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 0.1.0

Indice

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
);
```

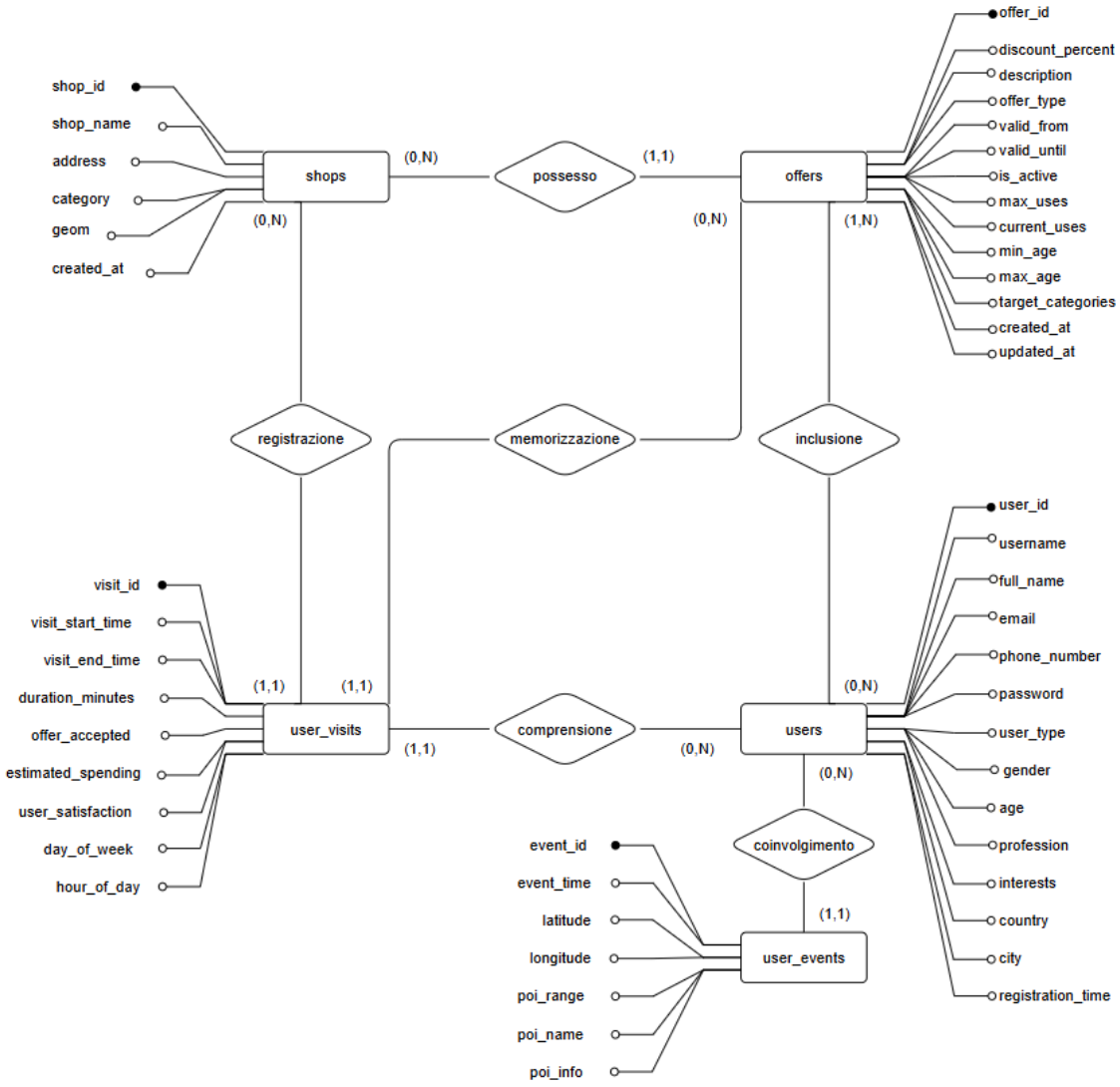
II - 2.5.5. Architettura Database: Schema ClickHouse (Eventi e Analytics)

```
-- 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:

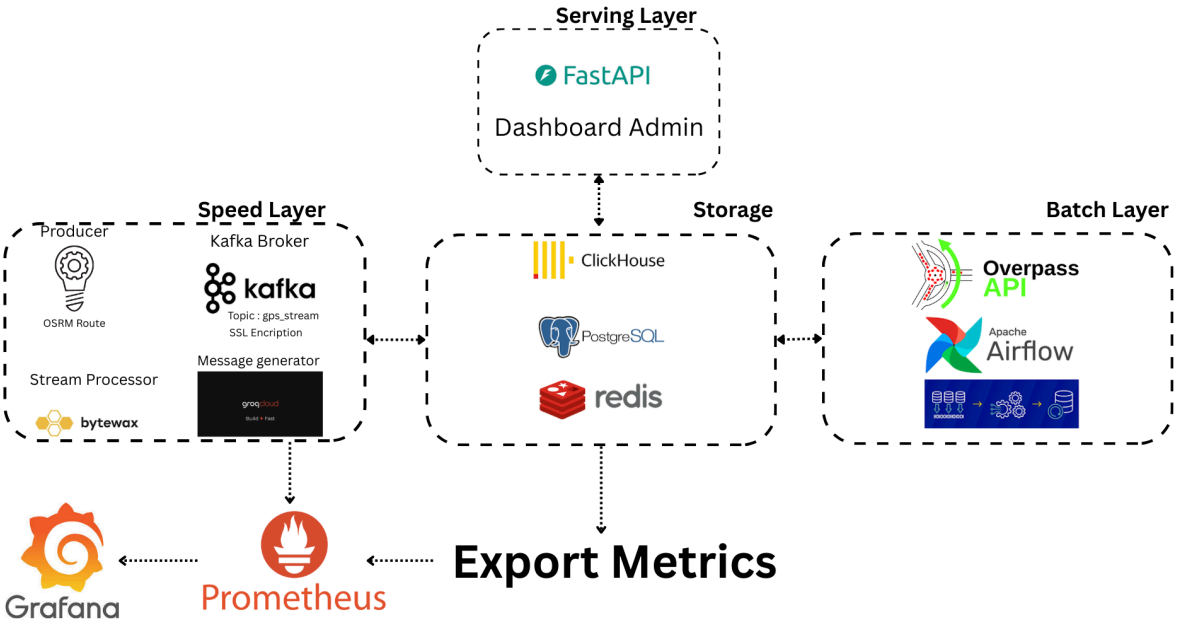
- Latenza sub-secondo preservata per notifiche contestuali.
- Riduzione dei costi (meno messaggi, meno cache invalidata, meno I/O superfluo).
- Ricalcoli batch controllati per qualità storica e correzione drift.
- Evoluzione agile: nuove feature introdotte prima nel batch, poi eventualmente ottimizzate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

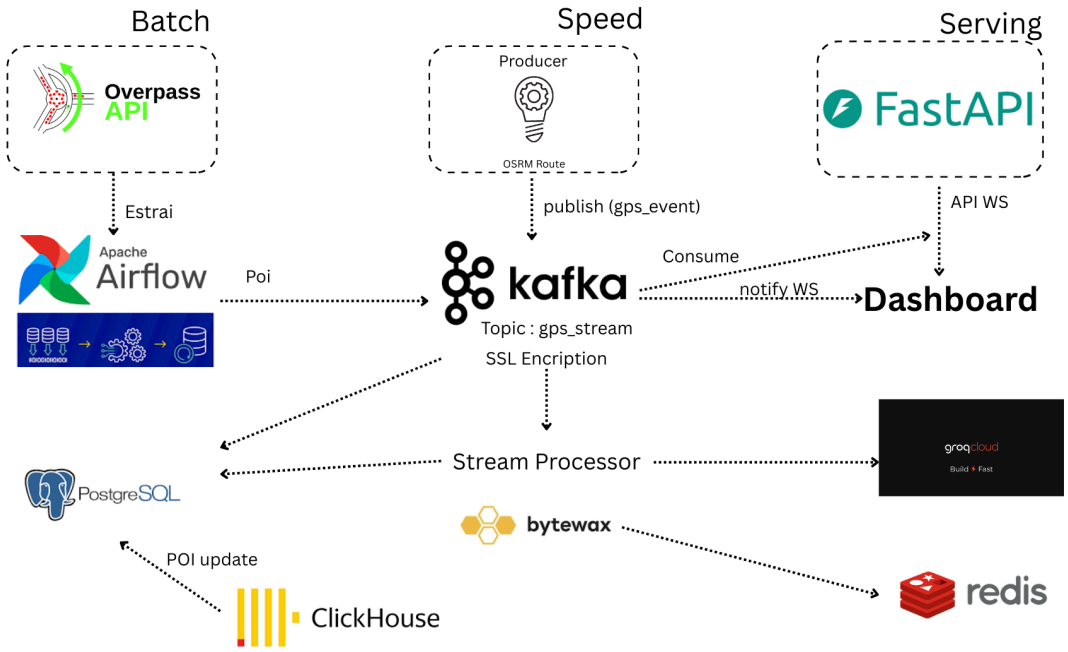
Compone viste logiche e servizi:

- 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.