ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Integrazione di RAG e LLM nello Sviluppo del Software

Tesi di laurea in:
Programmazione ad oggetti

Relatore
Prof. Viroli Mirko

Candidato
Bollini Simone

Correlatore
Dott. Aguzzi Gianluca

IV Sessione di Laurea
Anno Accademico 2023-2024

Abstract

I Large Language Model (LLM) addestrati per sviluppare il codice sono oggi altamente efficaci e in grado di generare soluzioni di qualità. L'addestramento fatto sui modelli è però su fonti generali, questo non da quindi la possibilità al modello di generare soluzioni su misura per una specifica richiesta utilizzando codice già creato dal programmatore o dalla propria azienda per casi simili. Da questo nasce l'esigenza di addestrare il modello per personalizzare le soluzioni proposte, contestualizzandole alla propria realtà aziendale e al proprio stile nel programmare ma il fine-tuning di un LLM è un processo molto costoso e difficile da utilizzare per mantenere aggiornato frequentemente la base di conoscenza del modello. Per rispondere a questa esigenza entra in gioco la Retrieval-Augmented Generation (RAG), che permette di aumentare la conoscenza del modello, recuperando informazioni da una propria base di conoscenza, esterna al modello, come librerie specifiche di un azienda, arricchendo il prompt della query di input che sarà elaborata dal LLM. Il RAG, ricercando semanticamente i chunk maggiormente somiglianti a quanto richiesto se trovati, li inserirà per aumentare il prompt del LLM, estendendo la base di informazioni sulla quale genererà l'output con la risposta. Questa tesi approfondisce questi concetti e sperimenta l'integrazione di un RAG specifico per codice Java e un LLM con lo scopo di ottenere dal LLM risposte personalizzate che solo con la conoscenza del LLM, anche se estremamente performante e completo, sarebbe stato impossibile ottenere.

A Giulia e ai miei figli, il dono più grande.
A tutta la mia famiglia.

Grazie a tutti voi.

Contents

Chapter 1

Introduzione

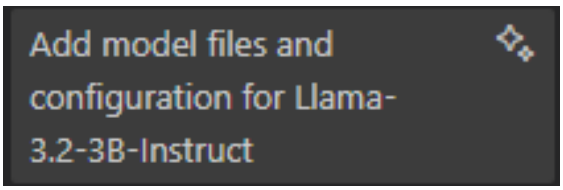
1.1 Essere programmatori nel 2025

Per sviluppare codice sono disponibili tantissimi (IDE), uno di questi è **Visual Studio Code**, mentre per condividere i progetti e lavorare in team lo strumento utilizzato potrebbe essere **GitHub**. Se richiesta memoria GPU per piccoli progetti accademici è disponibile ad esmpio **COLAB**, che permette di eseguire in remoto codice utilizzando GPU senza costi. Questi esempi sono parte di una panoramica di strumenti sempre più vasta, complessa e in rapita evoluzione, con un frequente cambio di software per realizzare un programma. Inoltre i possibili modi per realizzare i progetti è aumentanta, disponendo oggi di sempre più librerie e metodi per realizzare il codice. Un esempio d'utilizzo degli strumenti sopra elencati potrebbe seguire la seguente scaletta:

- Realizzazione iniziale del progetto in locale utilizzado Visual Studio Code
- Push del codice su GitHub per condividere il progetto con il team
- Esecuzione del codice su Colab per testare il codice su GPU ed eventualmente eseguire modifiche concluse con nuovo Push su GitHub
- Pull in locale per continuare a lavorare sul progetto

Questo modo dinamico di lavorare è recente ma non una novità, come invece lo sono alcuni specifici strumenti forniti da questi software tutti accumunati **dall'implementazione**

al loro interno di funzioni basate sull'IA. Queste funzioni sono in grado di completare il codice, suggerire correzioni e creare documentazione pertinente. GitHub ha introdotto **Copilot**, un assistente IA per la scrittura del codice, questo strumento è integrabile per vari IDE tra cui proprio Visual Studo Code. Un esempio semplice ma che offre già un idea della vastità e della potenza di queste funzioni è l'utility di **Github Copilot** 'Generate Commit Message with Copilot' che propone il testo da utilizzare come descrizione di un commit, ho provato a riscontrare quanto fosse contestualizzato e coerente con quanto aggiornato e ho ottenuto il seguente risultato:



Nel mio caso quanto proposto era corretto ed ho quindi eseguito il Commit con la descrizione proposta. Quanto è riuscito a fare Copilot è strabiliante, in pochi istanti ha analizzato il contesto ritornando come output una risposta semplice ma coerente rispetto a quanto cambiato. L'uso di questi strumenti sta rendendo il lavoro molto più dinamico e veloce, riducendo le interruzioni nel cercare soluzioni o per trovare le giuste parole per descrivere quanto fatto.

L'intelligenza artificiale sta rivoluzionando il modo in cui il software viene sviluppato, strumenti come Copilot utilizzando tutto il loro potenziale, possono creare la spina dorsale di un progetto in poco tempo lasciando al programmatore il compito di verificare e correggere solo in parte il codice proposto. In progetti complessi questo non riduce il ruolo del programmatore, anzi lo eleva a compiti di precisione e ad alto valore aggiunto delegando la stesura di parti del codice semplici e ripetitive al software stesso. Per questi motivi capire come funzionano oggi questi strumenti è importante, sapere come chiedere e formulare correttamente le domande al LLM è fondamentale, esplicitando nel dettaglio con parole chiave mirate come deve essere realizzato il codice per indirizzarlo nell'elaborazione e ragionamento corretto. Altro compito complesso per il programmatore è non farsi troppo ammaliare dalle soluzioni proposte perché non sempre necessarie per quanto richiesto oppure diverse da quanto già conosciuto per realizzare una determinata funzione. Questo nuovo modo di lavorare permette di conoscere nuove soluzioni ma comporta test e tempo non sempre disponibile, il programmatore deve sempre avere il controllo del progetto, accettando generazione del codice automatica solo dove consapevole di quanto proposto e del suo impatto anche in casi di revisione e manutenzione futuri. Il codice deve rimanere rapidamente leggibile e coerente in tutte le sue parti, far generare il codice in automatico può portare ad una perdità di coerenza e leggibilità. Proprio per questo l'ultimo miglio da percorrere per sfruttare questi strumenti è la personalizzazione delle risposte del LLM, per ottenere risposte rimanendo nel contesto e nello stile di quanto già realizzato e conosciuto, per fare questo entra in gioco il **Fine-Tuning** e i **RAG** che verrano ampiamente approfonditi.

Chapter 2

Addestrare un LLM per la Generazione del Codice

L'addestramento di LLM per la generazione del codice di programmazione richiede una serie di passaggi complessi e costi significativi. Conoscere questo processo, senza addentrarsi nel dettaglio, è utile per poter poi comprendere al meglio la successiva implementazione con le tecniche di **RAG**. La procedura si divide nelle seguenti fasi:

2.1 Scelta Modello

Gli LLM utilizzano tipicamente architetture basate su trasformatori, che sono particolarmente efficaci nell'elaborazione di sequenze di dati, come il testo e il codice. I trasformatori utilizzano meccanismi di auto-attenzione per valutare l'importanza di diversi elementi in una sequenza, permettendo al modello di comprendere le relazioni tra parole o token. Questa capacità è fondamentale nella generazione del codice, poiché le dipendenze tra variabili e funzioni possono estendersi su ampie sezioni del codice, richiedendo al modello di considerare un ampio contesto per trovare le risposte corrette. L'architettura del modello scelto influenzerà in maniera decisiva tutte le successive fasi di addestramento. È utile notare che sebbene i trasformatori siano attualmente lo standard, esistono anche altri approcci come le reti neurali ricorrenti (RNN e LSTM) e nuove tecniche in continua evoluzione

come i Large Concept Models [WFS+24].

2.2 Raccolta e Preparazione dei Dataset

La qualità e la quantità dei dati per l'addestramento è di primaria importanza per prepare un modello alla generazione di codice in maniera efficace. È quindi essenziale utilizzare per il training codice proveniente da molteplici fonti tra cui codice sorgente, file readme, documentazione tecnica, commenti nel codice, pagine Wiki, API e discussioni su forum specializzati in programmazione, arricchendo così il dataset con esempi pratici e ricchezza terminologica. In rete è possibile trovare diverso materiale open source tra cui dataset già etichettati. Alcuni dataset hanno un valore altissimo, per tutelare il costo per produrli per certi dataset è previsto il diritto d'autore. I dati si dividono in due tipologie:

- **Dati Strutturati**: seguono un formato specifico e predefinito, seguono la struttora in coppie (descrizione, codice).
- **Dati non Strutturati**: non sono organizzati e sono quindi più difficili da interpretare dal modello.

2.2.1 Pulizia e Pre-Processo

La raccolta di dati va visionata con cura, se non si conosce la provenienza del codice è possibile che contenga bug o codice obsoleto che possono essere trasmessi al modello. Con la rapida evoluzione del codice molte librerie e tecniche vengono rapidamente deprecate e superate per questo anche utilizzando i più noti modelli LLM ad oggi disponibili, può capitare di ricevere come output **codice obsoleto che risolve il quesito ma con soluzioni contenti tecniche, api e librerie deprecate o non più disponibili.** Per questo motivo i dati raccolti devono essere quindi puliti e pre-processati per rimuovere errori e informazioni non pertinenti, garantendo così un dataset di alta qualità per l'addestramento.
Il modello per poter elaborare il dataset ha bisogno che quest'ultimo venga diviso in parti più piccole chiamate token per mantenere l'integrita del dato [Sta24], i token

possono essere parole, parti di parole o singoli caratteri, e questa suddivisione è fondamentale per:

- **Gestione del contesto**: mantenere la relazione semantica tra i diversi elementi del codice
- **Efficienza computazionale**: processare grandi quantità di testo in modo ottimizzato
- **Limitazioni del modello**: rispettare i limiti massimi di input del modello (tipicamente tra 512 e 4096 token)
- **Preservazione della struttura**: mantenere la struttura sintattica del codice sorgente

Ad esempio, nel codice Java, i token potrebbero includere:

- Parole chiave (`public`, `class`, `static`)
- Identificatori (nomi di variabili e metodi)
- Operatori e simboli (`+`, `=`, {, })
- Stringhe letterali e commenti

2.3 Pre-Addestramento

Il pre-addestramento di un LLM specializzato nella generazione di codice ha lo scopo di fornire al modello una conoscenza generale della sintassi e delle strutture logiche dei linguaggio di programmazione. Durante questa fase il modello impare a generare codice partendo da dati non etichettati utilizzando tecniche come il *language modeling* autoregressivo per insegnare al modello di predire il token successivo in una sequenza. Questo approccio rende la generazione contestualmente e coerente di codice, sfruttando la capacità del modello di "ricordare" il contesto anche su ampie sequenze di dati.

2.4 Fine-Tuning

Il fine-tuning è la fase in cui il modello già pre-addestrato viene ulteriormente specializzato per la generazione di codice adattando e migliorando il modello per specifici domini di applicazione. Durante questa fase, il modello affina le sue capacità attraverso dataset specializzati composti da coppie descrizione-codice, documentazione tecnica e commenti, esempi di bug-fixing e refactoring. **Tecniche di Apprendimento**:

- **Supervisionato**: Training su coppie input-output predefinite, il modello impara a mappare input di descrizione con linguaggio naturale a output di codice corrispondente.
- **Per Rinforzo**: Ottimizzazione basata su feedback e metriche di qualità
- **Few-shot Learning**: Adattamento a nuovi contesti con pochi esempi

2.4.1 Overfitting

Il processo di fine-tuning richiede un attento bilanciamento nell'apprendere dai dati di addestramento cercando di evitare di incorrere in overfitting. L'overfitting si verifica quando il modello si specializza troppo sui dati di addestramento, riducendo la sua capacità di generalizzazione producendo risposte errate o incoerenti su nuovi dati. Per evitare l'overfitting vengono utilizzati set di validazione, regolarizzazione e tecniche di dropout.

2.5 Pre-Addestramento vs Fine-Tuning

È importante comprendere la distinzione tra queste due fasi:

Pre-Addestramento

Il pre-addestramento è la fase iniziale dove il modello:

- Acquisisce una comprensione **generale** del linguaggio di programmazione
- Viene addestrato su **grandi quantità** di codice sorgente generico

- Impara le strutture base e la sintassi del linguaggio
- Non è ancora specializzato per compiti specifici

Fine-Tuning

Il fine-tuning è invece la fase di specializzazione dove il modello:

- Si adatta a un **dominio specifico** o a compiti particolari
- Utilizza dataset specifici e composti da dati strutturati

2.6 Valutazione e Ottimizzazione

Una volta addestrato, il modello deve essere rigorosamente valutato utilizzando metriche specifiche per la generazione di codice, come la correttezza sintattica, la funzionalità e l'efficienza del codice prodotto. I risultati della valutazione possono essere utilizzati per ulteriori ottimizzazioni, come aggiustamenti dei pesi del modello, modifiche all'architettura o includere dati di addestramento aggiuntivi per affrontare eventuali carenze.

2.6.1 Metriche di Valutazione

- **Correttezza Sintattica**: Verifica che il codice generato sia sintatticamente corretto.
- **Funzionalità**: Verifica che il codice generato realizzi la funzionalità desiderata.
- **Efficienza**: Valuta le prestazioni del codice in termini di tempo di esecuzione e utilizzo delle risorse.

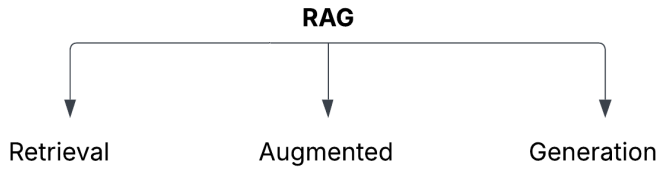
2.6.2 Tecniche di Ottimizzazione

- **Aggiustamento dei Pesi**: Modifica dei pesi del modello per migliorare le prestazioni.

- **Modifiche all'Architettura**: Introduzione di nuove componenti o modifiche a quelle esistenti.
- **Integrazione di Dati Aggiuntivi**: Utilizzo di ulteriori dati di addestramento per migliorare le prestazioni.

Chapter 3

RAG



3.1 Introduzione

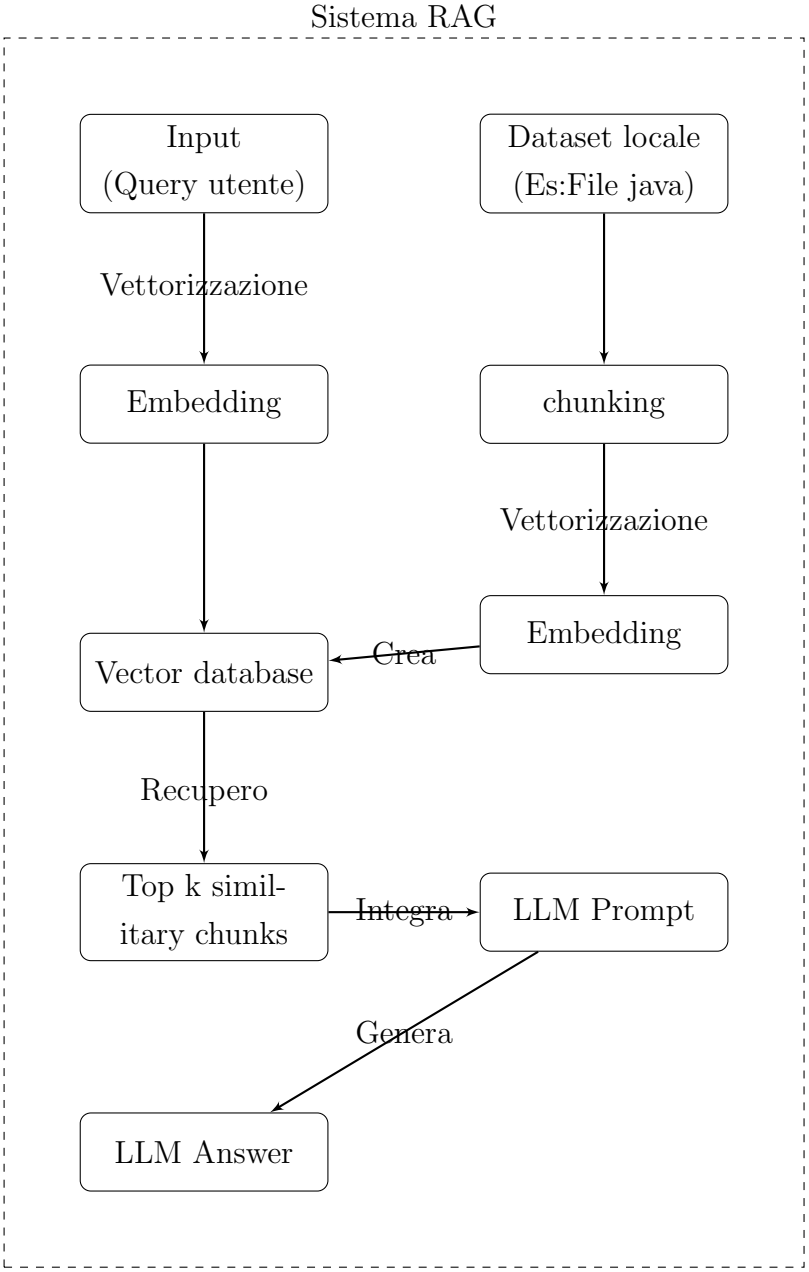
RAG **Retrieval-Augmented Generation**, (in italiano *Generazione Aumentata tramite Recupero*) è un sistema che permette di migliorare l'output di un LLM estendendo la sua conoscenza con nuove informazioni, al di fuori dai suoi dati di addestramento allo scopo di:

- ottenere risposte mirate e personalizzate contenti knowledge relativa a librerie e codice custom;
- migliorare il codice generato rendendolo più specifico al dominio riducendo le allucinazioni;
- facilitare l'assistenza da parte del modello nella fase di debugging migliorando la sua comprensione di sistemi complessi;

- supportare la creazione di documentazione aggiornata;
- permettere all'interno di un Team di migliorare la coerenza del codice scritto da diversi programmatori proponendo librerie e standard comuni;
- evitare risposte imprecise a causa della confusione terminologica, in cui diverse fonti utilizzano la stessa terminologia per parlare di cose diverse.

3.2 Funzionamento

Il sistema RAG si integra al LLM attivando un meccanismo di recupero delle informazioni per aumentare il prompt della richiesta. Il funzionamento si articola in diverse fasi qui sotto illustrate:



3.2.1 Creazione Vector Database

La propria *knowledge base* deve essere salvata in un database vettoriale, in modo da poter essere interrogata in maniera efficiente dal sistema RAG. Per creare questo database vengono utilizzati dati esterni al training set originale del LLM, provenienti da diverse fonti come:

- API e database interni
- Archivi documentali
- File di testo e codice

La creazione di un database ben strutturato e la parte più importante di tutto il processo, dividere il codice in chunk correttamente etichettando ogni elemento con i corretti metadati è fondamentale per la sucessiva fase di interrogazione. Il processo di creazione del Vector Database segue la seguente pipeline:

- **Chunking**: Divisione del codice in chunk
- **Embedding**: Conversione dei chunk in vettori numerici
- **Vector Store**: Memorizzazione degli embedding in un database vettoriale

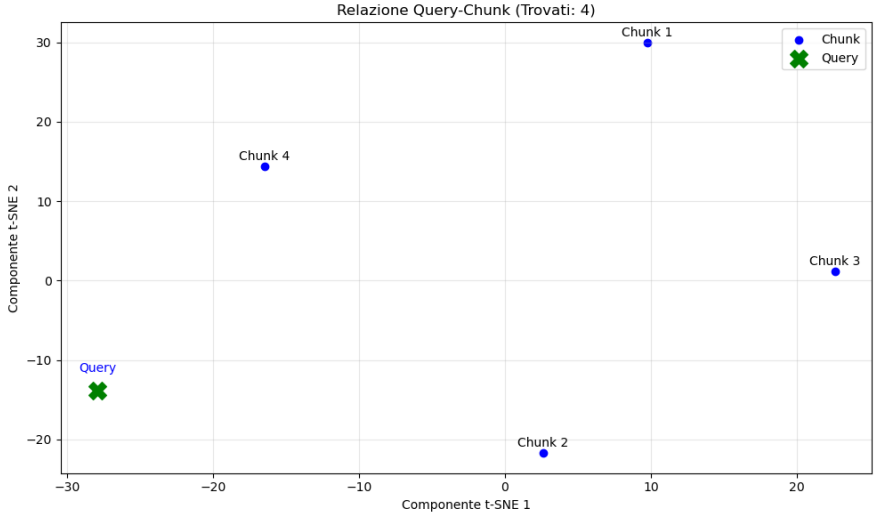
3.2.2 FASE 1: User query e function calling

Data la query d'input da parte dell'utente, il sistema RAG è avviato da una chiamata di funzione per ricercare nel **Vector Database** i chunk più rilevanti per la query. Nei modelli più complessi in RAG è di fatto un agente integrato nel sistema che viene chiamato all'occorreza qunado la base di conoscenza del LLM non è sufficiente per fornire una risposta adeguata, in questo modo viene anche razionalizzato e ottimizzato il costo computazionale del processo, attivato solo quando strettamente neceassario. Rimane comunque questo passaggio una scelta configurabile in base allo specifico utilizzo del sistema, ad esempio per un azienda che utilizza il LLM solo per compiti specifici può essere configurato il sistema in modo che chiamii la funzione RAG sempre.

3.2.3 Fase 2: Recupero delle Informazioni

Quando l'utente sottopone una query:

- La domanda viene convertita in un vettore
- Il sistema cerca nel database vettoriale le informazioni più pertinenti
- Viene calcolata la rilevanza attraverso calcoli matematici vettoriali



3.2.4 Fase 3: Aumento del Prompt

Se trovate, il sistema RAG arricchisce il prompt dell'utente con le informazioni recuperate, fornendo al LLM un contesto più ampio e dettagliato per generare una risposta coerente.

3.3 Perchè RAG

Il RAG permette di superare le limitazioni di conoscenza dei LLM, fornendo risposte accurate e contestualizzate grazie all'integrazione di conoscenze interne e personalizzate. Dopo aver costruito un sistema RAG è possibile eseguire rapidamente aggiornamenti al **Vector database**, cosa che sarebbe molto più difficile

da ottenere con il fine-tuning, che richiede tempo e risorse significative. Avere un LLM addestrato fin da subito su misura per le proprie sarebbe fantastico ma per quasi tutte le aziende richiede risorse impossibili da sostenere ed è quindi molto più facile costruire un RAG che intervenire direttamente sulla conoscenza del LLM solitamente di proprietà di terzi.

Chapter 4

Implementazione di un Sistema RAG per lo Sviluppo di codice per il linguaggio Java

4.1 Obiettivo

Questo caso studio si propone di verificare il livello di personalizzazione e qualità delle risposte di un LLM potenziando la query nel prompt di input attraverso la creazione di un sistema RAG di supporto, analizzando singolarmente le varie fasi che compongono il processo. Il sistema RAG è testato con della **classi JAVA uniche** create appositamente per il caso studio.

Problematica da affrontare:

Chiamate a più livelli di classi e metodi, dove il RAG potrebbe non essere in grado di estrapolare le informazione necessarie da inserire nel prompt per ottenere dal LLM risposte coerenti con quanto richiesto.

4.2 Architettura del Sistema

Il sistema RAG implementa un'architettura modulare composta da cinque componenti principali:

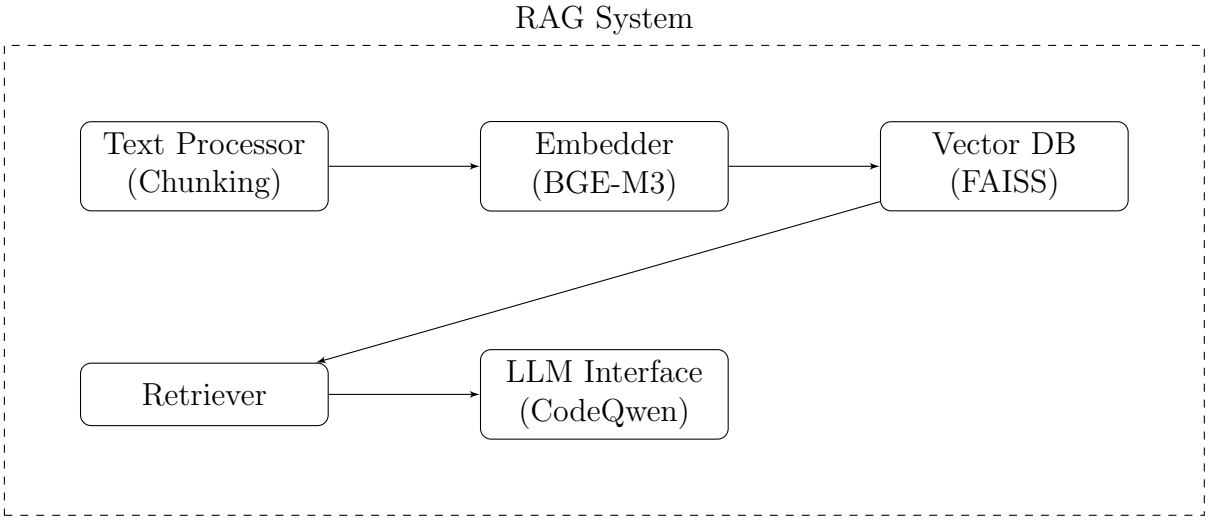


Figure 4.1: Architettura del sistema RAG

1. **Text Processor (Chunking)**:
   - Suddivide i file Java in chunk di un numero definito appositamente di token
   - Gestisce sovrapposizione di token tra chunk
   - Preserva il contesto del codice
2. **Embedder (BGE-M3)**:
   - Converte i chunk in vettori numerici
   - Utilizza il modello BGE-M3 per la generazione degli embedding
   - Normalizza i vettori per ottimizzare la ricerca
3. **Vector DB (FAISS)**:
   - Memorizza gli embedding in un database vettoriale
   - Ottimizza la ricerca per similarità
   - Garantisce recupero efficiente dei chunk rilevanti
4. **Retriever**:

- Esegue query semantiche sul database
- Recupera i k chunk più rilevanti
- Prepara il contesto per il LLM

5. **LLM Interface (CodeQwen e Llama 3.2)**:
   - Interfaccia con i modelli CodeQwen e Llama 3.2
   - Genera risposte basate sul contesto recuperato
   - Ottimizza il prompt per la generazione di codice

4.3 Software Utilizzati

4.3.1 Ollama

Ollama [Oll24] è un software che permette di utilizzare in locale LLM senza dover dipendere da servizi cloud esterni. Il software è stato scelto per la sua flessibilità, permettendo di integrare facilmente i modelli LLM nel sistema RAG.

4.3.2 LLM

Ogni LLM è specializzato per determinati scopi, per questo motivo per rendere più completa la ricerca sono stati utilizzati due modelli con caratteristiche differenti:

Llama 3.2

Llama 3.2 3B [AI24b], un modello di linguaggio open source. Il modello, con 3 miliardi di parametri, è ottimizzato per compiti di dialogo multilingue e si distingue per le sue capacità di recupero e sintesi delle informazioni. La scelta è ricaduta su questa versione per il suo equilibrio tra prestazioni e requisiti computazionali che permettono il suo utilizzo senza hardware troppo potente.

Codeqwen 1.5

Codeqwen [Tea24b] è un modello di linguaggio open source specializzato nella generazione di codice e documentazione tecnica. Con 7 miliardi di parametri, il

modello è stato addestrato su un ampio dataset di codice sorgente e documentazione tecnica, permettendo di generare codice coerente e ben strutturato. La scelta di questo modello è stata dettata, a differenza di llama3.2, dalla sua specializzazione nella programmazione e dalla sua capacità di generare codice di alta qualità.

4.3.3 LangChain

LangChain [Tea24a] è un framework open source progettato per costruire applicazioni basate su LLM. Fornisce strumenti avanzati per integrare modelli con dati esterni ed API, creare pipeline con chain e gestire database vettoriali, supportando l'implementazione di sistemi RAG.

4.3.4 BGE-M3

BGE-M3 [BAA24] è un modello di embedding testuale open source per la gestione di dati strutturati e non strutturati multilingue. Il modello permettendo di convertire testo in vettori numerici ad alta dimensionalità.

4.3.5 FAISS

FAISS (Facebook AI Similarity Search) [AI24a] è una libreria open source per la ricerca efficiente di similarità e il clustering di vettori densi. Progettata per gestire dataset su larga scala, FAISS supporta operazioni di ricerca anche su insiemi di vettori che superano la capacità della RAM, grazie a tecniche di indicizzazione avanzate e ottimizzazioni computazionali.

4.4 Dataset

Il dataset creato appositamente è composto da tre classi Java:

DateUtilCustom.java Classe personalizzata per gestire le date

GiorniMagici.java Classe per calcolare in maniera particolare dei giorni

BasketballStats.java Classe per calcolare statistiche relative al mondo del basket.

4.5 Scenario base del Caso Studio

La classe BasketballStats.java non ha nessuna relazione con le altre due classi, mentre DateUtilCustom.java e GiorniMagici.java sono strettamente correlate infatti GiorniMagici.java richiama metodi presenti in DateUtilCustom.java. Andremo a testare il sistema RAG con la seguente query:

- **Cosa ritorna il metodo `segnaleWow(LocalDate.of(2025, 2, 14))`?**

4.5.1 Codice di riferimento per rispondere alla query

In GiorniMagici.java è presente la seguente funzione:

Listing 4.1: Metodo segnaleWow in GiorniMagici.java

```
public static String segnaleWow(LocalDate data) {
    String wow = "il tuo segnale Wow e': " + DateUtilCustom.getMessaggioMagico(
        date);
    return wow;
}
```

Questa funzione richiama il metodo `getMessaggioMagico` presente in DateUtilCustom.java:

Listing 4.2: Metodo getMessaggioMagico in DateUtilCustom.java

```
public static String getMessaggioMagico(LocalDate datamagica) throws
    DateTimeParseException {
    DayOfWeek giornoSettimana = datamagica.getDayOfWeek();
    switch(giornoSettimana) {
        case MONDAY: return "La magia inizia nel silenzio...";
        case TUESDAY: return "I sussurri degli antichi si fanno sentire.";
        case WEDNESDAY: return "Il velo tra i mondi e' sottile oggi.";
        case THURSDAY: return "L'energia magica e' potente e chiara.";
        case FRIDAY: return "Attenzione agli incantesimi del crepuscolo.";
        case SATURDAY: return "Il giorno perfetto per scoprire segreti nascosti.";
        case SUNDAY: return "Riposa e rigenera il tuo potere magico.";
        default: return "Il giorno e' avvolto nel mistero...";
    }
}
```

4.5.2 Risultato Atteso

Essendo il 14 Febbraio 2025 un venerdì, ci aspettiamo come risposta:

> **"il tuo segnale Wow è: Attenzione agli incantesimi del crepuscolo."**

4.6 Implementazione

4.6.1 Creazione dei Chunk

I modelli di embedding hanno limiti massimi di input (512-4096 token) per questo spezzare il codice in chunk di dimensioni adeguate è obbligatorio oltre ad essere in ogni caso fondamentale. Inoltre occorre prestare attenzione alla dimensione dei chunk generati, se troppo piccoli riducono il contesto disponibile per il modello mentre se troppo grandi perdono focalizzazione semantica. Per suddividere il file Java in chunk viene utilizzata la libreria **langchain_text_splitters.** Il seguente codice Python mostra come suddividere i file Java in chunk di dimensione fissa, salvando i risultati in un file JSON.

Listing 4.3: Codice Python per la suddivisione dei file Java in chunk

```
from langchain_text_splitters import RecursiveCharacterTextSplitter
import json

# Funzione per caricare e suddividere un file Java
def process_file(file_path):
    with open(file_path, "r", encoding="utf-8") as f:
     lines = f.readlines()

    # Ricostruisce il testo mantenendo le informazioni sulle linee
    text = ''.join(lines)

    splitter = RecursiveCharacterTextSplitter(
    chunk_size=256, #molto basso per prevenire merge di metodi
    chunk_overlap=64,
    separators=[
        # I seguenti separatori sono stati usati per mantenere i metodi uniti
        # Prioritari: catturano la fine dei metodi
        "\n}\n\npublic ",
        "\n}\n\nprivate ",
        "\n}\n\nprotected ",
        "\n}\n\nstatic ",
```

```
            "\n}\n\n// End of method",

            # Secondari: separatori generici
            "\n}\n",  # Qualsiasi chiusura di blocco
            "\nclass ",  # Inizio nuove classi
            "\n@",  # Annotazioni
            "\n/**",  # Javadoc
            "\n * ",
            "\n"
        ],
        keep_separator=False, # per evitare riporto di separatori
        is_separator_regex=False
    )

        chunks = splitter.split_text(text)
        # Calcola le linee esatte per ogni chunk
        chunk_metadata = []
        cursor = 0
        for chunk in chunks:
            start_line = text.count('\n', 0, cursor) + 1
            chunk_length = len(chunk)
            end_line = text.count('\n', 0, cursor + chunk_length) + 1
            chunk_metadata.append({
                "start_line": start_line,
                "end_line": end_line,
                "text": chunk
            })
            cursor += chunk_length

        return chunk_metadata

    # Carica e suddividi i file Java
    files = ["my_project/DateUtilCustom.java", "my_project/GiorniMagici.java", "
        my_project/BasketballStats.java"]
    all_chunks = []

    for file_path in files:
        chunks_info = process_file(file_path)
        for chunk_info in chunks_info:
            chunk_text = chunk_info["text"]

            # Aggiungi contesto strutturale
            class_context = ""
            if "class " in chunk_text:
                class_name = chunk_text.split("class ")[1].split("{")[0].strip()
                class_context = f"Classe: {class_name}\n"

            all_chunks.append({
                "id": len(all_chunks) + 1,
                "text": f"// File: {file_path}\n{class_context}{chunk_text}",
```

```
                "source": file_path,
                "type": "code",
                "start_line": chunk_info["start_line"],
                "end_line": chunk_info["end_line"],
                "class": class_context.replace("Classe: ", "") if class_context
                    else ""
            })

    # Salva i chunk in un file JSON
    with open("chunks.json", "w", encoding="utf-8") as f:
        json.dump(all_chunks, f, indent=4, ensure_ascii=False)
```

Il chunking è costruito in maniera specifica per **codice java**, i separatori sono stati scelti per tentare di segmentare il codice secondo la struttura tipica dei metodi e delle classi, garantendo che il chunk contenga blocchi di codice "interi". L'opzione `keep_separator=False` crea punti di split più naturali per il codice Java, allineandosi meglio con la struttura dei metodi e delle classi. Per ciascun chunk, se nel testo è presente la stringa `class`, il codice estrae il nome della classe (prendendo il testo che segue `class` fino al primo {) e lo utilizza per creare un contesto strutturale (es `Classe: NomeClasse`). Questo contesto viene preappeso al testo del chunk e salvato anche come valore nel campo "class".

Il risultato nel file chunks.json è il seguente:

Listing 4.4: Esempio di chunks generati


```
[
    {
        "id": 1,
        "text": "// File: my_project/DateUtilCustom.java\
         nimport java.text.ParseException;\nimport java.
         text.SimpleDateFormat;\nimport java.time.
         DayOfWeek;\nimport java.time.LocalDate;\nimport
         java.time.format.DateTimeParseException;\nimport
         java.util.Calendar;\nimport java.util.Date;",
        "source": "my_project/DateUtilCustom.java",
        "type": "code",
        "start_line": 1,
        "end_line": 7,
        "class": ""
```

```
        },
        {
            "id": 2,
            "text": "// File: my_project/DateUtilCustom.java\
             nClasse: DateUtilCustom\nimport java.util.
             Calendar;\nimport java.util.Date;\nimport java.
             util.concurrent.TimeUnit;\npublic class
             DateUtilCustom {\n    /**\n     * Formatta una
             data nel formato \"dd/MM/yyyy\".\n     *\n     *
             @param date La data da formattare.",
            "source": "my_project/DateUtilCustom.java",
            "type": "code",
            "start_line": 7,
            "end_line": 16,
            "class": "DateUtilCustom\n"
        }
        ............continua
    ]
```

Ogni chunk mantiene:

- Il riferimento al file sorgente
- Il nome della classe
- Le righe di inizio e fine nel file originale
- Il contenuto del codice con la sua struttura

4.6.2 Arricchire i chunk con metadati relativi al codice

Oltre al testo del codice, è importante mantenere informazioni aggiuntive per facilitare la ricerca e l'interpretazione dei chunk. La seguente funzione `extract_method_name` aggiunge una stringa contestuale per ogni chunk che include:

- Il nome del metodo o della classe
- La classe di appartenenza
- Le righe di inizio e fine del codice

Listing 4.5: Funzione extract_method_name

```
    import re
    def extract_method_name(text):
        # Pattern per la firma di un metodo in Java
        method_pattern = r'(?:public|private|protected|static|final|synchronized|
            abstract|native)\s+[\w<>\[\]]+\s+(\w+)\s*\([^)]*\)'

        # Pattern per i costruttori
        constructor_pattern = r'(?:public|private|protected)\s+(\w+)\s*\([^)]*\)'

        # Cerca la firma di un metodo
        matches = re.findall(method_pattern, text)
        if matches:
            return matches[0]  # Restituisce il primo metodo trovato

        # Cerca costruttori
        constr_matches = re.findall(constructor_pattern, text)
        if constr_matches:
            return constr_matches[0] + " (costruttore)"

        # Cerca chiamate a metodi
        method_calls = re.findall(r'\.(\w+)\s*\(', text)
        if method_calls:
            return f"Chiamata a: {method_calls[-1]}"

        return "unknown_method"  # Default se non trova nulla
```

4.6.3 Generazione degli Embedding

Gli embedding trasformano i chunk in rappresentazioni vettoriali che catturano il significato semantico. Il seguente codice Python mostra come generare gli embedding e creare un database Faiss. Come precedentemente descritto, il modello di embedding utilizzato è BGE-M3, questo modello usa due rappresentazioni per complementarietà, la rappresentazione denza cattura relazioni semantiche mentre quella sparsa cattura relazioni sintattiche. Mentre sul database FAISS ad alta dimensionalità verrà settata la ricerca di somiglianza utilizzando la distanza euclidea tra i vettori.

Listing 4.6: Codice Python per la generazione degli embedding e la creazione di un database FAISS

```
        import json
        from sentence_transformers import SentenceTransformer
```

```
    from langchain_community.vectorstores import FAISS

    # 1. Carica i chunk dal file JSON
    with open("chunks.json", "r", encoding="utf-8") as f:
        chunks_data = json.load(f)

    chunks = [item["text"] for item in chunks_data]

    # 2. Carica il modello BGE-M3 e genera gli embedding
    embedder = SentenceTransformer('BAAI/bge-m3')
    embeddings = embedder.encode(
        [f"METHOD:{extract_method_name(c['text'])} CLASS:{c['class']} LINES:{c
            ['start_line']}-{c['end_line']} CONTENT:{c['text']}"
         for c in chunks_data],
        show_progress_bar=True
    )

    # 3. Crea un database FAISS
    vector_store = FAISS.from_embeddings(
        text_embeddings=list(zip(chunks, embeddings)),  # Abbina testi e
            embedding
        embedding=embedder,  # Modello per future operazioni
    )

    # 4. Salva il database
    vector_store.save_local("./faiss_db")
    print("Database FAISS creato e salvato in ./faiss_db.")
```

Il metodo *encode()* del modello BGE-M3 genera gli embedding per ogni chunk, chiamando la funzione *extract_method_name* per arricchire il contesto e creare vettori con relazioni semantiche strutturate.

4.6.4 Esecuzione di query sul Database FAISS

Una volta creato il database FAISS, è possibile eseguire ricerche semantiche sui chunk memorizzati:

Listing 4.7: Esecuzione di una query sul database FAISS

```
from langchain_community.vectorstores import FAISS
from langchain_huggingface import HuggingFaceEmbeddings

# 1. Carica il modello di embedding nel formato corretto
embedder = HuggingFaceEmbeddings(
    model_name="BAAI/bge-m3",
    model_kwargs={'device': 'cpu'},
    encode_kwargs={'normalize_embeddings': True}
```

```
    )

    # 2. Carica il database FAISS esistente
    vector_store = FAISS.load_local(
        folder_path="./faiss_db",
        embeddings=embedder,
        allow_dangerous_deserialization=True
    )

    # 3. Query di esempio
    query = "Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 1, 10))?"

    # 4. Cerca i chunk piu' simili
    docs = vector_store.similarity_search_with_score(
        query,
        k=5,
        score_threshold=0.90,  # bassa similarita'
        search_type="similarity",  # Piu' efficace per il codice
        lambda_mult=0.5        # Bilancia diversita'/rilevanza
    )

    # 5. Stampa i risultati con relativo score
    for i, (doc, score) in enumerate(docs):
        print(f"Risultato {i+1} (Score: {score:.4f}):")
        print(doc.page_content)
        print("-" * 40)
```

Risultati con query base (senza alcun riferimento al metodo utilizzato all'interno di segnale Wow)

- **Query:**
  "Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 1, 10))?"

- **Output:** viene restituito il chunk corretto con uno score di similarità di **0.6547**. Questo valore, basato sulla cosine similarity, non è particolarmente alto ma sufficiente per identificare il chunk corretto.

Nota: È importante riscontrare che viene restituito un solo chunk nonostante `k=5`. Questo accade perché nessun altro chunk supera la soglia di similarità impostata. Tale comportamento evidenzia una criticità: la funzione `segnaleWow` richiama un metodo presente nella libreria `DateUtilCustom` che non viene estratto

dal Dataset.

Riformulazione query (aggiungendo riferimento al metodo utilizzato all'interno di segnale Wow)

- **Per risolvere questo problema, la query è stata riformulata:**

  > **"Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 1, 10)) che utilizza la funzione getMessaggioMagico() della libreria DateUtilCustom?"**

- **L'output fornisce 5 risultati:**
  - Primo chunk **(score: 0.5276)**: contiene la funzione `segnaleWow`
  - Secondo, terzo e quarto chunk **(scores: 0.7188, 0.7258, 0.7605)**: contengono la funzione `getMessaggioMagico`
  - Quinto chunk **(score: 0.8958)**: funzione non rilevante relativa alle date

Conclusione: Sono state riscontrate due problematiche molto rilevanti, la prima riguarda la mancanza di estrazione di metodi da librerie esterne se non esplicitate nella query. Mentre la seconda guarda i chunck estratti, lo score ottenuto non è particolarmente alto e questo con un database più ampio potrebbe portare a risultati non coerenti. Per il secondo punto questa analisi ha portato alla decisione di abbassare `score_threshold` da 0.90 a 0.80, questa piccola correzione risolve in parte la problematica o almeno evita di propagarla ulteriormente preferendo non ottenere risultati piuttosto che ricevere risposte non coerenti.

4.6.5 Creazione della Pipeline RAG

Listing 4.8: Pipeline RAG

```
1  from langchain_community.vectorstores import FAISS
2  from langchain_huggingface import HuggingFaceEmbeddings
3  from langchain_ollama import OllamaLLM
4  from langchain.chains import create_retrieval_chain
5  from langchain.chains.combine_documents import create_stuff_documents_chain
```

```
6       from langchain.prompts import PromptTemplate
7
8       # Configurazione embedding
9       embedder = HuggingFaceEmbeddings(
10          model_name="BAAI/bge-m3",
11          model_kwargs={'device': 'cpu'},
12          encode_kwargs={'normalize_embeddings': True}
13      )
14
15      # Caricamento database FAISS
16      vector_store = FAISS.load_local(
17          folder_path="./faiss_db",
18          embeddings=embedder,
19          allow_dangerous_deserialization=True
20      )
21      # Aggiunta del database FAISS al retriever
22      retriever=vector_store.as_retriever(
23              search_kwargs={
24                  "k": 5,                    # Piu' documenti per contesto
25                  "score_threshold": 0.80, # medio-bassa similarita' inizialmente
                        era 0.90
26                  "search_type" :"similarity",  # Piu' efficace per il codice
27                  "lambda_mult":0.5       # Bilancia diversita'/rilevanza
28              }
29          )
30
31      varStileLLM = "Sei un programmatore che risponde conciso ma sintetico."
32
33      # Configurazione Template del prompt specifici per i modelli
34      LLAMA_TEMPLATE = """<|begin_of_text|>
35      <|start_header_id|>system""" + varStileLLM + """<|end_header_id|>
36      Contesto: {context}<|eot_id|>
37      <|start_header_id|>user<|end_header_id|>
38      Domanda: {input}<|eot_id|>
39      <|start_header_id|>assistant<|end_header_id|>"""
40
41      CODEQWEN_TEMPLATE = """<|im_start|>system """ + varStileLLM + """
42      {context}<|im_end|>
43      {{ if .Functions }}<|im_start|>functions
44      {{ .Functions }}<|im_end|>{{ end }}
45      <|im_start|>user
46      {input}<|im_end|>
47      <|im_start|>assistant
48      """
49
50      COMMON_PARAMS = {
51          "temperature": 0.3, #lasciamo una bassa creativita' non vogliamo che
                inventi risposte
52          "top_p": 0.85  # Bilancia creativita'/controllo nei token generati
53      }
```

```
54
55      # Caricamento modello
56      def load_model(model_name):
57          models = {
58              "llama3.2": {
59                  "template": LLAMA_TEMPLATE,
60                  "params": COMMON_PARAMS
61              },
62              "codeqwen": {
63                  "template": CODEQWEN_TEMPLATE,
64                  "params": COMMON_PARAMS
65              }
66          }
67          if model_name not in models:
68              raise ValueError(f"Modello non supportato: {model_name}")
69          return OllamaLLM(
70              model=model_name,
71              **models[model_name]["params"]
72          ), PromptTemplate(
73              template=models[model_name]["template"],
74              input_variables=["input", "context"]
75          )
76
77      # Inizializza il modello
78      llm, prompt = load_model("codeqwen")
79
80      # Catena RAG
81      document_chain = create_stuff_documents_chain(llm, prompt)
82      rag_chain = create_retrieval_chain(
83          retriever,
84          document_chain
85      )
86
87      # Funzione query
88      def ask_ollama(question):
89          try:
90              result = rag_chain.invoke({"input": question})
91              print("DOMANDA:", question)
92              print("RISPOSTA:")
93              print(result["answer"])
94              print("FONTI:")
95              for i, doc in enumerate(result["context"], 1):
96                  print(f"{i}. {doc.page_content[:150]}...")
97                  if 'source' in doc.metadata:
98                      print(f" Fonte: {doc.metadata['source']}")
99                  print("-" * 80)
100         except Exception as e:
101             print(f"ERRORE: {str(e)}")
102
103     # Esempio d'uso
```

```
104     if __name__ == "__main__":
105         ask_ollama("Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 1, 10))
                che utilizza la funzione getMessaggioMagico() della libreria
                DateUtilCustom?")
106         #ask_ollama("Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 1, 10))
                ?")
```

Spiegazione Pipeline del RAG

Seguendo la struttura precedentemente creata, per eseguire l'embedder della query di input viene utilizzato il modello BAAI/bge-m3 e caricato il database FAISS contenente la **knowledge base**. La chiamata iniziale alla funzione *ask_ollama()* richiede come parametro **la query di input** che verrà processata dalla pipeline RAG. Sfruttando le funzionalità della libreria LangChain [Lan24b], ***result*** sarà un array contente la risposta("answer") e il contesto("context") fornito alla query.

rag_chain.invoke()

Questa funzione esegue la catena RAG creata tramite il metodo *create_retrieval_chain()* che prende come parametri il retriever e il document chain.

- la funzione **create_stuff_documents_chain()** carica una catena di documenti prendendo in input il modello LLM e il template del prompt.
- **load_model()** carica il modello LLM e il template del prompt in base al modello scelto sfruttando OllamaLLM e PromptTemplate.

Temperature

Per i due LLM è stata data una temperature molto bassa **0.3** in modo da garantire da parte dei LLM risposte coerenti e precise senza provi ad inventarle.

Top_p

Il parametro `top_p` è stato impostato a 0.85 per bilanciare creatività e controllo nei token generati.

System

Come parametro di sistema da passare al LLM è stato esplicitamente richiesto di rispondere in italiano come esperto di programmazione ma solo se è sicuro.

4.7 Test Sistema RAG

Valutiamo se il sistema RAG è in grado di rispondere in maniera coerente alla query proposta. Ricordiamo che il risultato atteso è: **"il tuo segnale Wow è: Attenzione agli incantesimi del crepuscolo."**

4.7.1 Query base senza riferimenti al metodo utilizzato all'interno di segnaleWow

```
‘‘Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 2, 14))?’’
```

Fonti aggiunte al prompt

Il RAG aggiunge solo il chunk dove è presente la funzione `segnaleWow` e non il chunk con la funzione `getMessaggioMagico` della libreria DateUtilCustom per questo le risposte dei due modelli sono incomplete:

Output con LLM Llama3.2

L'output ottenuto utilizzando il modello Llama3.2 è stato:

> "Il metodo `segnaleWow` restituisce una stringa che contiene un messaggio magico associato alla data specificata. In questo caso, la data è il 14 Febbraio 2025. La risposta esatta sarebbe: "il tuo segnale Wow è : [messaggio magico]" dove [messaggio magico] è il risultato dell'applicazione di 'DateUtilCustom.getMessaggioMagico(date)."

Modifica Temperature

CodeQwen provava a dare risposta anche senza avere tutte le informazioni necessarie, ho deciso di abbassare la temperatura da 0.3 a 0.2 per vedere se il modello

CodeQwen avrebbe risposto in maniera più precisa. Cosa che è avvenuta:

Output con LLM CodeQwen

Utilizzando invece il modello CodeQwen, la risposta ottenuta è stata:

> "Se si chiama il metodo 'segnaleWow' con un'istanza di 'LocalDate' che rappresenta la data "14 febbraio 2025", il valore restituito sarà una stringa che indica il segnale Wow associato a tale data. "'java String result = GiorniMagici.segnaleWow(LocalDate.of(2025, 2, 14)); System.out.println(result); "' Il metodo 'getMessaggioMagico' viene richiamato su un'istanza di 'DateUtilCustom', passando la data "14 febbraio 2025" come parametro. Questo metodo presumibilmente restituisce un messaggio magico associato a tale data, ad esempio "La data del New Year è una data speciale". Il risultato finale sarà: "' il tuo segnale Wow è : La data del New Year è una data speciale"

4.7.2 Query Completa con riferimenti al metodo utilizzato all'interno di segnaleWow

```
''Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 2, 14))
che utilizza la funzione getMessaggioMagico() della libreria
DateUtilCustom?''
```

Fonti aggiunte dal RAG al prompt

In questo caso il RAG aggiunge 5 chunk, ovvero oltre a quello relativo alla funzione `segnaleWow` anche i chunk relativi alla funzione `getMessaggioMagico` della libreria DateUtilCustom. Ne consegue che **le risposte date dai due modelli sono coerenti con quanto atteso**.

Output con LLM Llama3.2

"Il metodo 'segnaleWow' restituisce una stringa che contiene il messaggio magico per la data specificata. Per determinare il messaggio esatto, dobbiamo analizzare la data fornita: 'LocalDate.of(2025, 2, 14)'. La data è il 14 febbraio 2025.

Il giorno della settimana corrisponde a 'FRIDAY'. Quindi, il messaggio magico restituito dal metodo 'getMessaggioMagico' sarà: **"Attenzione agli incantesimi del crepuscolo."**"

Output con LLM CodeQwen

"Se si chiama il metodo 'segnaleWow' con una data di 'LocalDate.of(2025, 2, 14)', utilizzando la funzione 'getMessaggioMagico()' della libreria 'DateUtilCustom', il risultato sarà: "' Il tuo segnale Wow è : **Il giorno perfetto per scoprire segreti nascosti.** "' Questo è dato che la data 2025-02-14 cade mercoledì, quindi il metodo 'getMessaggioMagico()' restituisce il messaggio "Il giorno perfetto per scoprire segreti nascosti."

4.7.3 Commento risultati ottenuti

I risultati ottenuti mostrano come il sistema RAG sia in grado di rispondere in maniera coerente alla query proposta. La ricerca dei chunk più simili funziona corrette soprattutto se si scrive la richiesta in maniera più dettagliata possibile. CodeQwen ha sbagliato a calcolare il giorno della settimana ma la risposta utilizza correttamente il metodo getMessaggioMagico().

4.8 Valutazione del RAG con llm as a judge

Utilizzando l'approccio ***"llm as a judge"*** per valutare automaticamente quanto prodotto dal sistema RAG generiamo una serie di domande al sistema RAG operando una valutazione automatizzata delle risposte prodotte da parte di più LLM. Il dataset è stato arricchito di altre classi java generate sinteticamente dal LLM Deepssek.

4.8.1 Domande

Passando il file contenente tutte le librerie a Deepseek, sono state generate 30 domande per valutare il sistema RAG. La domanda fatta al LLM è stata:

‘‘Dalle mie classi genera 30 domande/risposte per valutare il mio rag: la prima è: Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 2, 14)) che utilizza la funzione getMessaggioMagico() della libreria DateUtilCustom?’’

Il risulta è stato il seguente:

Listing 4.9: Domande generate da Deepssek

```
id,question,context
Q1,"Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 2, 14)) che utilizza la funzione getMessaggioMagico()?","GiorniMagici.java"
Q2,"Come calcolare la media battuta con 25 valide su 80 turni?","CalcolatoreStatisticheBaseball.java"
Q3,"Quale ERA risulta da 5 punti subiti in 7 inning?","CalcolatoreStatisticheBaseball.java"
Q4,"Come validare una password 'Secret123!'?","GestionePassword.java"
Q5,"Cosa restituisce invertiStringaMantenendoMaiuscole('AbCde')?","AnalizzatoreTesto.java"
Q6,"Come convertire 10km in miglia?","ConvertitoreUnita.java"
Q7,"Quale BMI risulta da 70kg e 1.75m?","CalcolatoreBMI.java"
Q8,"Cosa significa un momentum [12,15] vs [8,10]?","BasketballStats.java"
Q9,"Come cifrare 'HELLO' con Caesar shift 3?","StrumentiCrittografia.java"
Q10,"Quale temperatura a Roma (41.9) a luglio?","SimulatoreMeteo.java"
Q11,"Cosa restituisce isDataMagica(6, 5, 2030)?","GiorniMagici.java"
Q12,"Come calcolare rata mutuo 100k al 4\% in 20 anni?","GestioneFinanzePersonali.java"
Q13,"Cosa restituisce getMessaggioMagico(LocalDate.of(2024,12,25))?","DateUtilCustom.java"
Q14,"Come calcolare VAN al 5% per flussi [100,200,300]?","AnalizzatoreInvestimenti.java"
Q15,"Quale complessita' ciclomatica per codice con 3 if e 2 while?","AnalizzatoreCodice.java"
Q16,"Cosa restituisce modelloPredaPredatore(100,50,0.1,0.05)?","SimulatoreEcologico.java"
Q17,"Come generare password sicura di 12 caratteri?","GestionePassword.java"
Q18,"Cosa restituisce convertiGrigio(0xFFAABB)?","StrumentiGrafica.java"
Q19,"Come calcolare log base 3 di 27?","CalcolatriceScientifica.java"
Q20,"Quale categoria BMI per 28.7?","CalcolatoreBMI.java"
Q21,"Cosa restituisce isPastDate(1/1/2020)?","DateUtilCustom.java"
Q22,"Come aggiungere 15 giorni al 1/1/2024?","DateUtilCustom.java"
Q23,"Quale ROI per guadagno 1500 su costo 1000?","AnalizzatoreInvestimenti.java"
Q24,"Cosa restituisce calcolaEfficienzaGiocatore(20,10,8,2,5)?","BasketballStats.java"
Q25,"Come simulare crescita popolazione 1000 al 2% in 5 anni?","SimulatoreEcologico.java"
Q26,"Cosa restituisce analisiStatisticheSquadra(85, 70, 12, 28)?","BasketballStats.java"
Q27,"Come decifrare 'KHOOR' con Caesar shift 3?","StrumentiCrittografia.java"
```

```
29 Q28,"Quale percussocritico per attivita' [5,8,3]?","GestoreProgetti.java"
30 Q29,"Cosa restituisce giorniAlmiocompleannoSpecial(1/1/2000, 'Mario')?","
       DateUtilCustom.java"
31 Q30,"Come calcolare radice 5a di 3125?","CalcolatriceScientifica.java"
```

4.8.2 Punteggio delle domande

Per valutare le domande generate da **Deepssek**, è stato chiesto a **GPT4o** e a **Mistral** di fornire un "punteggio totale" che indichi la capacità di rispondere alla domanda senza ambiguità con il contesto dato. Date la vostra risposta su una scala da 1 a 5, dove 1 significa che la domanda non è affatto risolvibile in base al contesto e 5 significa che la domanda è chiaramente e inequivocabilmente risolvibile in base al contesto.

Risultati

Entrambi i modelli hanno valutato le domande generate da Deepssek con un punteggio medio di 4.63 su 5:

- **GPT4o**: 139 su 150
- **Mistral**: 139 su 150

4.8.3 Risposta alle domande da parte dei LLM con RAG

Essendo poco performanti le versioni dei LLM installati in locale, per questo test ho valutato di utilizzare le versioni commerciali di Deepseek, o1 in Copilot e Mistral presenti in rete per rispondere alle domande. Fornendo per ogni domanda lo stesso prompt creato dal mio sistema RAG, arricchito con i chunk estratti dalla funzione vector_store.similarity_search_with_score precedentemente creata.

4.8.4 Risultati

miao

Chapter 5

Conclusioni

da completare

5.1 Impatto sullo Sviluppo Software

L'integrazione di strumenti basati su AI nel processo di sviluppo software sta rivoluzionando il settore. Durante il periodo di sviluppo di questa tesi (Ottobre 2024 - Febbraio 2025), abbiamo osservato:

- Rapida evoluzione degli strumenti di AI per lo sviluppo software
- Crescente disponibilità di soluzioni open source
- Miglioramento continuo nelle capacità di generazione e comprensione del codice

5.2 Sfide e Prospettive Future

Bibliography

[AI24a] Meta AI. Faiss: A library for efficient similarity search, 2024. Facebook AI Similarity Search library documentation. URL: https://faiss.ai/.

[AI24b] Meta AI. Llama-3.2-3b: Open foundation and fine-tuned chat models, 2024. URL: https://huggingface.co/meta-llama/Llama-3.2-3B.

[BAA24] BAAI. Bge-m3: A multi-modal model understanding images and text, 2024. HuggingFace model repository for BGE-M3, a multi-modal model for image and text understanding. URL: https://huggingface.co/BAAI/bge-m3.

[Doc24] Huggingface Docs. Lora, dec 2024. URL: https://huggingface.co/docs/diffusers/training/lora.

[Doc25] GitHub Docs. Asking github copilot questions in your ide, jan 2025. URL: https://docs.github.com/en/copilot/using-github-copilot/asking-github-copilot-questions-in-your-ide#ai-models-for-copilot-chat.

[Fac24a] Hugging Face. Llm judge: Automated evaluation cookbook, 2024. Guide for automated LLM evaluation using judge models. URL: https://huggingface.co/learn/cookbook/llm_judge.

[Fac24b] Hugging Face. Rag evaluation cookbook, 2024. Guide for evaluating Retrieval Augmented Generation systems. URL: https://huggingface.co/learn/cookbook/rag_evaluation.

[FGT+20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020. URL: https://arxiv.org/abs/2002.08155.

[Git24] GitHub. Github copilot is more than a tool, it's an ally, dec 2024. URL: https://www.linkedin.com/pulse/github-copilot-more-than-tool-its-ally-github-qhnoc/.

[JWS+24] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation, 2024. URL: https://arxiv.org/abs/2406.00515, doi:10.48550/ARXIV.2406.00515.

[Lan24a] LangChain. Langchain integration: Ollama, 2024. Documentation for LangChain Ollama integration. URL: https://python.langchain.com/docs/integrations/llms/ollama/.

[Lan24b] LangChain. Langchain retrieval chain documentation, 2024. Create Retrieval Chain API reference. URL: https://python.langchain.com/api_reference/langchain/chains/langchain.chains.retrieval.create_retrieval_chain.html.

[Met24] Meta. Llama-3.3-70b-instruct, dec 2024. URL: https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct.

[Oll24] Ollama. Ollama documentation, 2024. GitHub repository. URL: https://github.com/ollama/ollama/tree/main/docs.

[Res24] Restack. Understanding tokenization in machine learning, 2024. Guide to tokenization concepts and implementation in ML. URL: https://www.restack.io/p/tokenization-knowledge-answer-machine-learning-cat-ai.

[SBO23] Ahmed R. Sadik, Sebastian Brulin, and Markus Olhofer. Coding by design: Gpt-4 empowers agile model driven development, 2023.

URL: https://arxiv.org/abs/2310.04304, doi:10.48550/ARXIV.2310.04304.

[Sta24] Stanford University. Code generation with large language models, 2024. CS224G Course Materials. URL: https://web.stanford.edu/class/cs224g/slides/Code%20Generation%20with%20LLMs.pdf.

[Tea24a] LangChain Team. Langchain documentation, 2024. URL: https://python.langchain.com/docs/introduction/.

[Tea24b] Qwen Team. Codeqwen1.5: A code-specialized language model, 2024. URL: https://qwenlm.github.io/blog/codeqwen1.5/.

[Tea24c] Qwen Team. Qwen2.5-coder-3b: A code-specialized language model, 2024. URL: https://huggingface.co/Qwen/Qwen2.5-Coder-3B.

[WDS+20] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45. Association for Computational Linguistics, 2020. URL: https://huggingface.co/docs/transformers/index.

[WFS+24] Ziyi Wang, Hui Fang, Weiyi Sun, Moshi Wu, Yixin Chen, and Rui Wang. A survey of code llms: A journey from code completion to ai-powered programming. *arXiv preprint arXiv:2412.08821*, 2024. URL: https://arxiv.org/abs/2412.08821, doi:10.48550/arXiv.2412.08821.