Università di Catania

Introduzione al Data Mining

A.A. 2025/2026

Corso di Laurea Magistrale in Informatica

Dipartimento di Matematica e Informatica

Relazione Progetto

eTranslator: A Multi-Agent SQL Parser

Torrisi Giovanni – 1000015103

Indice

1. Introduzione

Negli ultimi anni i Large Language Models (LLM) hanno dimostrato capacità sempre più avanzate nella generazione del testo, aprendo le porte ad un enorme quantità di potenziali casi d'uso. Grazie all'addestramento su grandi quantità di dati testuali, questi modelli sono in grado di catturare le complessità e le sfumature del linguaggio umano e di svolgere una vasta gamma di compiti.

Uno degli ambiti in cui il potenziale degli LLM risulta particolarmente evidente è quello della traduzione automatica del testo da una lingua all'altra. In particolare, anche SQL è un linguaggio, e dunque sarebbe interessante utilizzare questa tecnologia come supporto nell'interrogazione di basi di dati. Tradizionalmente, l'accesso a sistemi di gestione di database richiede la conoscenza di linguaggi formali specifici e una comprensione approfondita dello schema dei dati. Questo rappresenta una barriera significativa per utenti non esperti, limitando la fruibilità delle informazioni disponibili. L'impiego di LLM consente invece di ridurre tale complessità, permettendo agli utenti di formulare domande in linguaggio naturale che possono essere automaticamente tradotte in query eseguibili.

In questo contesto si inserisce il presente progetto. Il problema affrontato consiste nella traduzione automatica di richieste espresse in linguaggio naturale in query SQL eseguibili e che rispettino lo schema del database. L'obiettivo è realizzare un sistema in grado di fungere da intermediario tra l'utente e il database, migliorando l'accessibilità ai dati e riducendo il rischio di errori nelle interrogazioni.

Il progetto combina tecnologie di Data Mining, sistemi di gestione dei dati e orchestrazione di agenti basati su LLM. L'architettura proposta prevede una separazione chiara delle responsabilità, comprendendo una pipeline che coinvolge vari componenti che si occupano di

filtrare richieste fuori dominio, chiarire la richiesta utente, arricchire il contesto semantico, generare SQL, eseguire query, e commentare risultati.

Nei capitoli successivi verranno descritti in dettaglio il problema affrontato, i dati utilizzati, le tecnologie adottate e l'architettura software del sistema.

2. Problema

Nell'ambito aziendale di eVision, società che sviluppa software gestionali per il settore della Grande Distribuzione Organizzata (GDO), l'accesso ai dati riveste un ruolo centrale nei processi decisionali e di analisi.

Lo sviluppo del progetto nasce dall'esigenza di interrogare tali basi di dati riducendo la dipendenza da competenze tecniche avanzate nella scrittura di query SQL. L' utente deve poter formulare delle richieste in linguaggio naturale, che il sistema deve tradurre in query compatibili con lo schema del database aziendale.

Il sistema deve essere in grado di individuare errori sintattici o incompatibilità con lo schema del database. Per questo motivo, la generazione deve essere affiancata da una fase di esecuzione sul database, che consenta di verificare concretamente la correttezza della query prodotta.

Infine, siccome stiamo lavorando su una tabella con molte colonne e l'utente potrebbe non conoscere come sono distribuiti i valori nelle colonne semanticamente simili, il sistema potrebbe non riuscire ad identificare correttamente quale usare per fare le query. Inserire tutti i prodotti all'interno di un database vettoriale consentirebbe di fare

delle ricerche per similarità semantica e quindi rendere la query ancora più selettiva e accurata.

Alla luce di queste considerazioni, il problema può essere sintetizzato nella progettazione di un sistema che, nel contesto dei dati aziendali di eVision, sia in grado di:

- Tradurre richieste in linguaggio naturale in query SQL;
- Validare le query attraverso l'esecuzione sul database;
- Gestire eventuali errori sintattici;
- Restituire all'utente risultati interpretabili.

3. Dati

In eVision, i dati sono memorizzati all'interno di un database ClickHouse, un sistema di gestione di basi di dati colonnare ad alte prestazioni, progettato specificamente per scenari di Online Analytical Processing (OLAP). Tale tecnologia risulta particolarmente adatta alla gestione di grandi volumi di dati e all'esecuzione efficiente di query analitiche complesse.

Il database contiene un'unica tabella principale, denominata 'sales_data' composta da 41 colonne. Questa tabella raccoglie i dati relativi agli scontrini di vendita. In particolare, ogni riga della tabella corrisponde a un singolo prodotto scansionato alla cassa.

Le informazioni presenti nella tabella coprono diversi aspetti rilevanti del processo di vendita. Sono inclusi dati relativi al punto vendita in cui è avvenuta la transazione, come l'identificativo del negozio e la località, nonché informazioni temporali come data e ora, che permettono di analizzare le vendite in funzione del tempo. La tabella

contiene inoltre numerosi attributi descrittivi del prodotto, tra cui la descrizione, il prezzo, il peso, il codice identificativo, l'eventuale sconto applicato e il fornitore.

Un ulteriore elemento di rilievo è la presenza di informazioni relative alla gerarchia merceologica. La gerarchia merceologica nella GDO funziona tramite una struttura ad albero che parte da macro settori (es. Carni, Bevande, Pescheria) per arrivare a categoria sempre più specifiche (es.Bicchieri, Banane, Latte). Questo consente analisi a diversi livelli di aggregazione.

4. Tecnologie

Il sistema sviluppato si basa su un insieme di tecnologie selezionate con l'obiettivo di garantire quanta più modularità e scalabilità possibile. In questa sezione vengono introdotte le principali tecnologie utilizzate, illustrandone brevemente le motivazioni alla base della scelta e il ruolo ricoperto all'interno del progetto. Ulteriori dettagli implementativi verranno approfonditi nella sezione successiva dedicata all'architettura software.

4.1 Modello LLM

Per la generazione dei contenuti in linguaggio naturale e per la traduzione delle richieste dell'utente in query SQL è stato impiegato un Large Language Model (LLM). Questi modelli linguistici risultano particolarmente efficaci nel gestire task di comprensione semantica e generazione di testi. Nel nostro scenario in cui trattiamo un input non strutturato è perfetto.

Nel progetto, l'LLM è il componente centrale. Svolge i compiti di classificazione e interpretazione delle richieste dell'utente, della generazione query SQL e infine della spiegazione dei risultati ottenuti. Ogni task svolto viene supervisionato da un agente dedicato.

4.2 LangGraph

La libreria LangGraph è stata utilizzata per l'orchestrazione degli agenti LLM. Questa tecnologia consente di modellare il comportamento del sistema come un automa a stati finiti in cui c'è un grafo con nodi, archi e uno stato condiviso. LangGraph è stato scelto in quanto permette di separare in modo chiaro le responsabilità dei diversi agenti, e di implementare anche flussi di esecuzione complessi con cicli e archi condizionali.

4.3 ClickHouse

Nel contesto aziendale di eVision, i dati utilizzati nel progetto sono memorizzati all'interno di un database ClickHouse, che rappresenta la soluzione adottata dall'azienda per la gestione dei dati. Di conseguenza, la scelta di ClickHouse deriva dall'infrastruttura tecnologica già in uso presso l'azienda.

ClickHouse è un DBMS colonnare ad alte prestazioni, progettato per scenari di tipo OLAP, e risulta particolarmente adatto alla gestione di grandi volumi di dati e all'esecuzione efficiente di query analitiche complesse, caratteristiche tipiche del dominio applicativo della Grande Distribuzione Organizzata.

4.4 ClickHouse Connect

Per consentire l'interazione con il database ClickHouse è stata utilizzata la libreria ClickHouse Connect, che fornisce un'interfaccia Python per l'esecuzione di query e la gestione dei risultati.

Nel progetto, ClickHouse Connect viene utilizzata per eseguire le query generate dal sistema e per recuperare i risultati da restituire all'utente, costituendo il collegamento tra l'applicazione e il database.

4.5 Weaviate

Per migliorare la precisione delle query generate e ridurre l'ambiguità nella selezione delle colonne e dei prodotti, è stato utilizzato il database vettoriale Weaviate. Questa tecnologia consente di memorizzare rappresentazioni vettoriali di oggetti per poi effettuare ricerche di similarità semantica basate sul contenuto. In pratica un modello di embedding esterno trasforma ogni oggetto di una collezione Weaviate in un vettore di numeri che viene memorizzato insieme all'oggetto stesso. In seguito, l'utente che vuole effettuare delle interrogazioni al database può utilizzare delle query in linguaggio naturale: il sistema chiamerà di nuovo il modello di embedding esterno per creare un vettore relativo alla richiesta utente e poi quest'ultimo verrà confrontato da un punto di vista matematico con gli altri vettori associati agli oggetti della collezione tramite una misura di distanza tra vettori (tipicamente la distanza del coseno). I concetti correlati avranno una distanza bassa perché il modello di embedding fa in modo che oggetti vicini nello spazio vettoriale si riferiscano a concetti semanticamente correlati.

Nel progetto, Weaviate è stato impiegato per creare una collezione di oggetti rappresentativi dei prodotti venduti dai vari supermercati.

4.6 Streamlit

Per la realizzazione dell'interfaccia utente è stata utilizzata la libreria Streamlit, un framework Python orientato allo sviluppo rapido di applicazioni web interattive. Streamlit è stato scelto per la sua capacità di integrare componenti grafici direttamente dal codice applicativo, senza la necessità di dover sviluppare per intero una tradizionale infrastruttura front-end dedicata.

Nel progetto, Streamlit è stato utilizzato per creare un'interfaccia grafica che consente all'utente di inserire richieste in linguaggio naturale e di visualizzare i risultati prodotti dal sistema in seguito all'esecuzione delle query sul database. L'interfaccia funge da punto di accesso al sistema, facilitando l'interazione con le componenti sottostanti e rendendo il funzionamento del progetto facilmente dimostrabile anche a utenti non tecnici.

5. Requisiti Tecnici di Infrastruttura

Il presente capitolo rappresenta un approfondimento della sezione *Requisiti* descritta all'interno del file reame.md del progetto e ha l'obiettivo di chiarire in modo più dettagliato le dipendenze tecniche e infrastrutturali necessarie per l'esecuzione corretta del software. Trattandosi di un sistema che integra modelli linguistici, database analitici e componenti di ricerca semantica, è fondamentale disporre di un ambiente adeguatamente configurato.

Innanzitutto è necessario avere Python installato sul sistema, in quanto l'intera pipeline applicativa è stata sviluppata utilizzando questo linguaggio. Oltre all'ambiente di esecuzione, è indispensabile disporre delle credenziali di accesso a un database ClickHouse, comprensive di host, porta, username e password. Questo requisito è particolarmente rilevante poiché il software si basa su dati e su un contesto aziendale specifico: per ottenere risultati coerenti è necessario avere accesso allo stesso database utilizzato in fase di sviluppo oppure, in alternativa, a un database che presenti una struttura equivalente, sia in termini di schema che di semantica delle colonne. In assenza di questa condizione, il sistema potrebbe non essere in grado di generare query SQL adeguate.

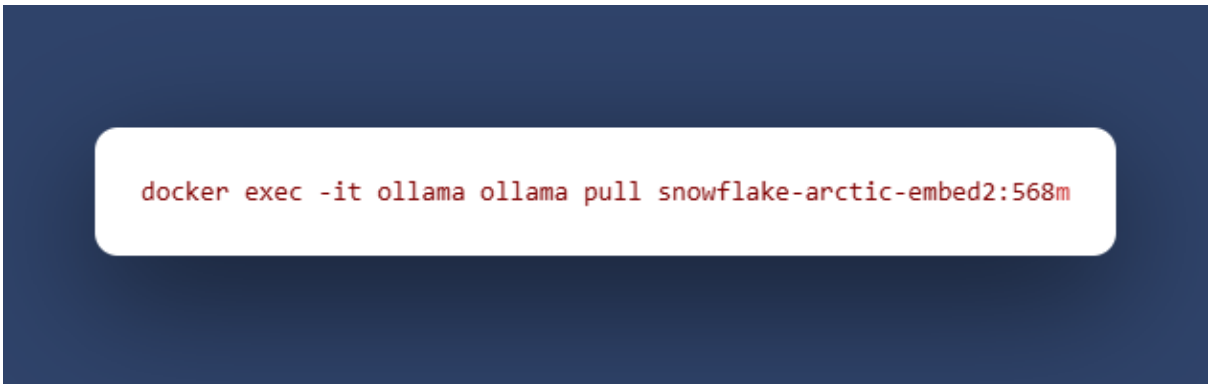
Un ulteriore requisito fondamentale è la disponibilità di una API Key per l'accesso a un Large Language Model (LLM). Il sistema è progettato in modo tale da essere indipendente dal provider specifico: è possibile utilizzare qualsiasi chiave API a disposizione, purché il modello sottostante sia sufficientemente performante per gestire task complessi. Qualora non si disponga di alcuna chiave, è possibile ottenerne una gratuitamente registrandosi sulla piattaforma Groq, accessibile al link https://console.groq.com/home, che mette a disposizione modelli LLM adatti a questo tipo di applicazioni.

Per quanto riguarda la componente di ricerca semantica, il progetto utilizza un database vettoriale Weaviate. Anche in questo caso non vi sono vincoli stringenti sull'istanza utilizzata: è possibile impiegare qualsiasi soluzione Weaviate, sia locale che remota. Durante l'implementazione del progetto è stata utilizzata un'istanza online messa a disposizione personalmente, ma l'architettura del sistema consente di integrare facilmente alternative differenti semplicemente configurando le variabili presenti nel file .env Questo approccio rende il sistema adattabile a diversi contesti infrastrutturali.

Un discorso analogo vale per il modello di embedding. Nel progetto è stato utilizzato un endpoint remoto basato su Ollama, gratuito e open source, sul quale era già installato il modello di embedding impiegato. Tuttavia, anche in questo caso, non si tratta di una scelta vincolante: è possibile sostituire il modello o l'endpoint senza modificare la logica applicativa, agendo unicamente sulla configurazione.

Infine, qualora **alcune delle risorse descritte non fossero disponibili in forma remota, il sistema può essere eseguito interamente in locale**. In questo scenario, l'utilizzo di **container Docker** rappresenta una soluzione particolarmente efficace, poiché consente di isolare i vari componenti, semplificare la configurazione dell'ambiente e garantire una maggiore riproducibilità dell'intero sistema. Ecco una possibile configurazione del file docker-compose.yml che si può utilizzare:

Con il seguente comando da terminale si può scaricare il modello di embedding utilizzato:



E poi si possono eseguire i file ‘create_csv.py’ e ‘create_collection.py’ come spiegato nel readme.md.

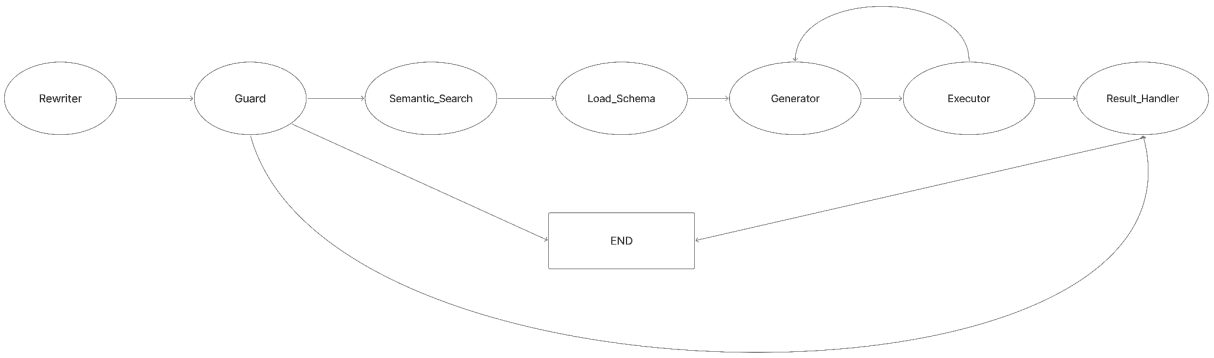
Il file docker-compose.yml mostrato rappresenta una configurazione generale, poi chi utilizza il software è libero di scegliere se usare entrambi i container o solo uno. In ogni caso nel file di configurazione delle variabili di ambiente (.env) è possibile andare a specificare tutte le variabili per sovrascrivere quelle specificate nel docker-compose.yml e il codice di ‘create_collection.py’ è stato implementato per consentire di inserire manualmente i parametri per fare gli embedding.

6. Architettura Software e Workflow

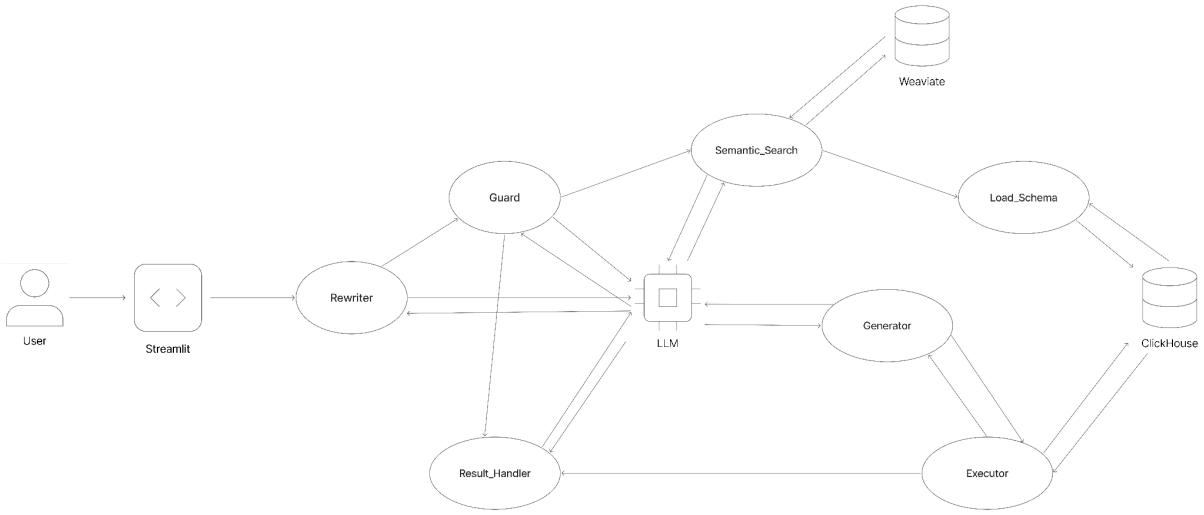
L’architettura del sistema è basata su un grafo di esecuzione realizzato mediante LangGraph, in cui ogni nodo rappresenta un agente o un componente con una responsabilità ben definita. Tra i vantaggi di questo approccio troviamo il fatto di poter stabilire con chiarezza il flusso di controllo, si possono facilmente estendere i componenti in futuro e si può rilevare immediatamente il punto in cui si verifica un errore.

Come in un normale grafo i nodi sono collegati tramite archi, che definiscono l’ordine di esecuzione delle diverse fasi del processo. Ogni nodo opera su uno stato condiviso, arricchendolo progressivamente con nuove informazioni utili alle fasi successive.

Di seguito uno schema generale del grafo dei nodi:



Di seguito uno schema più dettagliato che mostra tutti i componenti del sistema e le loro interazioni:



Di seguito è descritto l'approccio seguito per costruire la collezione weaviate e poi sono descritti i principali nodi che compongono il grafo e il ruolo svolto da ciascuno di essi.

6.1 Ricerca di similarità e embedding

Nel progetto è stato introdotto un database vettoriale Weaviate con l'obiettivo di rendere più selettive le interrogazioni generate a partire

dalle richieste in linguaggio naturale. La fonte primaria dei dati rimane il database ClickHouse aziendale; tuttavia, alcune informazioni presenti al suo interno risultano particolarmente adatte a essere utilizzate anche in un contesto di ricerca semantica.

In una fase preliminare è stata effettuata una query su ClickHouse direttamente da codice Python per raccogliere tutte le informazioni di interesse relative ai prodotti. I dati estratti sono stati successivamente salvati in un file CSV, che ha costituito la base per la creazione e il popolamento della collezione all'interno di Weaviate. Una volta definita la struttura della collezione, gli oggetti sono stati caricati nel database vettoriale a partire dal contenuto del file, associando a ciascun elemento il relativo embedding.

Per quanto riguarda la struttura degli oggetti memorizzati in Weaviate, sono stati selezionati esclusivamente alcuni campi ritenuti particolarmente significativi dal punto di vista descrittivo e semantico. In particolare, ogni oggetto contiene i seguenti attributi: il codice prodotto (cod prod), la descrizione del prodotto (descr prod) e le descrizioni dei livelli gerarchici dell'albero merceologico (descr_liv1, descr_liv2, descr_liv3, descr_liv4). La scelta di questi campi è stata guidata dal fatto che essi presentano un elevato numero di valori distinti e che, soprattutto, possono dare origine a diverse ambiguità quando l'utente formula richieste poco specifiche o utilizza denominazioni che non coincidono perfettamente con i valori presenti nel database ClickHouse.

Sebbene la maggior parte delle informazioni risieda già nel database analitico, questi campi risultano particolarmente critici in fase di interpretazione della richiesta utente, poiché non è sempre immediato stabilire quale colonna utilizzare nella query SQL o a quale prodotto ci si stia effettivamente riferendo. Tali informazioni sono

state utilizzate per effettuare confronti di similarità tra gli oggetti della collezione e i prodotti menzionati nella richiesta dell'utente, insieme a un insieme di sinonimi generati dinamicamente dal sistema (agente semantic_search). La ricerca è stata implementata adottando un approccio ibrido, che combina la somiglianza sintattica con quella semantica, così da sfruttare sia il matching testuale sia le rappresentazioni vettoriali.

Il vettore di embedding associato a ciascun oggetto è stato costruito utilizzando il modello snowflake-artic-embed 2:568, reso disponibile tramite un endpoint Ollama. Sebbene gli oggetti contengono anche il codice prodotto, tale campo è stato esplicitamente escluso dal processo di embedding, in quanto costituito esclusivamente da valori numerici e privo di un reale contenuto informativo dal punto di vista semantico.

6.2 Stato Condiviso

Nel sistema sviluppato la comunicazione tra i diversi nodi del grafo è resa possibile attraverso delle informazioni condivise, questo concetto è modellato come uno stato di un automa a stati finiti secondo il paradigma proposto da LangGraph. Le informazioni dello stato vengono progressivamente arricchite durante l'esecuzione del workflow.

Dal punto di vista implementativo, lo stato è stato definito tramite una classe Python basata su typedDict. Questa scelta consente di descrivere in modo esplicito la struttura dello stato, specificando per ciascun campo il tipo atteso, in questo modo la struttura dati definita risulta più leggibile pur mantenendo la flessibilità di un dizionario. Lo stato include innanzitutto la richiesta dell'utente, rappresentata

come una stringa che può essere raffinata nelle prime fasi della pipeline. Accanto a questa informazione è presente lo schema della tabella sales_data, recuperato dinamicamente dal database ClickHouse e rappresentato come una struttura chiave–valore che associa a ciascuna colonna il relativo tipo di dato. Un ruolo centrale nello stato è occupato dalla query SQL generata dal sistema, che viene costruita dall'agente dedicato e successivamente eseguita sul database. In caso di errore durante l'esecuzione, il messaggio restituito da ClickHouse viene memorizzato in un campo specifico dello stato, consentendo al sistema di analizzare l'errore e tentare una rigenerazione della query.

Lo stato prevede inoltre un campo per il risultato dell'esecuzione della query, che viene inizialmente acquisito in forma grezza e successivamente convertito in una struttura dati più adatta alla visualizzazione, come un dataframe. A valle dell'esecuzione, viene popolato anche un campo dedicato al commento finale, generato da un modello LLM con lo scopo di fornire una spiegazione testuale dei risultati ottenuti.

Per garantire un comportamento controllato del workflow, lo stato include un contatore dei tentativi di generazione della query SQL, utilizzato per limitare il numero massimo di retry in caso di errori sintattici. È inoltre presente un flag booleano che indica se la richiesta dell'utente è stata giudicata pertinente rispetto al contesto applicativo, permettendo di interrompere anticipatamente la pipeline nel caso di richieste fuori dominio.

Infine, lo stato mantiene una lista di prodotti ritenuti rilevanti dal punto di vista semantico individuati durante la fase di ricerca su Weaviate. Queste informazioni vengono successivamente sfruttate come supporto alla generazione della query SQL, contribuendo a

rendere le interrogazioni più selettive coerenti con la richiesta dell'utente.

Alla fine della pipeline tutte queste informazioni salvate nello stato sono utilizzate per visualizzare dinamicamente gli elementi dell'interfaccia grafica.

Diagramma UML della classe GraphState

6.3 Nodo *Rewriter*

Il nodo *Rewriter* rappresenta il primo nodo della pipeline ed è responsabile di riscrivere la query utente. Le interrogazioni formulate in linguaggio naturale dagli utenti possono essere mal strutturate, e quindi questo agente utilizza un llm per 'normalizzare' la richiesta ricevuta e limitare la propagazione di ambiguità lungo la pipeline.

Modifica il campo 'user_question' dello stato.

6.4 Nodo Guard

Il nodo *Guard* è responsabile del filtraggio delle richieste dell'utente. Questo nodo utilizza un LLM per valutare se la richiesta espressa in linguaggio naturale sia pertinente rispetto al contesto applicativo del sistema.

In particolare, il nodo verifica che la domanda dell'utente sia riconducibile a un'interrogazione dei dati di vendita e non a richieste fuori ambito. In caso di richiesta non pertinente, il workflow può essere interrotto, evitando l'esecuzione di operazioni non necessarie nelle fasi successive.

Questo agente implementa l'AI Design Pattern Guardrail.

Modifica il campo 'is_relevant' dello stato.

6.5 Nodo *Semantic Search*

Il nodo *Semantic Search* ha il compito di disambiguare e arricchire semanticamente la richiesta dell'utente, con particolare riferimento ai prodotti citati nella domanda. In una prima fase, il nodo utilizza un LLM per fare del preprocessing per modificare la richiesta dell'utente per renderla quanto più chiara e meno ambigua possibile, e successivamente generare sinonimi o varianti semantiche dei prodotti o concetti a partire dalla richiesta. Successivamente, i termini ottenuti vengono utilizzati per interrogare il database vettoriale Weaviate, al fine di individuare oggetti semanticamente correlati alla richiesta. Il risultato di questa operazione è un insieme di entità rilevanti che possono essere sfruttate nelle fasi successive per 'aiutare' l'agente generatore a capire quali colonne e quali valori usare per rendere la query SQL più selettiva e coerente con l'intento dell'utente.

Modifica il campo 'relevant_items' dello stato.

6.6 Nodo *Load Schema*

Il nodo *Load Schema* è dedicato al recupero dello schema della tabella 'sales_data' dal database ClickHouse. Questo nodo esegue una query sul database per ottenere informazioni strutturali, come i nomi delle colonne e i relativi tipi.

La scelta di isolare questa operazione in un nodo dedicato è motivata dalla necessità di riutilizzare lo schema nelle fasi successive del workflow. Poiché le informazioni sullo schema possono essere necessarie più volte, ad esempio in caso di errori sintattici nella query generata, il nodo consente di interrogare il database una sola volta e

di memorizzare il risultato nello stato condiviso, migliorando l'efficienza complessiva del sistema.

Modifica il campo 'table_schema' dello stato.

6.7 Nodo *Generator*

Il nodo *Generator* rappresenta il componente centrale del sistema. Questo agente utilizza un LLM per tradurre la richiesta dell'utente in una query SQL eseguibile sul database ClickHouse.

Le istruzioni per la generazione della query sono specificate nel prompt. In particolare, si tiene conto di diverse informazioni disponibili nello stato del grafo, tra cui:

- la richiesta originale dell'utente;
- lo schema della tabella recuperato dal nodo *Load Schema*;
- metadati relativi al significato semantico delle colonne;
- direttive operative definite in fase di progettazione.

Questo approccio consente di guidare il modello linguistico nella produzione di query coerenti sia dal punto di vista sintattico sia semantico.

Modifica il campo 'sql_query' dello stato.

6.8 Nodo *Executor*

Il nodo *Executor* ha il compito di eseguire la query SQL generata sul database ClickHouse. Questo nodo rappresenta il punto di contatto diretto tra il workflow basato su agenti e il sistema di gestione dei dati.

In questa fase viene verificata la correttezza effettiva della query attraverso la sua esecuzione. Il sistema è in grado di gestire eventuali errori restituiti dal database, attivando un meccanismo di rigenerazione della query (in cui il flusso ritorna al nodo *Generator* e si tiene conto dell'errore nel prompt del LLM) o dopo 3 tentativi falliti mostrando un messaggio informativo per l'utente.

Modifica i campi 'query_result' e 'query_error' dello stato.

6.9 Nodo *Result Handler*

Il nodo *Result Handler* è responsabile dell'elaborazione finale dei risultati ottenuti dall'esecuzione della query. In particolare, il risultato viene convertito in una struttura dati adeguata, come un dataframe, per facilitarne la visualizzazione e l'ulteriore elaborazione.

In aggiunta, il nodo utilizza un LLM per commentare il risultato prodotto, fornendo una descrizione testuale che aiuti l'utente a interpretare correttamente i dati restituiti.

Modifica il campo 'final_comment' dello stato.

6.10 Archi del grafo

Gli archi del grafo definiscono il flusso di esecuzione tra i nodi, stabilendo l'ordine con cui le diverse operazioni vengono effettuate. Ogni arco rappresenta una dipendenza logica tra due componenti e consente il passaggio delle informazioni accumulate nello stato condiviso. Gli archi sono specificati nel durante la definizione del grafo e comprendono sia archi diretti che archi condizionali. Gli archi diretti sono caratterizzati da un nodo sorgente e un nodo destinazione ben definiti e l'arco viene attraversato sempre. Gli archi condizionali invece hanno una sorgente ben definita e più possibili destinazioni. La decisione su quale percorso seguire viene effettuata tramite delle funzioni di instradamento specificate in fase di dichiarazione dell'arco. All'interno del progetto le decisioni di routing vengono prese valutando lo stato del grafo. In particolare si fa una verifica sul campo 'is_relevant' per decidere se dal nodo *Guard* si debba continuare l'esecuzione della pipeline o no, e poi successivamente si fanno delle verifiche sui campi 'query_error' e 'retry_count' per decidere se dal nodo *Executor* si debba andare avanti nella pipeline o tornare indietro al nodo *Generator*

7. Demo

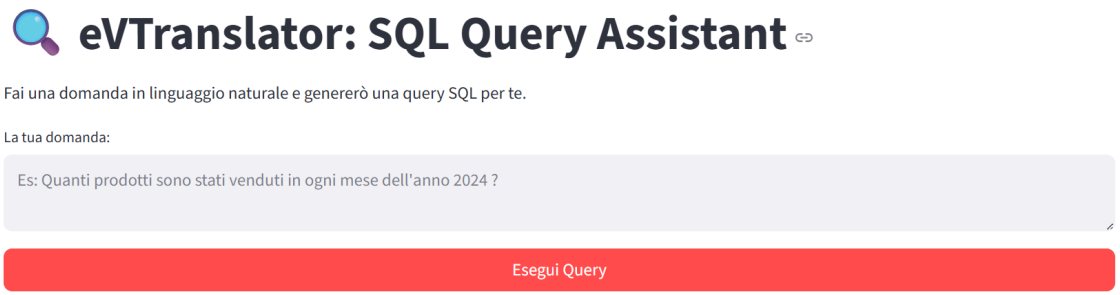
La demo è stata realizzata mediante l'utilizzo della libreria Streamilit. Fornisce un'interfaccia grafica intuitiva che consente di interagire con il sistema senza la necessità di utilizzare strumenti a riga di comando. L'interfaccia presenta una casella di testo in cui l'utente può inserire la propria richiesta in linguaggio naturale e un pulsante che permette di avviare l'esecuzione dell'intera pipeline. A seguito dell'attivazione,

il sistema esegue il workflow descritto nelle sezioni precedenti e visualizza dinamicamente i risultati ottenuti, adattando l'output grafico in base all'esito della richiesta.

Nel caso in cui la richiesta dell'utente non sia pertinente rispetto al contesto applicativo, l'interfaccia mostra un messaggio di avviso che informa l'utente dell'impossibilità di procedere con l'elaborazione. Questo comportamento è coerente con il meccanismo di filtraggio implementato nel nodo *Guard* e consente di evitare esecuzioni inutili.

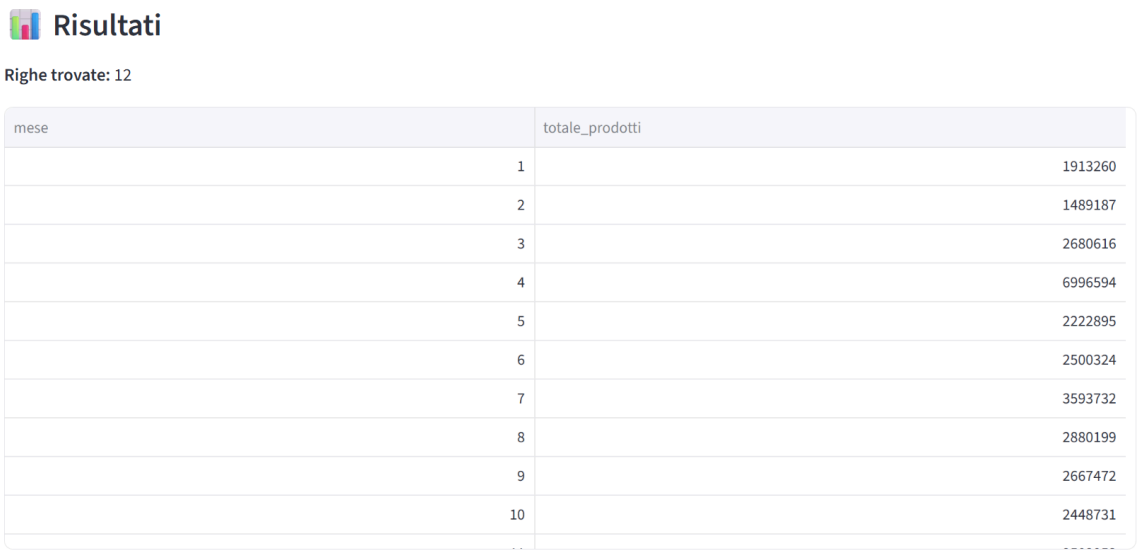
Qualora la pipeline venga completata con successo, l'interfaccia visualizza diversi elementi informativi. In primo luogo, viene mostrato il codice SQL generato dal sistema, permettendo all'utente di comprendere come la richiesta in linguaggio naturale sia stata tradotta in una query. Successivamente, viene presentata una tabella contenente i risultati restituiti dal database, accompagnata da un pulsante che consente di scaricare i dati in formato CSV per eventuali analisi successive. Infine, il sistema fornisce un commento testuale che facilita la comprensione dei risultati ottenuti.

Come detto in precedenza è previsto un meccanismo di gestione degli errori nel caso in cui il modello non riesca a generare una query SQL sintatticamente corretta. Dopo un massimo di tre tentativi falliti, l'interfaccia visualizza l'errore restituito dal database, corredato anche in questo caso da un commento esplicativo .



Schermata iniziale

Query SQL generata



Risultati dell'esecuzione su ClickHouse

Gestione di una domanda non coerente con il contesto

Gestione degli errori ripetuti nella generazione della query SQL

8. Conclusioni

Il progetto presentato ha mostrato come l'integrazione tra LLM, database analitici e una interfaccia grafica interattiva possa dare origine a un sistema efficace per l'interrogazione di basi di dati a partire da richieste in linguaggio naturale. L'obiettivo iniziale di migliorare l'accessibilità ai dati aziendali è stato pienamente soddisfatto. L' elemento centrale del sistema è l'adozione di un'architettura ad agenti, che ha permesso di suddividere il workflow in componenti con responsabilità ben definite. Questo approccio ha reso l'architettura estremamente modulare, consentendo di modificare o migliorare in futuro singoli nodi del grafo senza impattare sull'intero sistema. Inoltre, l'utilizzo di LangGraph e in generale la modellazione del processo come un grafo di esecuzione

ha semplificato la gestione del workflow consentendo di creare un sistema robusto.

Durante la fase di sviluppo è stato dedicato particolare impegno al prompt engineering dei diversi componenti basati su LLM, riconosciuto come uno strumento fondamentale per ottenere buone prestazioni. In particolare, l'inclusione nel prompt della semantica delle colonne della tabella e di informazioni relative a prodotti semanticamente correlati alla richiesta dell'utente ha contribuito a migliorare la qualità delle query SQL generate e la loro coerenza con i dati.

Durante lo sviluppo sono stati utilizzati principalmente i modelli LLM openai/gpt-oss-120b e llama-3.3-70b-versatile, mentre per la generazione degli embedding è stato impiegato il modello snowflake-arctic-embed2:568m.

Alla luce del fatto che sono presenti 4 agenti che ne fanno uso, il punto critico del sistema è il modello LLM. Quest'ultimo deve necessariamente essere di buona qualità e fornire alte prestazioni. Un discorso analogo si può fare anche per il modello di embedding, ma nel contesto del progetto è meno rilevante rispetto al modello LLM.

Ad esempio, il meccanismo di generazione dei sinonimi implementato nell'agente *Semantic Search* è stato introdotto proprio con l'obiettivo di fornire al modello di embedding un contesto semantico più ricco, migliorando l'efficacia delle ricerche di similarità; ma comunque se il modello LLM non è di buona qualità ne risente tutta la pipeline. Infatti, già all'inizio della pipeline l'agente 'Semantic_search' potrebbe non essere in grado di individuare i prodotti citati nella richiesta e di conseguenza non verrà generato nessun sinonimo. Questo causa a cascata dei problemi con la query

su weaviate che restituisce dei risultati non correlati con la richiesta e quindi l'agente generatore ha grandi probabilità di sbagliare la query o comunque di usare dei valori non completamente esatti.

Un'altra limitazione correlata è il consumo di token, che rappresenta un aspetto rilevante poiché la pipeline prevede più chiamate a modelli linguistici durante le diverse fasi del workflow. Questo elemento può incidere sui costi operativi in un contesto di utilizzo continuativo. In un'ottica di adozione in ambiente di produzione, l'utilizzo di piani premium rappresenterebbe un investimento opportuno.

Dopo aver fatto queste osservazioni importanti si può comunque concludere che utilizzando i due modelli sopra citati con hardware Groq i risultati ottenuti sono stati molto soddisfacenti. Infatti il sistema riesce a rispondere correttamente alla maggioranza delle richieste che gli vengono sottoposte. Se poi chi utilizza il software ha anche una conoscenza pregressa dei dati e nella richiesta fornisce qualche indicazione circa la colonna specifica a cui si riferisce o il corretto valore delle entry a cui è interessato allora le interrogazioni diventano ancora più efficaci.