che unifica vista «storica consolidata» + «delta recenti»



III - 1.1. Strati Principali

III - 1.1.1. Batch Layer

Funzioni principali:

- Acquisizione e normalizzazione POI (Overpass API ⟶ PostGIS)
- Ricalcolo aggregati storici e metriche di comportamento (ClickHouse)
- Generazione e aggiornamento feature utente (propensione visita, vettori interessi)
- Precomputazione e consolidamento rotte ciclabili (OSRM$_G$ offline)
- Costruzione tabelle gold (negozi, offerte, mapping categorie, segmenti)
- Pulizia e compattazione eventi (merge partizioni, deduplicazione)

Orchestrazione: Apache Airflow (cicli giornalieri / orari). Output persistito in:

- ClickHouse (storico eventi + aggregati)
- PostgreSQL + PostGIS (geospaziale, relazioni commerciali)

III - 1.1.2. Speed Layer

Responsabile della reattività:

- Ingest real-time posizioni via Kafka (topic: gps_stream, partition key=user_id)
- Pipeline Bytewax:
  1. Prossimità geospaziale (query PostGIS, raggio 200m configurabile)
  2. Enrichment rapido profilo (Redis ⟶ fallback snapshot ClickHouse)
  3. Generazione messaggio personalizzato (servizio LLM + cache Redis)
  4. Simulazione probabilistica visita (emissione user_visit_delta)
- Persistenza delta «hot» (events_delta, user_visit_delta) in ClickHouse
- Gestione TTL$_G$ cache (profili, messaggi LLM) per mantenere freschezza

III - 1.1.3. Serving Layer

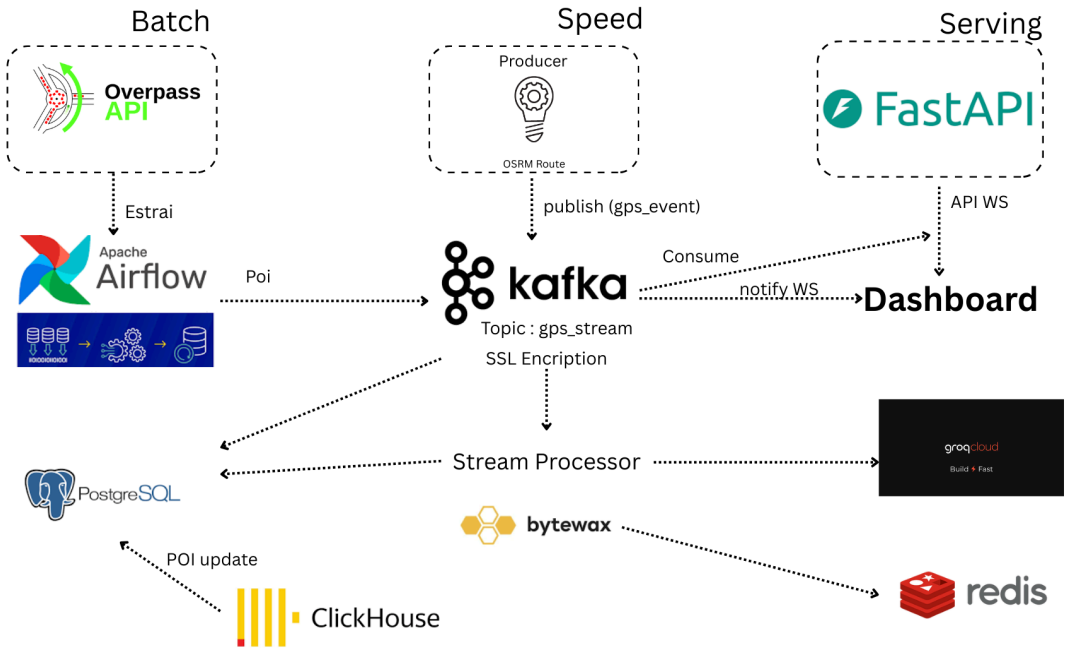
Compone viste logiche e servizi:

- Materializzazione/virtualizzazione: unione batch_base + delta recenti (finestra temporale > last_compaction_ts)
- API (REST/WebSocket) per:
  - Stato corrente utente (posizione, POI vicini, offerte personalizzate)
  - Query analitiche storiche (solo fonte batch)
- Dashboard real-time:
  - WebSocket per streaming posizioni e notifiche
  - Mappa interattiva con layer dinamici (rotte, POI, heatmap visite)
- Redis come acceleration layer (profilo utente a caldo, messaggi LLM già calcolati)

III - 1.2. Componenti Tecnologici

- Data Sources: Simulatore movimento (OSRM), Overpass API, generatori offerte batch
- Messaging: Apache Kafka (SSL/TLS, consumer group gps_consumers_group)
- Stream Processing: Bytewax (operatori custom business logic)
- Batch Orchestration: Airflow (DAG$_G$ di estrazione, feature engineering, compattazione)
- Storage:
  - ClickHouse (eventi time-series, analytics OLAP, viste unificate)
  - PostgreSQL + PostGIS (POI, geometrie, offerte, relazioni commerciali)
  - Redis (cache profili, risposte LLM, short-lived state)
- LLM Message Generator: servizio HTTP con caching semantico
- Observability: Prometheus$_G$ (metriche), Grafana (dashboard), log centralizzati

III - 1.3. Flusso dei Dati



1. Ingest Posizioni:
   - Producer simula movimento utenti (OSRM) e pubblica eventi GPS su Kafka (gps_stream)
2. Speed Processing:
   - Bytewax consuma in ordine per utente

- Enrichment geospaziale (PostGIS) + profilo (Redis/ClickHouse)
- Generazione messaggi personalizzati (LLM) + caching
- Emissione notifiche (WebSocket) e scrittura delta (ClickHouse)

3. Batch Processing:
   - Airflow estrae nuovi POI, rigenera feature e aggregati
   - Compatta delta in tabelle batch canonicali
   - Aggiorna segmentazioni / offerte, invalida cache profili obsoleti
4. Serving Unificato:
   - Query combinano dati batch + delta con watermark temporale
   - Dashboard riceve stream posizioni + notifiche push
5. Monitoring & Feedback Loop:
   - Metriche ingest lag, throughput, LLM latency, cache hit ratio
   - Alert su deviazioni SLA_G (lag, error rate, memoria cache)

III - 1.4. Consistenza e Fusione

- Modello dati append-only nei delta; compattazione batch sostituisce partizioni storiche
- Vista logica: SELECT **FROM events_batch WHERE ts < :watermark UNION ALL SELECT** FROM events_delta WHERE ts >= :watermark
- Deduplicazione su (user_id, event_uuid) durante compattazione
- Priorità ai record più recenti (delta) in caso di conflitto
- Topic di controllo (batch_sync) per invalidazione cache coordinata

III - 1.5. Riepilogo

NearYou applica pienamente la Lambda Architecture per conciliare latenza real-time e consistenza storica verificabile. La separazione funzionale dei layer permette evoluzione rapida della personalizzazione (LLM + feature store) preservando affidabilità e scalabilità orizzontale dell'intero ecosistema dati.

III - 2. Diagramma delle classi

Il diagramma illustra l'organizzazione strutturale del sistema attraverso le classi implementate e le loro interdipendenze. Le classi sono raggruppate per domini funzionali, omettendo i dettagli implementativi delle librerie esterne per mantenere la leggibilità.

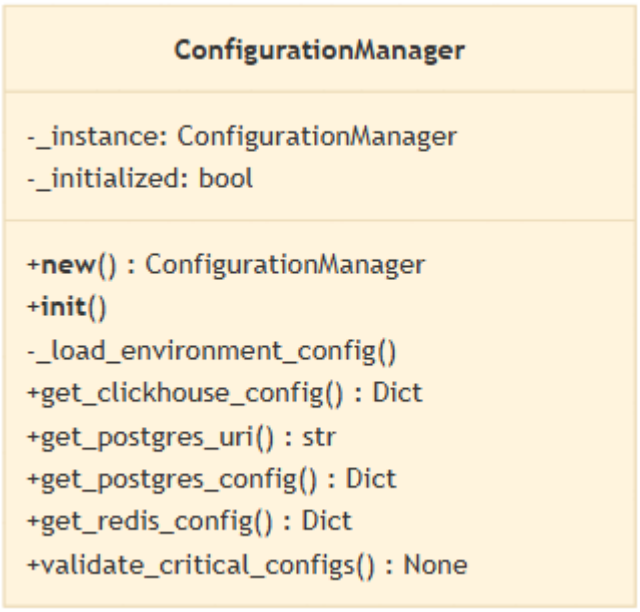
IV. Documentazione delle Classi - Sistema NearYou

IV - 1. Gestione Configurazione

Questa sezione rappresenta le classi impiegate nella gestione della configurazione del sistema.

IV - 1.1. Struttura delle classi

IV - 1.1.1. ConfigurationManager



Attributi

- `_instance: Optional[ConfigurationManager]` - Istanza singleton della classe
- `_initialized: bool` - Flag per prevenire re-inizializzazione
- `environment: str` - Ambiente di esecuzione (development, production, staging)
- `kafka_broker: str` - Indirizzo del broker Kafka
- `kafka_topic: str` - Topic Kafka per i messaggi GPS
- `clickhouse_host: str` - Host del database ClickHouse
- `postgres_host: str` - Host del database PostgreSQL
- `jwt_secret: str` - Chiave segreta per JWT
- `redis_host: str` - Host del server Redis per cache

Costruttori

- `__new__() -> ConfigurationManager` - Implementa pattern Singleton
- `__init__()` - Inizializza configurazioni da variabili d'ambiente

Metodi

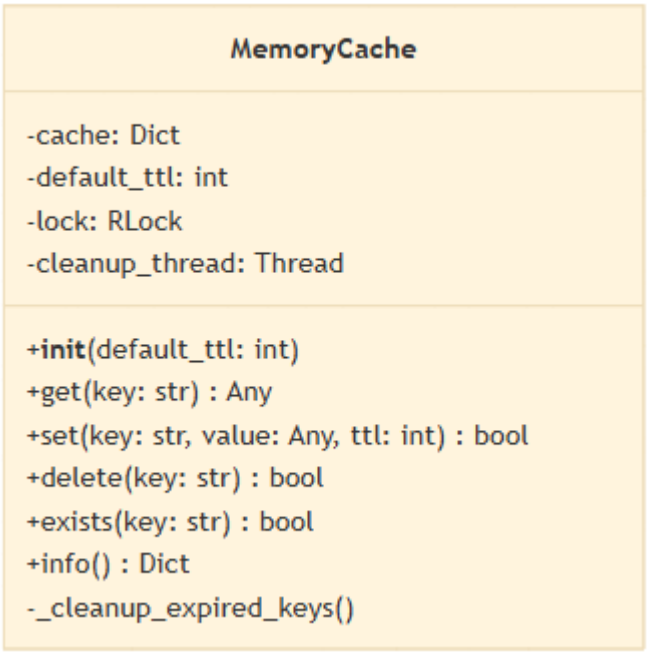
- `_load_environment_config()` - Carica tutte le configurazioni dalle variabili d'ambiente
- `get_clickhouse_config() -> Dict[str, Any]` - Restituisce configurazione ClickHouse
- `get_postgres_uri() -> str` - Genera URI di connessione PostgreSQL
- `get_postgres_config() -> Dict[str, Any]` - Restituisce configurazione PostgreSQL come dictionary
- `get_redis_config() -> Dict[str, Any]` - Restituisce configurazione Redis
- `validate_critical_configs() -> None` - Valida configurazioni critiche per ambienti di produzione

IV - 2. Sistema di Cache

Questa sezione rappresenta le classi per la gestione del caching nel sistema.

IV - 2.1. Struttura delle classi

IV - 2.1.1. MemoryCache



Attributi

- `cache: Dict` - Dictionary per memorizzare coppie chiave-valore con scadenza
- `default_ttl: int` - Time-to-live predefinito in secondi (default 86400)
- `lock: threading.RLock` - Lock per thread-safety
- `cleanup_thread: threading.Thread` - Thread per pulizia automatica chiavi scadute

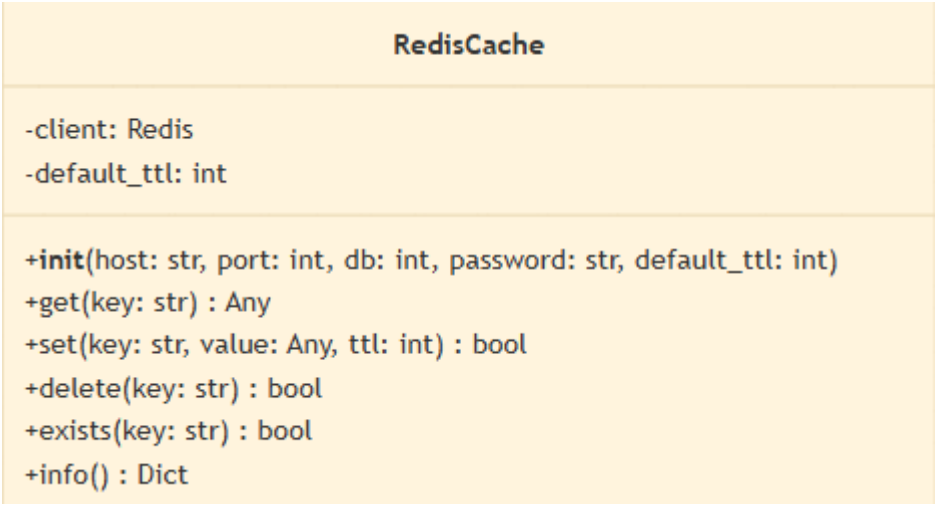
Costruttori

- `__init__(default_ttl: int = 86400)` - Inizializza cache in-memory con pulizia periodica

Metodi

- `get(key: str) -> Optional[Any]` - Recupera valore dalla cache
- `set(key: str, value: Any, ttl: Optional[int] = None) -> bool` - Salva valore nella cache
- `delete(key: str) -> bool` - Elimina chiave dalla cache
- `exists(key: str) -> bool` - Verifica se la chiave esiste ed è valida
- `info() -> Dict[str, Any]` - Restituisce statistiche sulla cache
- `_cleanup_expired_keys()` - Thread di background per pulizia chiavi scadute

IV - 2.1.2. RedisCache



Attributi

- `client: redis.Redis` - Client Redis per connessione al server
- `default_ttl: int` - Time-to-live predefinito per le chiavi

Costruttori

- `__init__(host: str, port: int, db: int, password: str, default_ttl: int)` - Inizializza connessione Redis

Metodi

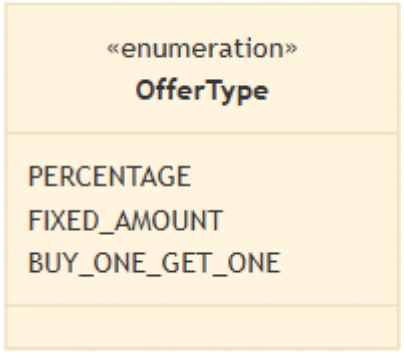
- `get(key: str) -> Optional[Any]` - Recupera valore da cache con deserializzazione JSON
- `set(key: str, value: Any, ttl: Optional[int] = None) -> bool` - Salva valore con serializzazione JSON
- `delete(key: str) -> bool` - Elimina chiave dalla cache Redis
- `exists(key: str) -> bool` - Verifica esistenza chiave
- `info() -> Dict[str, Any]` - Restituisce statistiche del server Redis

IV - 3. Modelli Dati

Questa sezione rappresenta le classi che definiscono i modelli dati del sistema.

IV - 3.1. Struttura delle classi

IV - 3.1.1. OfferType



Attributi

- `PERCENTAGE: str` - Sconto percentuale
- `FIXED_AMOUNT: str` - Importo fisso di sconto
- `BUY_ONE_GET_ONE: str` - Promozione prendi 2 paghi 1

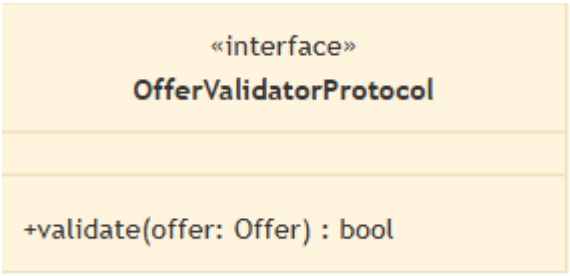
Costruttori

Enum$_G$ definito staticamente con valori costanti per i tipi di offerta disponibili.

Metodi

Nessun metodo specifico, utilizzato come enum per tipizzazione delle offerte.

IV - 3.1.2. OfferValidatorProtocol



Attributi

Nessun attributo, definisce solo l'interfaccia per la validazione.

Costruttori

Protocol interface$_G$, non ha costruttori diretti.

Metodi

- `validate(offer: Offer) -> bool` - Metodo astratto per validare un'offerta

IV - 3.1.3. OfferValidator

Attributi

Nessun attributo di istanza.

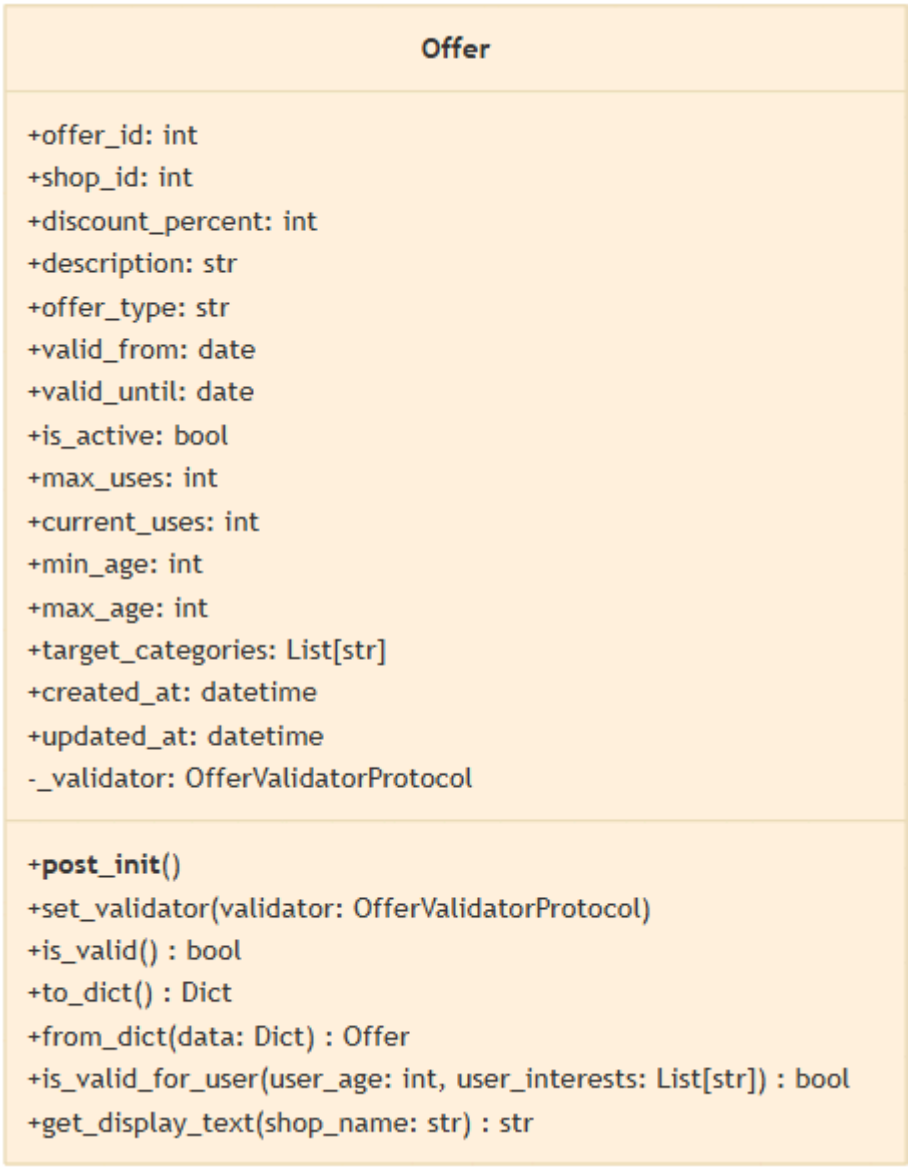
Costruttori

- `__init__()` - Costruttore standard per implementazione default

Metodi

- `validate(offer: Offer) -> bool` - Valida vincoli base dell'offerta (percentuale, date, età)

IV - 3.1.4. Offer



Attributi

- `offer_id: Optional[int]` - ID univoco dell'offerta
- `shop_id: int` - ID del negozio che offre la promozione
- `discount_percent: int` - Percentuale di sconto (0-100)
- `description: str` - Descrizione testuale dell'offerta
- `offer_type: str` - Tipo di offerta (da OfferType enum)
- `valid_from: Optional[date]` - Data inizio validità
- `valid_until: Optional[date]` - Data fine validità
- `is_active: bool` - Flag attivazione offerta
- `max_uses: Optional[int]` - Numero massimo di utilizzi
- `current_uses: int` - Utilizzi attuali
- `min_age: Optional[int]` - Età minima target
- `max_age: Optional[int]` - Età massima target
- `target_categories: Optional[List[str]]` - Categorie di interesse target

- `_validator: OfferValidatorProtocol` - Validatore per l'offerta

Costruttori

- `__init__()` - Inizializza offerta con parametri opzionali
- `__post_init__()` - Post-inizializzazione per valori di default

Metodi

- `set_validator(validator: OfferValidatorProtocol) -> None` - Imposta validatore custom
- `is_valid() -> bool` - Verifica validità usando il validatore corrente
- `to_dict() -> Dict[str, Any]` - Converte in dictionary per DB
- `from_dict(data: Dict[str, Any]) -> Offer` - Crea istanza da dictionary
- `is_valid_for_user(user_age: int, user_interests: List[str]) -> bool` - Verifica compatibilità utente
- `get_display_text(shop_name: str) -> str` - Genera testo descrittivo per display

IV - 3.1.5. OfferBuilder

Attributi

- `_offer: Offer` - Istanza dell'offerta in costruzione

Costruttori

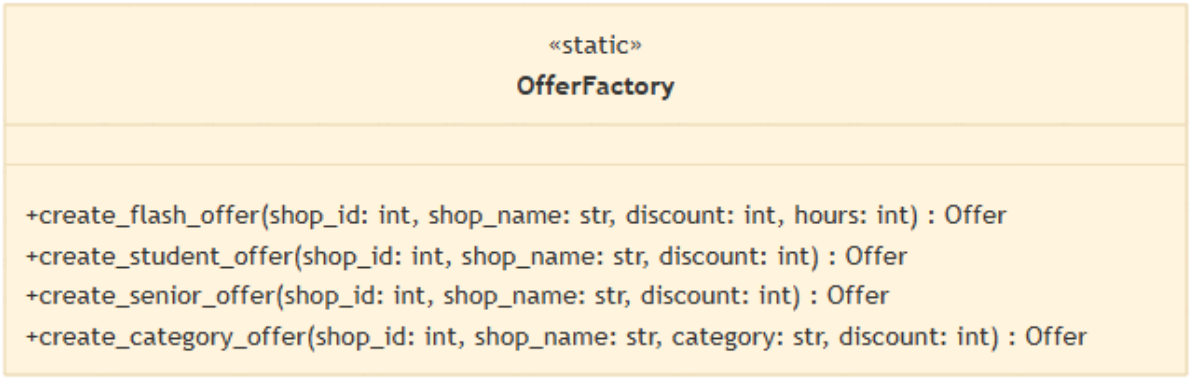
- `__init__()` - Inizializza builder$_G$ e resetta stato

Metodi

- `reset() -> OfferBuilder` - Resetta builder per nuovo utilizzo

- `shop(shop_id: int) -> OfferBuilder` - Imposta ID negozio
- `discount(percentage: int) -> OfferBuilder` - Imposta percentuale sconto
- `description(text: str) -> OfferBuilder` - Imposta descrizione
- `offer_type(offer_type: OfferType) -> OfferBuilder` - Imposta tipo offerta
- `valid_period(from_date: date, until_date: date) -> OfferBuilder` - Imposta periodo validità
- `valid_for_days(days: int) -> OfferBuilder` - Imposta validità in giorni da oggi
- `max_uses(uses: int) -> OfferBuilder` - Imposta numero massimo utilizzi
- `age_target(min_age: int, max_age: int) -> OfferBuilder` - Imposta target età
- `interest_target(categories: List[str]) -> OfferBuilder` - Imposta target interessi
- `active(is_active: bool) -> OfferBuilder` - Imposta stato attivazione
- `build() -> Offer` - Costruisce e valida l'offerta finale
- `build_unsafe() -> Offer` - Costruisce senza validazione

IV - 3.1.6. OfferFactory



Attributi

Nessun attributo di istanza, tutti metodi statici.

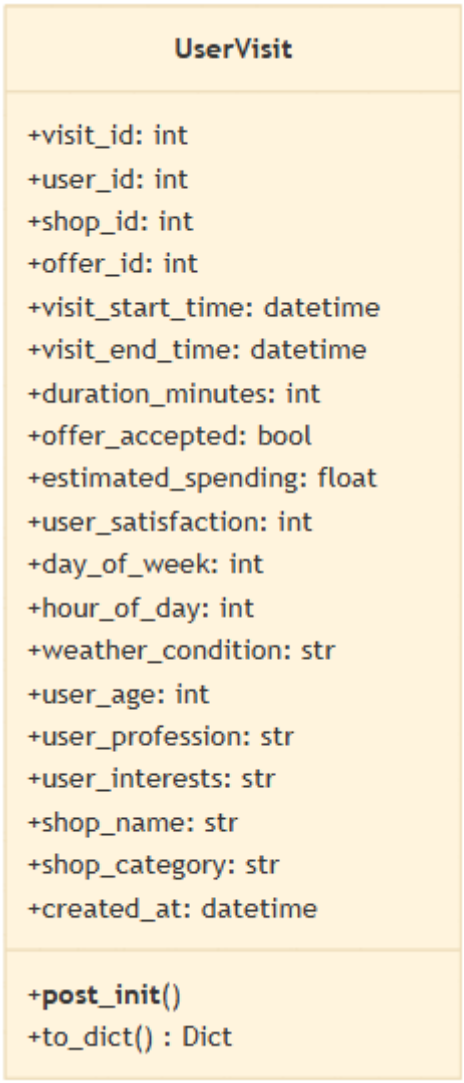
Costruttori

Classe con soli metodi statici, nessun costruttore necessario.

Metodi

- `create_flash_offer(shop_id: int, shop_name: str, discount: int, hours: int) -> Offer` - Crea offerta flash a breve termine
- `create_student_offer(shop_id: int, shop_name: str, discount: int) -> Offer` - Crea offerta per studenti
- `create_senior_offer(shop_id: int, shop_name: str, discount: int) -> Offer` - Crea offerta per senior
- `create_category_offer(shop_id: int, shop_name: str, category: str, discount: int) -> Offer` - Crea offerta specifica per categoria

IV - 3.1.7. UserVisit



Attributi

- `visit_id: Optional[int]` - ID univoco della visita
- `user_id: int` - ID dell'utente
- `shop_id: int` - ID del negozio visitato
- `offer_id: int` - ID dell'offerta utilizzata
- `visit_start_time: Optional[datetime]` - Orario inizio visita
- `visit_end_time: Optional[datetime]` - Orario fine visita
- `duration_minutes: int` - Durata visita in minuti
- `offer_accepted: bool` - Flag accettazione offerta
- `estimated_spending: float` - Spesa stimata
- `user_satisfaction: int` - Soddisfazione utente (1-10)
- `day_of_week: Optional[int]` - Giorno della settimana
- `hour_of_day: Optional[int]` - Ora del giorno
- `weather_condition: str` - Condizioni meteorologiche
- `user_age: int` - Età utente
- `user_profession: str` - Professione utente

- `user_interests: str` - Interessi utente
- `shop_name: str` - Nome del negozio
- `shop_category: str` - Categoria del negozio

Costruttori

- `__init__()` - Inizializza record visita
- `__post_init__()` - Calcola campi derivati (giorno settimana, ora, durata)

Metodi

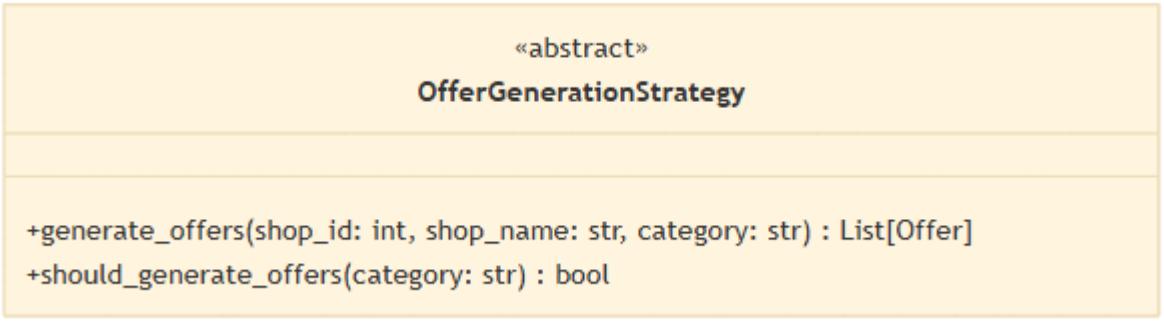
- `to_dict() -> Dict[str, Any]` - Converte in dictionary per inserimento DB

IV - 4. Servizi Business Logic

Questa sezione rappresenta le classi che implementano la logica di business con pattern Strategy$_G$.

IV - 4.1. Struttura delle classi

IV - 4.1.1. OfferGenerationStrategy



Attributi

Classe astratta, nessun attributo specifico.

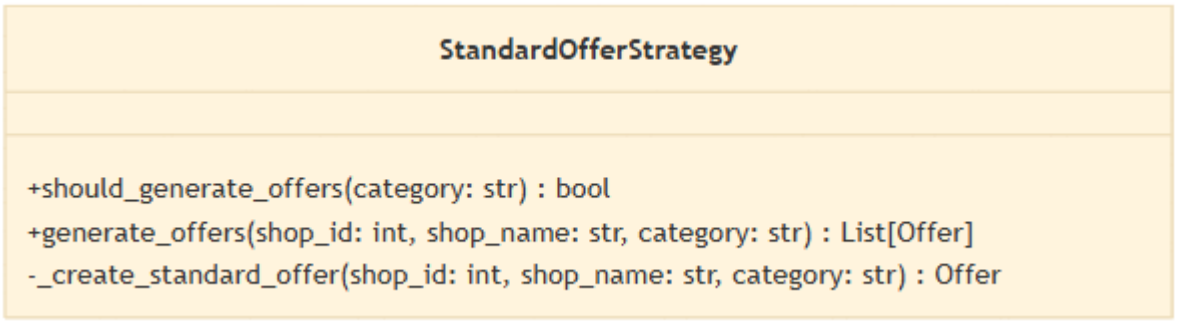
Costruttori

Classe astratta, costruttori implementati dalle sottoclassi.

Metodi

- `generate_offers(shop_id: int, shop_name: str, category: str) -> List[Offer]` - Metodo astratto per generare offerte
- `should_generate_offers(category: str) -> bool` - Metodo astratto per decidere se generare offerte

IV - 4.1.2. StandardOfferStrategy



Attributi

Nessun attributo di istanza.

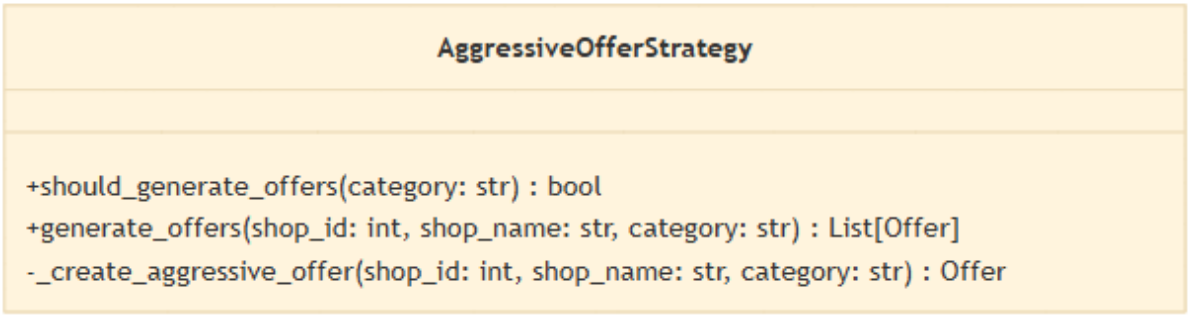
Costruttori

- `__init__()` - Costruttore standard

Metodi

- `should_generate_offers(category: str) -> bool` - Verifica probabilità generazione per categoria
- `generate_offers(shop_id: int, shop_name: str, category: str) -> List[Offer]` - Genera offerte standard randomizzate
- `_create_standard_offer(shop_id: int, shop_name: str, category: str) -> Optional[Offer]` - Crea singola offerta standard

IV - 4.1.3. AggressiveOfferStrategy



Attributi

Nessun attributo di istanza.

Costruttori

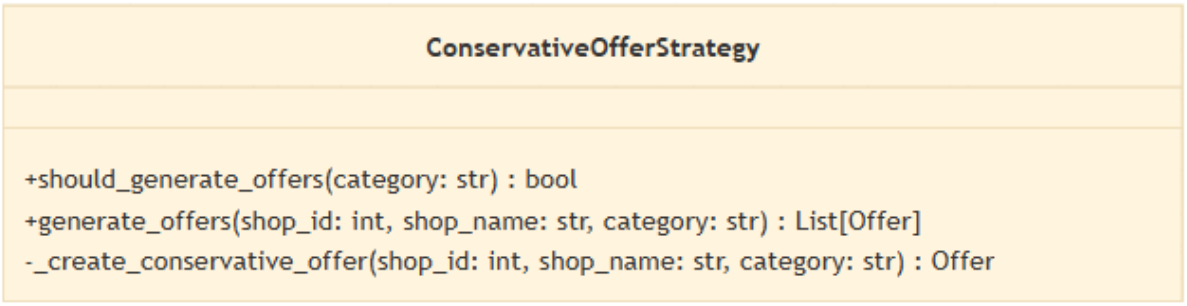
- `__init__()` - Costruttore standard

Metodi

- `should_generate_offers(category: str) -> bool` - Sempre True per strategia aggressiva

- `generate_offers(shop_id: int, shop_name: str, category: str) -> List[Offer]` - Genera più offerte con sconti maggiori
- `_create_aggressive_offer(shop_id: int, shop_name: str, category: str) -> Optional[Offer]` - Crea offerta con sconti potenziati

IV - 4.1.4. ConservativeOfferStrategy



Attributi

Nessun attributo di istanza.

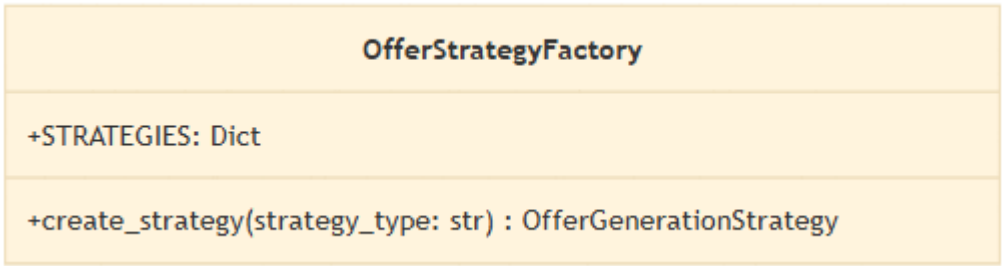
Costruttori

- `__init__()` - Costruttore standard

Metodi

- `should_generate_offers(category: str) -> bool` - Probabilità ridotta rispetto a standard
- `generate_offers(shop_id: int, shop_name: str, category: str) -> List[Offer]` - Genera meno offerte con sconti moderati
- `_create_conservative_offer(shop_id: int, shop_name: str, category: str) -> Optional[Offer]` - Crea offerta con approccio conservativo

IV - 4.1.5. OfferStrategyFactory



Attributi

- `STRATEGIES: Dict` - Dizionario mapping tipi strategia a classi

Costruttori

Classe con metodi di classe, nessun costruttore di istanza.

Metodi

- `create_strategy(strategy_type: str) -> OfferGenerationStrategy` - Crea istanza strategia basata su tipo

IV - 4.1.6. OffersService

Attributi

- `postgres_config: Dict[str, Any]` - Configurazione connessione PostgreSQL
- `strategy: OfferGenerationStrategy` - Strategia di generazione offerte attiva

Costruttori

- `__init__(postgres_config: Dict[str, Any], strategy_type: str)` - Inizializza servizio con configurazione e strategia

Metodi

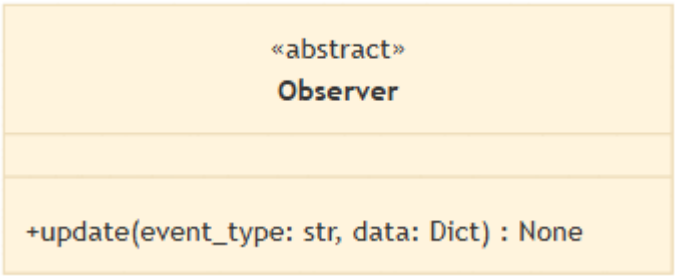
- `set_strategy(strategy: OfferGenerationStrategy) -> None` - Cambia strategia a runtime
- `get_connection()` - Ottiene connessione PostgreSQL
- `generate_offers_for_all_shops() -> int` - Genera offerte per tutti i negozi
- `insert_offers(offers: List[Offer]) -> int` - Inserisce offerte nel database
- `get_active_offers_for_shop(shop_id: int) -> List[Dict[str, Any]]` - Recupera offerte attive per negozio
- `cleanup_expired_offers() -> int` - Disattiva offerte scadute

IV - 5. Data Pipeline

Questa sezione rappresenta le classi per la gestione della pipeline dati con pattern Observer$_G$.

IV - 5.1. Struttura delle classi

IV - 5.1.1. Observer



Attributi

Classe astratta, nessun attributo specifico.

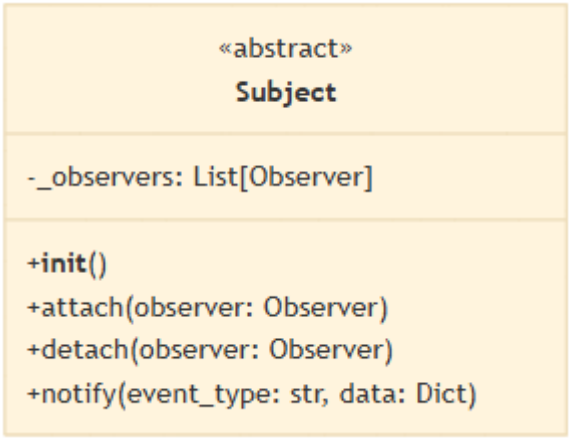
Costruttori

Classe astratta, costruttori implementati dalle sottoclassi.

Metodi

- `update(event_type: str, data: Dict[str, Any]) -> None` - Metodo astratto per ricevere notifiche

IV - 5.1.2. Subject



Attributi

- `_observers: List[Observer]` - Lista degli observer registrati

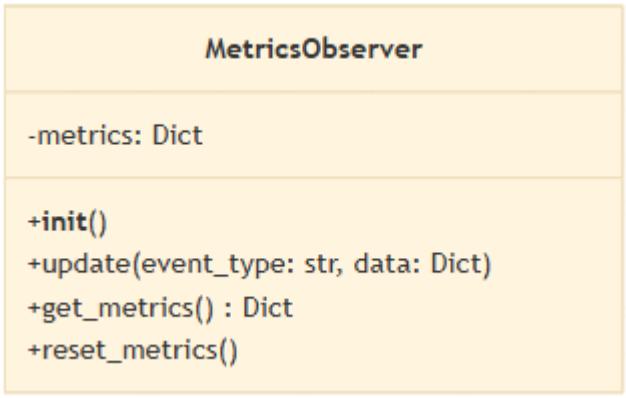
Costruttori

- `__init__()` - Inizializza lista observer vuota

Metodi

- `attach(observer: Observer) -> None` - Registra un observer
- `detach(observer: Observer) -> None` - Rimuove un observer
- `notify(event_type: str, data: Dict[str, Any]) -> None` - Notifica tutti gli observer

IV - 5.1.3. MetricsObserver



Attributi

- `metrics: Dict[str, int]` - Dizionario con contatori delle metriche

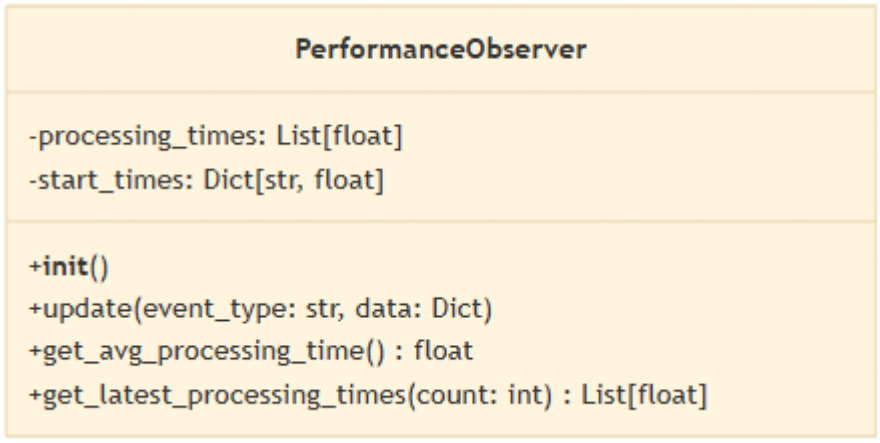
Costruttori

- `__init__()` - Inizializza contatori a zero

Metodi

- `update(event_type: str, data: Dict[str, Any]) -> None` - Aggiorna metriche in base al tipo evento
- `get_metrics() -> Dict[str, int]` - Restituisce copia delle metriche attuali
- `reset_metrics() -> None` - Resetta tutti i contatori

IV - 5.1.4. PerformanceObserver



Attributi

- `processing_times: List[float]` - Lista dei tempi di elaborazione
- `start_times: Dict[str, float]` - Dizionario con timestamp di inizio per evento

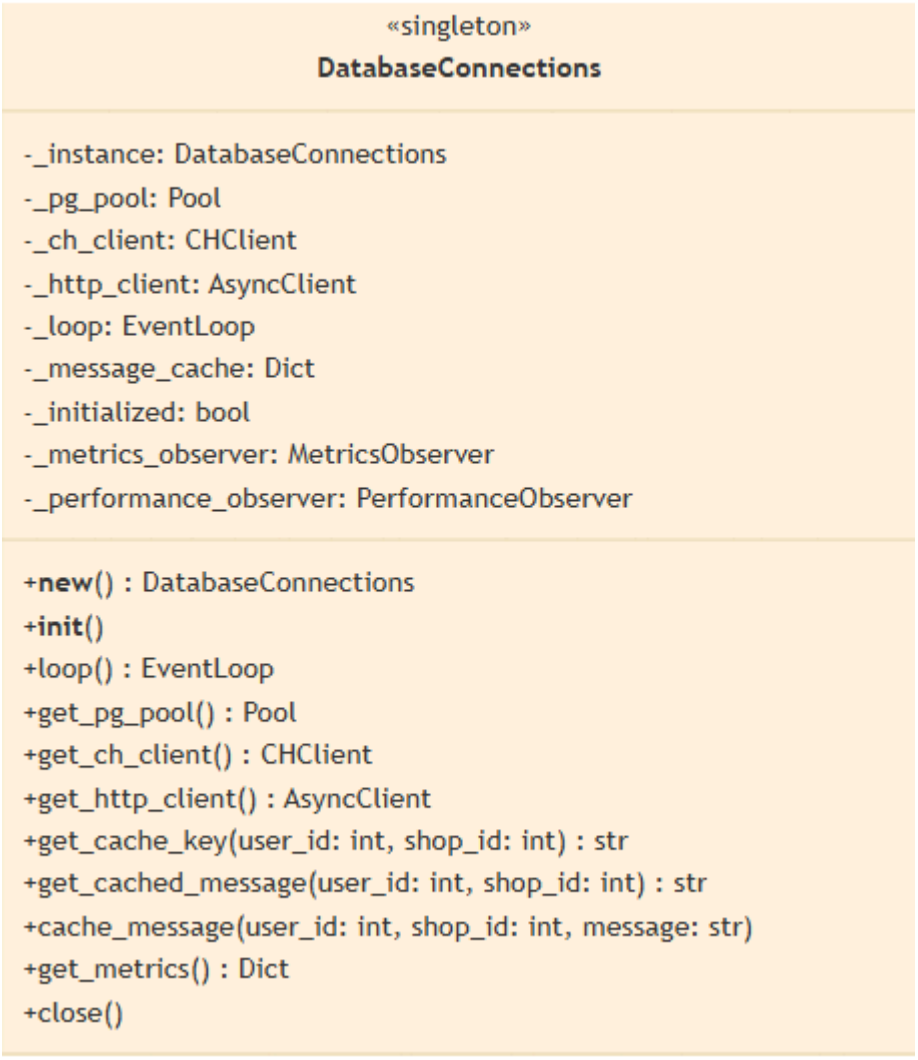
Costruttori

- `__init__()` - Inizializza liste vuote

Metodi

- `update(event_type: str, data: Dict[str, Any]) -> None` - Traccia tempi di inizio/fine elaborazione
- `get_avg_processing_time() -> float` - Calcola tempo medio di elaborazione
- `get_latest_processing_times(count: int) -> List[float]` - Restituisce ultimi N tempi

IV - 5.1.5. DatabaseConnections



Attributi

- `_instance: Optional[DatabaseConnections]` - Istanza singleton
- `_pg_pool: asyncpg.Pool` - Pool di connessioni PostgreSQL
- `_ch_client: CHClient` - Client ClickHouse
- `_http_client: httpx.AsyncClient` - Client HTTP asincrono
- `_loop: asyncio.EventLoop` - Event loop asincrono
- `_message_cache: Dict` - Cache messaggi in-memory
- `_initialized: bool` - Flag inizializzazione
- `_metrics_observer: MetricsObserver` - Observer per metriche
- `_performance_observer: PerformanceObserver` - Observer per performance

Costruttori

- `__new__() -> DatabaseConnections` - Implementa pattern Singleton
- `__init__()` - Inizializza connessioni e observer

Metodi

- `loop() -> EventLoop` - Ottiene o crea event loop
- `get_pg_pool() -> asyncpg.Pool` - Ottiene pool PostgreSQL (lazy init)
- `get_ch_client() -> CHClient` - Ottiene client ClickHouse (lazy init)
- `get_http_client() -> httpx.AsyncClient` - Ottiene client HTTP (lazy init)
- `get_cache_key(user_id: int, shop_id: int) -> str` - Genera chiave cache
- `get_cached_message(user_id: int, shop_id: int) -> Optional[str]` - Recupera messaggio da cache
- `cache_message(user_id: int, shop_id: int, message: str) -> None` - Salva messaggio in cache
- `get_metrics() -> Dict[str, Any]` - Ottiene metriche da observer
- `close()` - Chiude tutte le connessioni

IV - 6. Utilità

Questa sezione rappresenta le classi di supporto e utility del sistema.

IV - 6.1. Struttura delle classi

IV - 6.1.1. Utilità Database

Funzioni helper per gestione database:

- `wait_for_clickhouse_database(client: Client, db_name: str, timeout: int, max_retries: int) -> bool` - Attende disponibilità database ClickHouse
- `wait_for_broker(host: str, port: int, timeout: int) -> None` - Attende disponibilità broker Kafka

IV - 6.1.2. Configurazione Logging

Funzioni per configurazione sistema di logging:

- `setup_logging(log_level: Optional[str])` - Configura logging con formato JSON o text

IV - 6.1.3. Metriche FastAPI

Utilità per integrazione metriche Prometheus:

- `setup_metrics(app: FastAPI, app_name: Optional[str]) -> None` - Configura instrumentazione Prometheus per FastAPI

IV - 6.1.4. Operatori Pipeline

Funzioni per elaborazione stream dati Bytewax:

- `parse_kafka_message(kafka_msg: KafkaSourceMessage) -> Tuple[str, Dict[str, Any]]` - Parsa messaggi Kafka
- `validate_message(parsed_data: Tuple[str, Dict[str, Any]]) -> bool` - Valida messaggi parsati

- `enrich_with_nearest_shop(item: Tuple[str, Dict]) -> List[Tuple[str, Dict]]` - Arricchisce con negozio più vicino
- `check_proximity_and_generate_message(item: Tuple[str, Dict]) -> List[Tuple[str, Dict]]` - Genera messaggio se in prossimità
- `write_to_clickhouse(item: Tuple[str, Dict]) -> None` - Scrive evento in ClickHouse

IV - 7. Design Patterns

IV - 7.1. Introduzione

Il progetto NearYou implementa un sistema complesso di raccomandazioni geo-localizzate che utilizza diversi design patterns per garantire modularità, estensibilità e manutenibilità del codice. L'architettura del sistema sfrutta quattro principali design patterns:

- **Singleton Pattern**$_G$: Per la gestione centralizzata di configurazioni e connessioni
- **Factory Pattern**$_G$: Per la creazione flessibile di cache e strategie
- **Strategy Pattern**: Per algoritmi intercambiabili di generazione offerte
- **Observer Pattern**$_G$: Per il monitoraggio real-time del sistema

Questi patterns lavorano insieme per fornire un'architettura robusta e scalabile per un sistema di streaming real-time che processa eventi GPS, genera raccomandazioni personalizzate e monitora le performance del sistema.

IV - 7.2. Singleton Pattern

IV - 7.2.1. Panoramica

Il Singleton Pattern è implementato in tre componenti chiave del sistema per garantire che esistano singole istanze condivise di risorse critiche:

1. `ConfigurationManager` : Gestione centralizzata della configurazione
2. `CacheManager` : Gestione unificata del sistema di cache
3. `DatabaseConnections` : Gestione delle connessioni ai database

IV - 7.2.2. ConfigurationManager

Implementazione

```
class ConfigurationManager:
    """
    Singleton pattern implementation for configuration management.
    Ensures that only one configuration instance exists throughout the
application.
    """
    _instance: Optional['ConfigurationManager'] = None
    _initialized = False

    def __new__(cls) -> 'ConfigurationManager':
        if cls._instance is None:
            cls._instance = super().__new__(cls)
```

```
        return cls._instance

    def __init__(self):
        # Prevent re-initialization
        if ConfigurationManager._initialized:
            return

        ConfigurationManager._initialized = True

        # Initialize all configuration values
        self._load_environment_config()

        # Validate critical configs in production/staging
        if self.environment in ["production", "staging"]:
            self.validate_critical_configs()
```

Caratteristiche Avanzate

Meccanismo di Implementazione

- **Thread-Safety**: Implementazione thread-safe attraverso l'override del metodo `__new__`
- **Lazy Initialization**$_G$: La configurazione viene caricata solo al primo accesso
- **Prevenzione Re-inizializzazione**: Flag `_initialized` impedisce multiple inizializzazioni
- **Validazione Ambiente**: Controlli specifici per ambienti production/staging

Vantaggi Architetturali

- **Consistenza Globale**: Tutti i componenti accedono alla stessa configurazione
- **Performance**: Elimina la necessità di rilegger e file di configurazione
- **Memory Efficiency**: Una sola istanza in memoria per tutta l'applicazione
- **Centralizzazione**: Punto unico di accesso per tutte le configurazioni

IV - 7.2.3. CacheManager

Implementazione

```
class CacheManager:
    """
    Singleton pattern implementation for cache management.
    Ensures single cache instance throughout the application.
    """
    _instance: Optional['CacheManager'] = None
    _cache: Optional[CacheInterface] = None

    def __new__(cls) -> 'CacheManager':
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance
```

```
    def initialize(self, cache_type: str = "auto", **config) -> None:
        """Initialize cache with given configuration."""
        if self._cache is None:
            self._cache = CacheFactory.create_cache(cache_type, **config)
            logger.info(f"Cache manager initialized with
{type(self._cache).__name__}")

    def get_cache(self) -> CacheInterface:
        """Get the cache instance."""
        if self._cache is None:
            raise RuntimeError("Cache not initialized. Call initialize()
first.")
        return self._cache
```

Integrazione con Factory Pattern

Il `CacheManager` utilizza il `CacheFactory` per creare l'istanza di cache appropriata, dimostrando come i design patterns scelti possano collaborare efficacemente:

```
# Utilizzo integrato
cache_manager = CacheManager()  # Singleton
cache_manager.initialize("auto", **config)  # Factory Pattern
cache = cache_manager.get_cache()  # Accesso unificato
```

IV - 7.2.4. DatabaseConnections

Implementazione con Observer Pattern

```
class DatabaseConnections(Subject):
    """
    Singleton pattern for database connections management.
    Also implements Subject for Observer pattern.
    """
    _instance: Optional['DatabaseConnections'] = None

    def __new__(cls) -> 'DatabaseConnections':
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            # Initialize Subject
            super(DatabaseConnections, cls._instance).__init__()
            # Initialize default observers
            cls._instance._setup_default_observers()
        return cls._instance
```

Questa implementazione combina Singleton e Observer patterns, permettendo monitoring centralizzato delle operazioni database.

IV - 7.3. Factory Pattern

IV - 7.3.1. Panoramica

Il Factory Pattern è implementato principalmente attraverso due factory classes:

1. `CacheFactory` : Creazione di istanze cache (Redis/Memory/Auto)
2. `OfferStrategyFactory` : Creazione di strategie di generazione offerte

IV - 7.3.2. CacheFactory

Implementazione Completa

```
class CacheFactory:
    """
    Factory pattern implementation for cache creation.
    Creates appropriate cache instances based on configuration.
    """

    @staticmethod
    def create_cache(cache_type: str = "auto", **config) -> CacheInterface:
        """Create cache instance based on type and configuration."""
        if cache_type == "memory":
            return CacheFactory._create_memory_cache(**config)
        elif cache_type == "redis":
            return CacheFactory._create_redis_cache(**config)
        elif cache_type == "auto":
            return CacheFactory._create_auto_cache(**config)
        else:
            raise ValueError(f"Unknown cache type: {cache_type}")

    @staticmethod
    def _create_auto_cache(**config) -> CacheInterface:
        """Automatically choose cache type based on availability."""
        try:
            redis_cache = CacheFactory._create_redis_cache(**config)
            # Test connection
            redis_cache.set("__test__", "connection_test", ttl=1)
            redis_cache.delete("__test__")
            logger.info("Auto-selected RedisCache (connection successful)")
            return redis_cache
        except Exception as e:
            logger.warning(f"Redis not available ({e}), falling back to
MemoryCache")
            return CacheFactory._create_memory_cache(**config)
```

Caratteristiche Avanzate

Modalità Auto-Selection La modalità «auto» del `CacheFactory` implementa un meccanismo intelligente di fallback[G]:

1. **Tentativo Redis**: Prova prima a creare e testare una connessione Redis

2. **Test Connessione**: Esegue operazioni di test per verificare la disponibilità
3. **Fallback Automatico**: In caso di errore, usa automaticamente MemoryCache
4. **Logging Intelligente**: Registra le decisioni per debugging e monitoring

IV - 7.3.3. OfferStrategyFactory

Implementazione

```
class OfferStrategyFactory:
    """Factory for creating different offer generation strategies."""

    @staticmethod
    def create_strategy(strategy_type: str) -> OfferGenerationStrategy:
        """Create strategy instance based on type."""
        strategy_map = {
            "standard": StandardOfferStrategy,
            "aggressive": AggressiveOfferStrategy,
            "conservative": ConservativeOfferStrategy
        }

        strategy_class = strategy_map.get(strategy_type.lower())
        if not strategy_class:
            available = list(strategy_map.keys())
            raise ValueError(f"Unknown strategy type: {strategy_type}.
Available: {available}")

        return strategy_class()
```

Estensibilità

Il design permette facile aggiunta di nuove strategie:

```
# Aggiungere una nuova strategia
class PremiumOfferStrategy(OfferGenerationStrategy):
    # Implementazione strategia premium...
    pass

# Registrazione nella factory
strategy_map["premium"] = PremiumOfferStrategy
```

IV - 7.4. Strategy Pattern

IV - 7.4.1. Panoramica

Il Strategy Pattern è implementato per la generazione delle offerte commerciali, permettendo algoritmi intercambiabili per diverse strategie di business.

IV - 7.4.2. Architettura delle Strategie

Interfaccia Base

```
class OfferGenerationStrategy(ABC):
    """Abstract base class for offer generation strategies."""

    @abstractmethod
    def generate_offers(self, shop_id: int, shop_name: str, category: str)
-> List[Offer]:
        """Generate offers for a specific shop."""
        pass

    @abstractmethod
    def should_generate_offers(self, category: str) -> bool:
        """Determine if offers should be generated for this category."""
        pass
```

IV - 7.4.3. Implementazioni Concrete

StandardOfferStrategy

Strategia Bilanciata

- **Sconti**: 5-25% (bilanciato)
- **Durata**: 3-14 giorni (media durata)
- **Probabilità Generazione**: Basata su configurazione categoria
- **Targeting**: Età casuale (30% probabilità)

```
class StandardOfferStrategy(OfferGenerationStrategy):
    """Standard offer generation strategy with randomized parameters."""

    def should_generate_offers(self, category: str) -> bool:
        probability = CATEGORY_OFFER_PROBABILITY.get(category.lower(), 0.5)
        should_generate = random.random() <= probability
        return should_generate

    def generate_offers(self, shop_id: int, shop_name: str, category: str)
-> List[Offer]:
        if not self.should_generate_offers(category):
            return []

        num_offers = random.randint(MIN_OFFERS_PER_SHOP,
MAX_OFFERS_PER_SHOP)
        offers = []

        for i in range(num_offers):
            offer = self._create_standard_offer(shop_id, shop_name,
category)
            if offer:
```

```
                offers.append(offer)

        return offers
```

AggressiveOfferStrategy

Strategia Aggressiva

- **Sconti**: 15-50% (sconti elevati)
- **Durata**: 1-7 giorni (breve durata, urgenza)
- **Probabilità Generazione**: Sempre genera (100%)
- **Caratteristiche**: Massimizza conversioni immediate

```
class AggressiveOfferStrategy(OfferGenerationStrategy):
    """Aggressive strategy with high discounts and short duration."""

    def should_generate_offers(self, category: str) -> bool:
        return True  # Always generate offers in aggressive mode

    def _get_aggressive_discount_range(self, category: str) -> Tuple[int,
int]:
        """Get higher discount ranges for aggressive strategy."""
        base_range = CATEGORY_DISCOUNT_RANGES.get(category.lower(),
DEFAULT_DISCOUNT_RANGE)
        # Boost discounts by 10-25 percentage points
        min_discount = min(50, max(15, base_range[0] + 10))
        max_discount = min(50, base_range[1] + 25)
        return (min_discount, max_discount)
```

IV - 7.4.4. Utilizzo Runtime

Switching Dinamico

```
class OffersService:
    """Service for managing offers with strategy pattern."""

    def __init__(self, postgres_config: Dict[str, Any], strategy_type: str
= "standard"):
        self.postgres_config = postgres_config
        self.strategy = OfferStrategyFactory.create_strategy(strategy_type)

    def set_strategy(self, strategy: OfferGenerationStrategy) -> None:
        """Change strategy at runtime."""
        self.strategy = strategy
        logger.info(f"Strategy changed to: {type(strategy).__name__}")

    def generate_shop_offers(self, shop_id: int, shop_name: str, category:
str) -> List[Offer]:
```

```
        """Generate offers using current strategy."""
        return self.strategy.generate_offers(shop_id, shop_name, category)
```

IV - 7.5. Observer Pattern

IV - 7.5.1. Panoramica

Il Observer Pattern è implementato per monitorare eventi del sistema in tempo reale, permettendo raccolta di metriche e monitoring delle performance.

IV - 7.5.2. Architettura Observer

Subject Base

```
class Subject(ABC):
    """Abstract subject that observers can subscribe to."""

    def __init__(self):
        self._observers: List[Observer] = []

    def attach(self, observer: Observer) -> None:
        """Attach an observer."""
        if observer not in self._observers:
            self._observers.append(observer)
            logger.debug(f"Attached observer: {type(observer).__name__}")

    def detach(self, observer: Observer) -> None:
        """Detach an observer."""
        if observer in self._observers:
            self._observers.remove(observer)
            logger.debug(f"Detached observer: {type(observer).__name__}")

    def notify(self, event_type: str, data: Dict[str, Any]) -> None:
        """Notify all observers."""
        for observer in self._observers:
            try:
                observer.update(event_type, data)
            except Exception as e:
                logger.error(f"Error notifying observer
{type(observer).__name__}: {e}")
```

IV - 7.5.3. Implementazioni Observer

MetricsObserver

Raccolta Metriche Operative Monitora eventi operativi del sistema:

- Eventi processati
- Messaggi generati
- Visite simulate
- Errori di sistema

- Cache hits/misses
- Negozi trovati

```python
class MetricsObserver(Observer):
    """Observer for collecting metrics."""

    def __init__(self):
        self.metrics = {
            "events_processed": 0,
            "messages_generated": 0,
            "visits_simulated": 0,
            "errors": 0,
            "shops_found": 0,
            "cache_hits": 0,
            "cache_misses": 0
        }

    def update(self, event_type: str, data: Dict[str, Any]) -> None:
        """Update metrics based on event."""
        if event_type == "event_processed":
            self.metrics["events_processed"] += 1
        elif event_type == "message_generated":
            self.metrics["messages_generated"] += 1
        elif event_type == "visit_simulated":
            self.metrics["visits_simulated"] += 1
        # ... altri eventi

    def get_metrics(self) -> Dict[str, int]:
        """Get current metrics snapshot."""
        return self.metrics.copy()
```

PerformanceObserver

Monitoring Performance Traccia performance del sistema:

- Tempi di processing eventi
- Latenza operazioni database
- Throughput del sistema
- Analisi trend temporali

```python
class PerformanceObserver(Observer):
    """Observer for monitoring performance metrics."""

    def __init__(self):
        self.processing_times = []
        self.active_processes = {}

    def update(self, event_type: str, data: Dict[str, Any]) -> None:
```

```
        """Track processing performance."""
        if event_type == "processing_start":
            event_id = data.get("event_id")
            if event_id:
                self.active_processes[event_id] = time.time()

        elif event_type == "processing_end":
            event_id = data.get("event_id")
            if event_id and event_id in self.active_processes:
                start_time = self.active_processes.pop(event_id)
                duration = time.time() - start_time
                self.processing_times.append(duration)

                # Keep only last 1000 measurements
                if len(self.processing_times) > 1000:
                    self.processing_times = self.processing_times[-1000:]
```

IV - 7.5.4. DatabaseConnections come Subject

Implementazione Integrata

```
class DatabaseConnections(Subject):
    """Database connections manager that also acts as event subject."""

    def __init__(self):
        super().__init__()
        self._setup_default_observers()

    def _setup_default_observers(self):
        """Setup default system observers."""
        self.metrics_observer = MetricsObserver()
        self.performance_observer = PerformanceObserver()

        self.attach(self.metrics_observer)
        self.attach(self.performance_observer)

        logger.info("Default observers attached to DatabaseConnections")

    def get_metrics(self) -> Dict[str, Any]:
        """Get comprehensive system metrics."""
        return {
            "metrics": self.metrics_observer.get_metrics(),
            "performance": {
                "avg_processing_time":
self.performance_observer.get_avg_processing_time(),
                "total_measurements":
len(self.performance_observer.processing_times)
            }
        }
```

IV - 8. Tabella dei Requisiti - Stato di Implementazione

ID	Descrizione	Stato
REQUISITI FUNZIONALI		
RF1.1	Il sistema deve supportare autenticazione JWT$_G$ con username/password via endpoint `/api/token`	Soddisfatto
RF1.2	I token$_G$ devono avere scadenza configurabile tramite `JWT_EXPIRATION_S` (default 1 ora)	Soddisfatto
RF1.3	Le credenziali devono essere validate contro database ClickHouse con hash$_G$ sicuri	Soddisfatto
RF1.4	Il sistema deve supportare logout con invalidazione sessione client-side	Soddisfatto
RF1.5	WebSocket deve autenticare via token JWT per connessioni real-time	Soddisfatto
RF2.1	Il sistema deve simulare movimenti utenti con `producer.py` su percorsi Milano OSRM	Soddisfatto
RF2.2	Gli eventi GPS devono essere prodotti in Kafka topic `gps_stream` con SSL	Soddisfatto
RF2.3	I percorsi devono essere calcolati usando OSRM self-hosted con profilo cycling	Soddisfatto
RF2.4	Gli eventi devono contenere: user_id, latitude, longitude, timestamp, poi_info	Soddisfatto
RF2.5	Il producer deve supportare readiness checks per Kafka, ClickHouse e OSRM	Soddisfatto
RF3.1	Il sistema deve processare eventi GPS via Bytewax dataflow con Observer pattern	Soddisfatto
RF3.2	Deve calcolare distanza usando query PostGIS `ST_DWithin` per soglia 200m	Soddisfatto
RF3.3	Deve implementare ConnectionManager singleton per pooling database	Soddisfatto
RF3.4	Deve prevenire messaggi duplicati con cache visit tracking	Soddisfatto
RF3.5	Deve supportare metriche real-time con PerformanceObserver	Soddisfatto
RF4.1	Il sistema deve generare messaggi via HTTP service `/generate-message`	Soddisfatto
RF4.2	Deve supportare provider LLM configurabili (Groq/OpenAI) via `LLM_PROVIDER`	Soddisfatto
RF4.3	Deve implementare cache Redis per messaggi con TTL configurabile	Soddisfatto
RF4.4	Deve gestire offerte con Strategy Pattern (Standard/Aggressive/Conservative)	Soddisfatto
RF4.5	Deve supportare Builder Pattern per creazione offerte complesse	Soddisfatto
RF4.6	Factory Pattern deve creare offerte tipizzate (Flash/Student/Senior/Category)	Soddisfatto
RF4.7	Validation Strategy deve validare vincoli offerte (età, interessi, date)	Soddisfatto
RF5.1	Deve servire interfaccia web tramite `/dashboard/user` con static files	Soddisfatto
RF5.2	Deve implementare mappa Leaflet con marker categorizzati per shop types	Soddisfatto
RF5.3	Deve visualizzare percorso utente come polyline con history	Soddisfatto
RF5.4	Deve supportare filtri categoria con mapping predefinito	Soddisfatto
RF5.5	Deve implementare fallback WebSocket→HTTP polling automatico	Soddisfatto
RF6.1	Deve memorizzare eventi in ClickHouse tabelle `user_events` e `user_visits`	Soddisfatto
RF6.2	Deve gestire shop data in PostgreSQL/PostGIS con indici spaziali	Soddisfatto
RF6.3	Deve mantenere profili utenti in ClickHouse tabella `users`	Soddisfatto
RF6.4	Deve tracciare storico visite con vista materializzata `mv_daily_shop_stats`	Soddisfatto
RF6.5	Deve supportare offers storage in PostgreSQL con vincoli temporali	Soddisfatto
RF7.1	Deve implementare Redis cache per messaggi LLM con serializzazione JSON	Soddisfatto
RF7.2	Deve supportare Memory cache con LRU$_G$ eviction come fallback	Soddisfatto
RF7.3	Deve implementare TTL configurabile via `CACHE_TTL`	Soddisfatto
RF7.4	Deve fornire cache statistics e hit rate monitoring	Soddisfatto
REQUISITI FUNZIONALI DESIDERABILI		
RFD1.1	Il sistema dovrebbe implementare lazy loading notifiche con IntersectionObserver$_G$	Soddisfatto
RFD1.2	Dovrebbe supportare tema scuro/chiaro configurabile dall'utente con localStorage$_G$	Soddisfatto

ID	Descrizione	Stato
RFD1.3	Dovrebbe fornire local cache frontend per shop areas e notifications	Soddisfatto
RFD1.4	Dovrebbe limitare e ottimizzare rendering markers (max 100 shops)	Soddisfatto
RFD1.5	Dovrebbe implementare design responsivo$_G$ con breakpoint mobile	Soddisfatto
RFD2.1	Il sistema dovrebbe esporre analisi Prometheus tramite FastAPI instrumentator$_G$	Soddisfatto
RFD2.2	Dovrebbe implementare dashboard Grafana con pannelli assemblati dinamicamente	Soddisfatto
RFD2.3	Dovrebbe supportare logging strutturato con livelli configurabili	Soddisfatto
RFD2.4	Dovrebbe includere exporters per database e servizi (Postgres, Redis, ClickHouse)	Soddisfatto
RFD2.5	Dovrebbe supportare Loki/Promtail per log aggregation$_G$	Soddisfatto
RFD2.6	Dovrebbe implementare cAdvisor$_G$ per container monitoring	Soddisfatto
RFD3.1	Il sistema dovrebbe supportare statistiche utente dettagliate per periodi configurabili	Soddisfatto
RFD3.2	Dovrebbe implementare paginazione$_G$ avanzata per notifiche e promozioni	Soddisfatto
RFD3.3	Dovrebbe fornire filtri dinamici Grafana (età, professione, categoria, popolarità)	Soddisfatto
RFD3.4	Dovrebbe supportare mappe interattive con route visualization$_G$	Soddisfatto
RFD3.5	Dovrebbe implementare shop visit simulation con feedback realtime	Soddisfatto
RFD4.1	Il sistema dovrebbe implementare connection pooling$_G$ avanzato	Soddisfatto
RFD4.2	Dovrebbe supportare cache distribuita con clustering$_G$ Redis	Soddisfatto
RFD4.3	Dovrebbe fornire push gateway$_G$ per metriche batch	Soddisfatto
RFD4.4	Dovrebbe implementare graceful degradation$_G$ per service failures	Soddisfatto
REQUISITI FUNZIONALI FACOLTATIVI		
RFF1.1	Il sistema potrebbe integrare API$_G$ meteo per context-aware messaging	Non Implementato
RFF1.2	Potrebbe supportare payment integration per offer redemption	Non Implementato
RFF2.1	Potrebbe utilizzare reinforcement learning$_G$ per offer optimization	Non Implementato
RFF3.1	Potrebbe supportare A/B testing$_G$ per offer strategies	Non Implementato

Totale requisiti: 72
Soddisfatti: 68
Non implementati: 4 (solo requisiti facoltativi)