

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

Correlatori

Dott. Aguzzi Gianluca

Dott. Farabegoli Nicolas

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 generiche, questo non dà quindi la possibilità al modello di generare soluzioni su misura per una specifica richiesta utilizzando casistiche già create 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. Il fine-tuning di un LLM è un processo molto costoso e non scalabile per essere aggiornato frequentemente. Per rispondere a questa esigenza entra in gioco la Retrieval-Augmented Generation (RAG), che permette aumentare la conoscenza del modello, recuperando informazioni da una base di conoscenza esterna al modello, come librerie specifiche di un'azienda, arricchendo il prompt della query di input. Ricercando semanticamente i chunk maggiormente somiglianti a quanto richiesto se trovati, si inseriranno per completare la query inviata al 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 e un LLM con lo scopo di ottenere dal LLM risposte personalizzate che solo con la conoscenza del LLM anche se estremamente performante e preparato sarebbe stato impossibile ottenere.


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

Grazie a tutti voi.

Contents

Abstract	iii
1 Introduzione	1
1.1 Essere programmatori nel 2025	1
2 Addestrare un LLM per la Generazione del Codice	5
2.1 Scelta Modello	5
2.2 Raccolta e Preparazione dei Dataset	6
2.2.1 Pulizia e Pre-Processo	6
2.3 Pre-Addestramento	7
2.4 Fine-Tuning	8
2.5 Pre-Addestramento vs Fine-Tuning	10
2.5.1 Pre-Addestramento	10
2.5.2 Fine-Tuning	10
2.6 Valutazione e Ottimizzazione	11
2.6.1 Metriche di Valutazione	11
2.6.2 Tecniche di Ottimizzazione	11
3 RAG	13
3.1 Introduzione	13
3.2 Funzionamento	15
3.2.1 Creazione degli Embedding	16
3.2.2 Fase 1: Function Calling	16
3.2.3 Fase 2: Recupero delle Informazioni	16
3.2.4 Fase 3: Aumento del Prompt	17
3.3 Perché RAG	17
4 Implementazione di un Sistema RAG per lo Sviluppo del Software	19
4.1 Obiettivo	19
4.2 Architettura del Sistema	19

CONTENTS

4.3	Software Utilizzati	21
4.3.1	Ollama 	21
4.3.2	LLM	21
4.3.3	LangChain	22
4.3.4	BGE-M3	22
4.4	Dataset	22
4.5	Implementazione	23
4.5.1	Creazione dei Chunk	23
4.5.2	Arricchire i chunk con metadati relativi al codice	27
4.5.3	Generazione degli Embedding	28
4.5.4	Esecuzione di query sul Database FAISS	29
4.5.5	Creazione della Pipeline RAG	31
4.5.6	Risultati del Sistema RAG	34
4.5.7	Valutazione del RAG	37
5	Conclusioni	39
5.1	Impatto sullo Sviluppo Software	39
5.2	Sfide e Prospettive Future	39
		41
	Bibliography	41

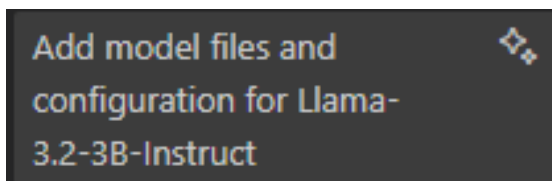
Chapter 1

Introduzione

1.1 Essere programmatori nel 2025

Sono disponibili tantissimi (IDE) per lo sviluppo del codice uno di questi è **Visual Studio Code**, mentre **GitHub** può essere lo strumento dove condividere i progetti e lavorare in team. Se richiesta memoria GPU per piccoli progetti accademici è disponibile **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 la complessità dei progetti è aumentata disponendo sempre di più librerie e metodi per realizzare il codice. Un esempio d'utilizzo con gli strumenti sopra elencato potrebbe essere la realizzazione iniziale del progetto in locale utilizzando Visual Studio Code per poi riportare il tutto su GitHub, in un secondo momento il codice viene ripreso e aperto su Colab dove a sua volta il programma viene modificato ed infine rieseguito il Push sul progetto radice presente su GitHub. Ora nel 2025, la cosa che accomuna questi strumenti, è **l'implementazione al loro interno di funzioni basate sull'IA**, 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 Studio Code. Un esempio semplice ma che offre già un'idea della vastità e della potenza di queste funzioni è l'utilità di **Github Copilot** 'Generate Commit Message with Copilot' che pro-

pone 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 perdita 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 verranno ampiamente approfonditi.

Chapter 2

Addestrare un LLM per la Generazione del Codice

L'addestramento di modelli di linguaggio di grandi dimensioni (LLM) per la generazione di codice di programmazione richiede una serie di passaggi e risorse computazionali significative. Conoscere questo processo è utile per la successiva integrazione con la 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.

2.2 Raccolta e Preparazione dei Dataset

La qualità e la quantità dei dati per l'addestramento è di primaria importanza per preparare 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 struttura 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. 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. Con la rapida evoluzione del codice è importante rimuovere frammenti di codice obsoleto **Cutilizzando questi LLM al giorno d'oggi capita spesso che propongano soluzioni utilizzando librerie deprecate.** Il modello per poter elaborare il dataset ha bisogno che quest'ultimo venga diviso in parti più piccole chiamate token per mantenere l'integrità 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 richiede un approccio specifico. A differenza del pre-addestramento generico, utilizzando i dataset precedentemente preparati il modello deve imparare a:

- Predire il completamento del codice
- Comprendere la struttura sintattica dei linguaggi di programmazione
- Riconoscere pattern comuni nel codice
- Identificare le relazioni tra diversi blocchi di codice

Un esempio pratico di pre-addestramento può essere implementato utilizzando la libreria transformers [WDS⁺20, FGT⁺20]:

```
1 from transformers import RobertaConfig, RobertaTokenizerFast
2
3 # Configurazione del modello per il codice
4 config = RobertaConfig(
```

```
5     vocab_size=50000, # Dimensione del vocabolario
6     max_position_embeddings=514, # Lunghezza massima sequenza
7     num_attention_heads=12, # Teste di attenzione
8     num_hidden_layers=6, # Strati nascosti
9     type_vocab_size=1 # Tipo di vocabolario
10 )
11
12 # Tokenizer specializzato per il codice
13 tokenizer = RobertaTokenizerFast.from_pretrained(
14     "microsoft/codebert-base",
15     max_length=512,
16     truncation=True,
17     padding=True
18 )
```

Durante questa fase, il modello sviluppa una comprensione profonda della sintassi e della semantica del codice, che verrà poi raffinata durante il fine-tuning per compiti specifici di generazione del codice.

2.4 Fine-Tuning

Il fine-tuning è la fase in cui il modello viene specializzato per la generazione di codice, documentazione e risposta a quesiti specifici del contesto di programmazione. Durante questa fase, il modello affina le sue capacità attraverso:

- **Dataset Specializzati:** Utilizzo di dataset contenenti:
 - Coppie di descrizioni-implementazioni
 - Documentazione tecnica e commenti
 - Esempi di bug fixing e refactoring
- **Tecniche di Apprendimento:**
 - **Apprendimento Supervisionato:** Training su coppie input-output predefinite
 - **Apprendimento per Rinforzo:** Ottimizzazione basata su feedback e metriche di qualità
 - **Few-shot Learning:** Adattamento a nuovi contesti con pochi esempi

Un esempio pratico di fine-tuning può essere implementato utilizzando la libreria transformers [WDS⁺20]:

```
1 from transformers import Trainer, TrainingArguments
2 from datasets import load_dataset
3
4 # Caricamento del dataset per il fine-tuning
5 dataset = load_dataset("code_search_net", "python")
6
7 # Configurazione del training
8 training_args = TrainingArguments(
9     output_dir="./results",
10     num_train_epochs=3,
11     per_device_train_batch_size=8,
12     per_device_eval_batch_size=8,
13     warmup_steps=500,
14     weight_decay=0.01,
15     logging_dir="./logs",
16     logging_steps=10,
17     evaluation_strategy="epoch"
18 )
19
20 # Inizializzazione del trainer
21 trainer = Trainer(
22     model=model,                # Modello pre-addestrato
23     args=training_args,         # Argomenti di training
24     train_dataset=dataset["train"],
25     eval_dataset=dataset["validation"],
26     tokenizer=tokenizer,        # Tokenizer specializzato per il codice
27 )
28
29 # Avvio del fine-tuning
30 trainer.train()
```

Durante il fine-tuning, il modello sviluppa capacità specifiche come:

- Generazione di codice a partire da descrizioni in linguaggio naturale
- Completamento intelligente del codice basato sul contesto
- Creazione di documentazione tecnica
- Identificazione e correzione di bug
- Refactoring del codice seguendo best practices

Il processo di fine-tuning richiede un attento bilanciamento tra:

- **Overfitting:** Evitare che il modello memorizzi i dati di training
- **Generalizzazione:** Mantenere la capacità di adattarsi a nuovi contesti
- **Prestazioni:** Ottimizzare la velocità e la qualità delle risposte

2.5 Pre-Addestramento vs Fine-Tuning

È importante comprendere la distinzione tra queste due fasi dell'addestramento:

2.5.1 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

2.5.2 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 più piccoli ma **mirati**
- Affina le conoscenze per generare codice per specifici casi d'uso

Analogia: Si può paragonare a:

- Pre-addestramento: Imparare la grammatica e il vocabolario di base di una lingua
- Fine-tuning: Specializzarsi nel linguaggio tecnico di un settore specifico

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

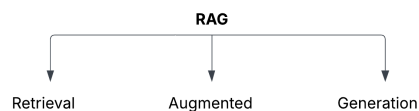
- **Correttezza Sintattica:** Verifica che il codice generato sia sintatticamente corretto.
- **Funzionalità:** Verifica che il codice generato realizzi la funzionalità desiderata.
- **Efficienza:** Valuta le prestazioni del codice in termini di tempo di esecuzione e utilizzo delle risorse.

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



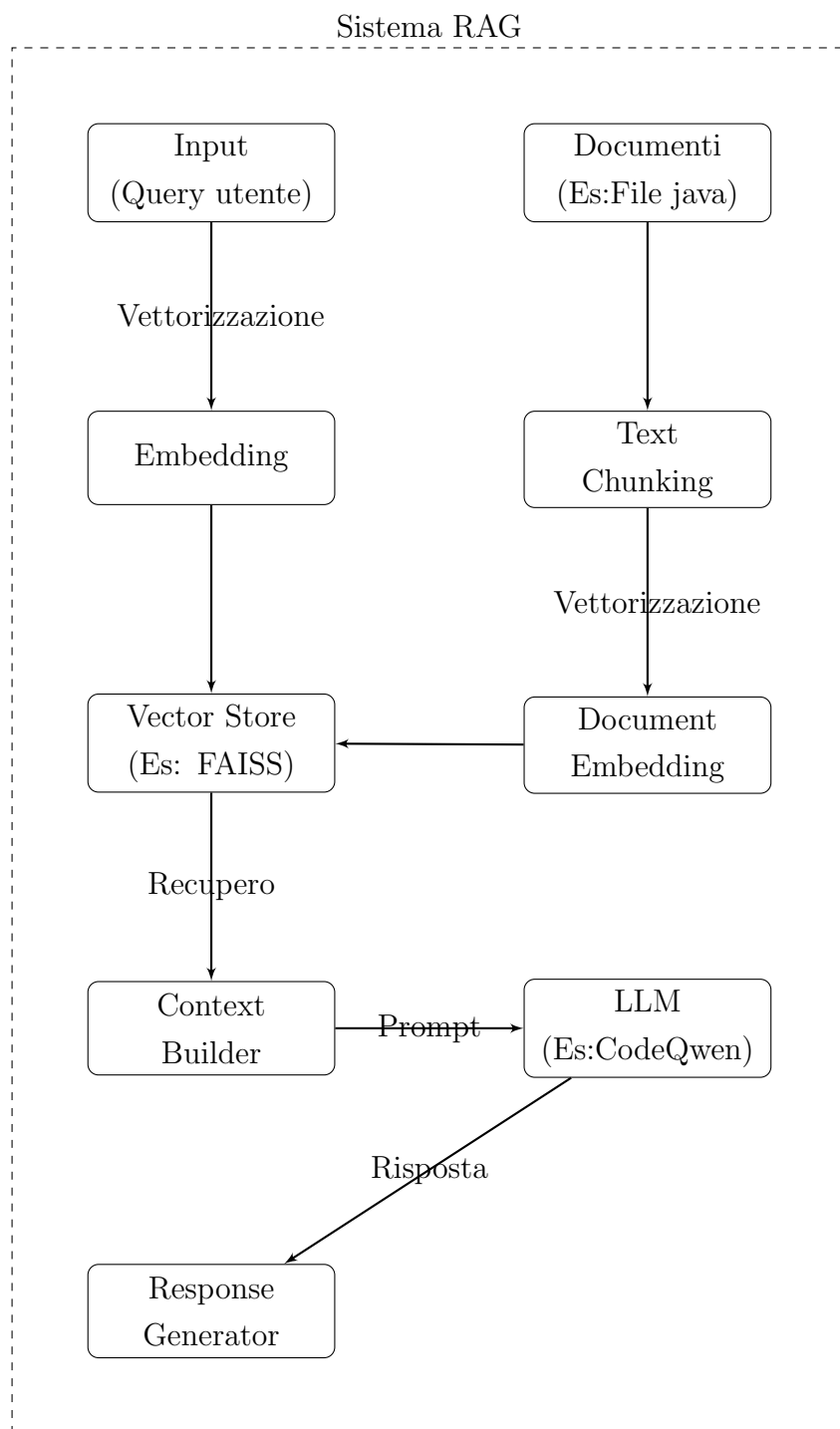
3.1 Introduzione

Il 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 personalizzate provenienti da 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;
- realizzare naturalmente senza forzature, codice più moderno 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



RAG è un sistema che integra il processo di generazione del linguaggio con un

meccanismo di recupero delle informazioni. Il funzionamento si articola in diverse fasi, come sopra illustrato.

3.2.1 Creazione degli Embedding

Il sistema RAG utilizza dati esterni al training set originale del LLM, provenienti da diverse fonti come:

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

Questi dati vengono convertiti in rappresentazioni numeriche (embedding) e archiviati in un database vettoriale, creando una knowledge base accessibile dal RAG.

3.2.2 Fase 1: Function Calling

Il sistema RAG inizia con una chiamata di funzione per ricercare nei dati di embedding:

- La query dell'utente attiva una chiamata di funzione
- Il sistema cerca nei dati di embedding le informazioni pertinenti
- Se trovate, queste informazioni vengono aggiunte al prompt

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

Il sistema RAG arricchisce il prompt dell'utente:

- Aggiunge le informazioni recuperate al contesto
- Utilizza tecniche di prompt engineering per ottimizzare la comunicazione con il LLM
- Fornisce al modello un contesto arricchito per generare risposte più accurate

3.3 Perchè RAG

Il RAG permette di superare le limitazioni dei LLM, fornendo risposte accurate e contestualizzate grazie all'integrazione di conoscenze interne e personalizzate. Costruire un RAG è una valida alternativa al fine-tuning e permette di avere rapidamente aggiornamenti sulla base dati. Cosa più difficile da ottenere con il fine-tuning, che richiede tempo e risorse significative. Inoltre è più facile costruire un RAG che intervenire direttamente sul LLM solitamente di proprietà di terzi.

Chapter 4

Implementazione di un Sistema RAG per lo Sviluppo del Software

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. 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:

1. **Text Processor (Chunking):**

- Suddivide i file Java in chunk di un numero definito appositamente di token
- Gestisce sovrapposizione di token tra chunk

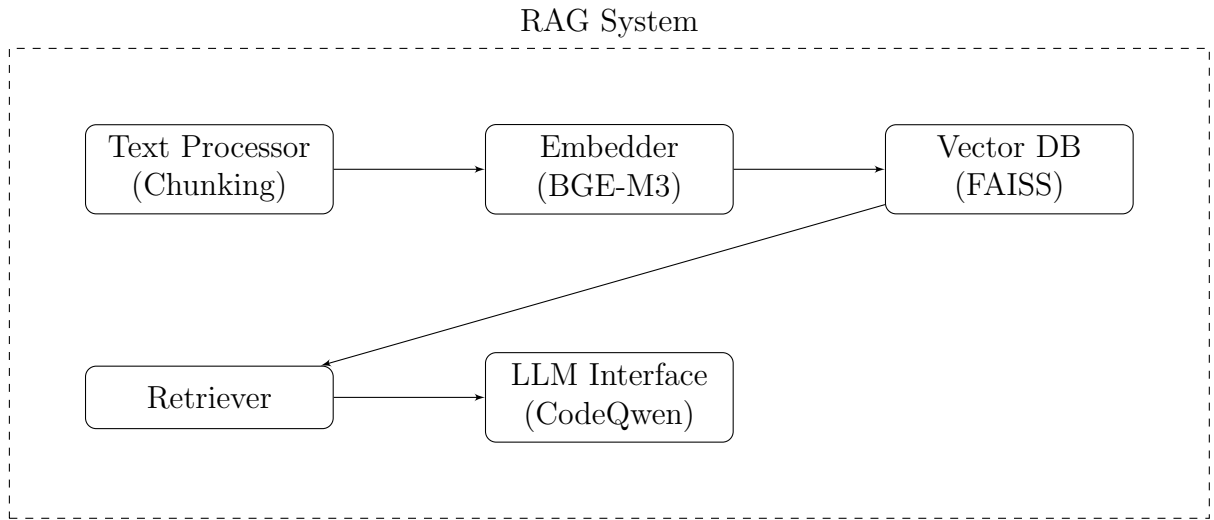


Figure 4.1: Architettura del sistema RAG

- 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. Nella scelta dei modelli è stato obbligatorio eseguire un pre filtraggio considerando solo modelli che permettessero di eseguire function calling.

Llama 3.2

Llama 3.2 3B [AI24], 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 database vettoriale open source per la gestione di dati strutturati e non strutturati multilingue, sviluppato da BigGraph Engine.

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, questa classe non ha nessuna relazione diretta con le altre due classi, mentre DateUtilCustom.java e GiorniMagici.java sono strettamente correlate. Andremo a testare il sistema RAG con il seguente scenario:

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

Codice di riferimento per rispondere alla query

In GiorniMagici.java è presente la seguente funzione:

Listing 4.1: Metodo segnaleWow in GiorniMagici.java

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

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

Listing 4.2: Metodo `getMessaggioMagico` in `DateUtilCustom.java`

```
1 public static String getMessaggioMagico(LocalDate datamagica) throws  
    DateTimeParseException {  
2     DayOfWeek giornoSettimana = datamagica.getDayOfWeek();  
3     switch(giornoSettimana) {  
4         case MONDAY: return "La magia inizia nel silenzio...";  
5         case TUESDAY: return "I sussurri degli antichi si fanno sentire.";  
6         case WEDNESDAY: return "Il velo tra i mondi e' sottile oggi.";  
7         case THURSDAY: return "L'energia magica e' potente e chiara.";  
8         case FRIDAY: return "Attenzione agli incantesimi del crepuscolo.";  
9         case SATURDAY: return "Il giorno perfetto per scoprire segreti nascosti.";  
10        case SUNDAY: return "Riposa e rigenera il tuo potere magico.";  
11        default: return "Il giorno e' avvolto nel mistero...";  
12    }  
13 }
```

Risultato Atteso

Essendo il 10 gennaio 2025 un venerdì, ci aspettiamo come risposta:

“il tuo segnale Wow è: Attenzione agli incantesimi del crepuscolo.”

4.5 Implementazione

4.5.1 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 fondamentale oltre che in ogni caso obbligatorio. 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

```

1  from langchain_text_splitters import RecursiveCharacterTextSplitter
2  import json
3
4  # Funzione per caricare e suddividere un file Java
5  def process_file(file_path):
6      with open(file_path, "r", encoding="utf-8") as f:
7          lines = f.readlines()
8
9      # Ricostruisce il testo mantenendo le informazioni sulle linee
10     text = ''.join(lines)
11
12     splitter = RecursiveCharacterTextSplitter(
13         chunk_size=512,
14         chunk_overlap=128,
15         separators=[
16             "\n}\n\npublic", # I seguenti separatori sono stati usati per provare
17                             # a mantenere i metodi uniti
18             "\n}\n\nprivate",
19             "\n}\n\nprotected",
20             "\n}\n\n//",      # Nuovo separatore per commenti
21             "\nclass ",
22             "\n@Override",    # Cattura le implementazioni di interfacce
23             "\n@Test",        # Per eventuali test case
24             "\n/**",          # Separatore per Javadoc
25             "\n * ",
26             "\n"
27         ],
28         keep_separator=True,
29         is_separator_regex=False
30     )
31
32     chunks = splitter.split_text(text)
33     # Calcola le linee esatte per ogni chunk
34     chunk_metadata = []
35     cursor = 0
36     for chunk in chunks:
37         start_line = text.count('\n', 0, cursor) + 1
38         chunk_length = len(chunk)
39         end_line = text.count('\n', 0, cursor + chunk_length) + 1
40         chunk_metadata.append({
41             "start_line": start_line,
42             "end_line": end_line,

```



```

42         "text": chunk
43     })
44     cursor += chunk_length
45
46     return chunk_metadata
47
48     # Carica e suddividi i file Java
49     files = ["my_project/DateUtilCustom.java", "my_project/GiorniMagici.java", "
50             my_project/BasketballStats.java"]
51     all_chunks = []
52
53     for file_path in files:
54         chunks_info = process_file(file_path)
55         for chunk_info in chunks_info:
56             chunk_text = chunk_info["text"]
57
58             # Aggiungi contesto strutturale
59             class_context = ""
60             if "class " in chunk_text:
61                 class_name = chunk_text.split("class ")[1].split("{")[0].strip()
62                 class_context = f"Classe: {class_name}\n"
63
64             all_chunks.append({
65                 "id": len(all_chunks) + 1,
66                 "text": f"// File: {file_path}\n{class_context}{chunk_text}",
67                 "source": file_path,
68                 "type": "code",
69                 "start_line": chunk_info["start_line"],
70                 "end_line": chunk_info["end_line"],
71                 "class": class_context.replace("Classe: ", "") if class_context
72                 else ""
73             })
74
75     # Salva i chunk in un file JSON
76     with open("chunks.json", "w", encoding="utf-8") as f:
77         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=True` fa sì che il separatore venga mantenuto nel chunk risultante. 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

```
1      [
2          {
3              "id": 1,
4              "text": "// File: my_project/DateUtilCustom.java\n
                    nClasse: DateUtilCustom\npublic class
                    DateUtilCustom {\n    public static String
                    getMessaggioMagico(LocalDate datamagica) throws
                    DateTimeParseException {\n        DayOfWeek
                    giornoSettimana = datamagica.getDayOfWeek();\n
                    switch(giornoSettimana) {\n
                    case MONDAY: return \"La magia inizia nel
                    silenzio...\";\n
                    case TUESDAY: return
                    \"I sussurri degli antichi si fanno sentire.\";\n
                    case WEDNESDAY: return \"Il velo tra
                    i mondi e' sottile oggi.\";\n
                    case
                    THURSDAY: return \"L'energia magica e' potente e
                    chiara.\";\n
                    case FRIDAY: return \"
                    Attenzione agli incantesimi del crepuscolo.\";\n
                    case SATURDAY: return \"Il giorno
                    perfetto per scoprire segreti nascosti.\";\n
                    case SUNDAY: return \"Riposa e
                    rigenera il tuo potere magico.\";\n
                    default: return \"Il giorno e' avvolto nel
                    mistero...\";\n        }\n    }\n}\",
5              "source": "my_project/DateUtilCustom.java",
6              "type": "code",
7              "start_line": 1,
8              "end_line": 29,
9              "class": "DateUtilCustom"
10         },
11         {
12             "id": 2,
13             "text": "// File: my_project/GiorniMagici.java\n
                    nClasse: GiorniMagici\npublic class GiorniMagici
                    {\n    public static String segnaleWow(LocalDate
                    data) {\n        String wow = \"il tuo segnale
```

```

14         Wow e': \" + DateUtilCustom.getMessaggioMagico(
15             date);\n            return wow;\n        }\n}\",
16     "source": "my_project/GiorniMagici.java",
17     "type": "code",
18     "start_line": 1,
19     "end_line": 8,
20     "class": "GiorniMagici"
21 }
.....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

4.5.2 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`

```

1  import re
2  def extract_method_name(text):
3      # Cerca firme di metodi Java standard
4      method_pattern = r'(?:(public|private|protected|static|final|synchronized|
5          abstract|native)\s+[\w<>\\[\]]+\s+(\w+)\s*\([^)]*\))'
6      # Cerca costruttori

```

```
7     constructor_pattern = r'(?:(public|private|protected)\s+(\w+)\s*\([^)]*\))'
8
9     matches = re.findall(method_pattern, text)
10    if matches:
11        return matches[0] # Restituisce il primo metodo trovato
12
13    constr_matches = re.findall(constructor_pattern, text)
14    if constr_matches:
15        return constr_matches[0] + " (costruttore)"
16
17    # Cerca chiamate a metodi nel chunk
18    method_calls = re.findall(r'\.(\w+)\s*\(', text)
19    if method_calls:
20        return f"Chiamata a: {method_calls[-1]}"
21
22    return "unknown_method" # Default se non trova nulla
```

4.5.3 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. Il modello di embedding utilizzato è BGE-M3, un modello pre-addestrato per la generazione di embedding con dimensioni 1024(denso) + 250K (sparso). Il modello usa due rappresentazioni per complementarità, la rappresentazione densa cattura relazioni semantiche mentre quella sparsa cattura relazioni sintattiche. Mentre FAISS [Met24b] (Facebook AI Similarity Search) è una libreria ottimizzata per la ricerca di similarità in spazi ad alta dimensionalità. Utilizza 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

```
1     import json
2     from sentence_transformers import SentenceTransformer
3     from langchain_community.vectorstores import FAISS
4
5     # 1. Carica i chunk dal file JSON
6     with open("chunks.json", "r", encoding="utf-8") as f:
7         chunks_data = json.load(f)
8
9     chunks = [item["text"] for item in chunks_data]
10
11    # 2. Carica il modello BGE-M3 e genera gli embedding
```

```

12     embedder = SentenceTransformer('BAAI/bge-m3')
13     embeddings = embedder.encode(
14         [f"METHOD:{extract_method_name(c['text'])} CLASS:{c['class']} LINES:{c
15             ['start_line']}-{c['end_line']} CONTENT:{c['text']}"]
16         for c in chunks_data],
17         show_progress_bar=True
18     )
19
20     # 3. Crea un database FAISS
21     vector_store = FAISS.from_embeddings(
22         text_embeddings=list(zip(chunks, embeddings)), # Abbina testi e
23         embedding
24         embedding=embedder, # Modello per future operazioni
25     )
26
27     # 4. Salva il database
28     vector_store.save_local("./faiss_db")
29     print("Database FAISS creato e salvato in ./faiss_db.")

```

4.5.4 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

```

1     from langchain_community.vectorstores import FAISS
2     from langchain_huggingface import HuggingFaceEmbeddings
3
4     # 1. Carica il modello di embedding nel formato corretto
5     embedder = HuggingFaceEmbeddings(
6         model_name="BAAI/bge-m3",
7         model_kwargs={'device': 'cpu'}, # Usa 'cuda' per GPU
8         encode_kwargs={'normalize_embeddings': True}
9     )
10
11     # 2. Carica il database FAISS esistente
12     vector_store = FAISS.load_local(
13         folder_path="./faiss_db",
14         embeddings=embedder,
15         allow_dangerous_deserialization=True
16     )
17
18     # 3. Query di esempio
19     query = "Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 1, 10))?"
20
21     # 4. Cerca i chunk piu' simili

```

```
22 docs = vector_store.similarity_search_with_score(  
23     query,  
24     k=5,  
25     score_threshold=0.90, # bassa similarita'  
26     search_type="similarity", # Piu' efficace per il codice usare mmr per  
27     diversita'  
28     lambda_mult=0.5 # Bilancia diversita'/rilevanza  
29 )  
30  
31 # 5. Stampa i risultati con relativo score  
32 for i, (doc, score) in enumerate(docs):  
33     print(f"Risultato {i+1} (Score: {score:.4f}):")  
34     print(doc.page_content)  
35     print("-" * 40)
```

Query Base

Con la query:

“Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 1, 10))?”

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`.

Query Migliorata

Per risolvere questo problema, la query è stata riformulata:

“Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 1, 10)) che utilizza la funzione getMessageMagico() della libreria DateUtilCustom?”

Risultati Ottenuti 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 Questa analisi ha portato alla decisione di abbassare `score_threshold` da 0.90 a 0.80, è preferibile non ottenere risultati piuttosto che ricevere risposte non coerenti.

4.5.5 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
26         "search_type": "similarity", # Piu' efficace per il codice
27         "lambda_mult": 0.5 # Bilancia diversita'/rilevanza
28     }
29 )
30
31 # Configurazione Template del prompt specifici per i modelli
```

```

32 LLAMA_TEMPLATE = """<|begin_of_text|>
33 <|start_header_id|>system<|end_header_id|>
34 Contesto: {context}<|eot_id|>
35 <|start_header_id|>user<|end_header_id|>
36 Domanda: {input}<|eot_id|>
37 <|start_header_id|>assistant<|end_header_id|>"""
38
39 CODEQWEN_TEMPLATE = """<|im_start|>system
40 {context}<|im_end|>
41 {{ if .Functions }}<|im_start|>functions
42 {{ .Functions }}<|im_end|>{{ end }}
43 <|im_start|>user
44 {input}<|im_end|>
45 <|im_start|>assistant
46 """
47
48 COMMON_PARAMS = {
49     "temperature": 0.3,
50     "top_p": 0.85, # Bilancia creativita'/controllo nei token generati
51     "system": "Rispondi in italiano come esperto di programmazione ma solo se
52                 sei sicuro."
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)

```



```

81     rag_chain = create_retrieval_chain(
82         retriever,
83         document_chain
84     )
85
86     # Funzione query
87     def ask_ollama(question):
88         try:
89             result = rag_chain.invoke({"input": question})
90             print("DOMANDA:", question)
91             print("RISPOSTA:")
92             print(result["answer"])
93             print("FONTI:")
94             for i, doc in enumerate(result["context"], 1):
95                 print(f"{i}. {doc.page_content[:150]}...")
96                 if 'source' in doc.metadata:
97                     print(f"    Fonte: {doc.metadata['source']}")
98             print("-" * 80)
99         except Exception as e:
100             print(f"ERRORE: {str(e)}")
101
102     # Esempio d'uso
103     if __name__ == "__main__":
104         ask_ollama("Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 1, 10))
105             che utilizza la funzione getMessaggioMagico() della libreria
106             DateUtilCustom?")
107         #ask_ollama("Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 1, 10))
108             ?")

```

Spiegazione pipeline

Come nei passaggi precedenti viene usata la stessa configurazione con il modello di embedder BAAI/bge-m3 e viene poi caricato il database FAISS precedentemente creato. La chiamata iniziale alla funzione `ask_ollama()` chiede come parametro la query di input che verrà processata dalla pipeline RAG. Sfruttando la libreria LangChain, `result` ritorna la risposta("answer") e il contesto("context") fornito alla query.

`rag_chain.invoke` esegue la catena RAG creata tramite il metodo `create_retrieval_chain` che prende come parametri il retriever e il document chain.

`document_chain` è una catena di documenti che prende 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.

Per i due LLM è stata data una temperature molto bassa 0.3 per garantire risposte coerenti e precise senza che inventi. Il parametro `top_p` è stato impostato a 0.85 per bilanciare creatività e controllo nei token generati. Mentre in system è stato specificato di rispondere in italiano come esperto di programmazione ma solo se è sicuro.

4.5.6 Risultati del Sistema RAG

Query Base con informazioni incomplete

Come query di test è stata utilizzata:

```
“Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 1, 10))?”
```

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 Wow specifico in base al giorno indicato. Tuttavia, non è chiaro come il messaggio sia calcolato.

Listing 4.9: Implementazione del metodo `segnaleWow`

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

In questo caso, il messaggio Wow viene calcolato chiamando l' metodo `'getMessaggioMagico'` di `'DateUtilCustom'`, che non è mostrato nel codice fornito. Tuttavia, possiamo supporre che questo metodo restituisca un valore di tipo stringa.

Il problema è che il metodo `'segnaleWow'` non contiene alcun calcolo per determinare il segnale Wow in base al giorno indicato. Pertanto, senza conoscere il comportamento del metodo `'getMessaggioMagico'`, non possiamo dire con certezza cosa ritorni il metodo `'segnaleWow'`.

Tuttavia, se supponessimo che `'DateUtilCustom.getMessaggioMagico'` restituisca un valore di tipo stringa che contiene una data o un numero, allora il risultato del metodo `'segnaleWow'` dipenderebbe dal comportamento di questo metodo.

Per esempio, se `'DateUtilCustom.getMessaggioMagico'` restituisce la data corrente, il metodo `'segnaleWow'` potrebbe restituire un messaggio Wow che contiene la data del 10 gennaio 2025: `“java public static String segnaleWow (LocalDate date) String wow = "il tuo segnale Wow è :” + DateUtilCustom.getMessaggioMagico(date) ; return wow; “`

In questo caso, il risultato del metodo `'segnaleWow'` con l'input `'LocalDate.of(2025, 1, 10)'` sarebbe: `“java public static String segnaleWow (LocalDate date) String wow = "il tuo segnale Wow è :” + DateUtilCustom.getMessaggioMagico(date) ; return wow; “` In questo caso, il risultato sarebbe: `“java "il tuo segnale Wow è :2025-01-10" “`

Tuttavia, senza conoscere il comportamento di `'DateUtilCustom.getMessaggioMagico'`, non possiamo essere certi del risultato.

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 corrisponde a `"2025-01-10"`, il metodo restituirà una stringa che indica il segnale Wow associato ad tale data.

```
“java String result = GiorniMagici.segnaleWow(LocalDate.of(2025, 1, 10)); System.out.println(result);
```

Il risultato sarà: ”il tuo segnale Wow è : L’anno 2025 ha un giorno magico che non è un numero perfetto.” Questo è perchè la data specificata (2025-01-10) corrisponde ad un anno che non ha un numero perfetto come il 28 febbraio del 2024, quindi il metodo utilizza il messaggio ”L’anno 2025 ha un giorno magico che non è un numero perfetto.” proveniente da ‘DateUtilCustom.getMessaggioMagico(date)’.

Query Completa con tutte le informazioni richieste

```
“Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 1, 10))  
che utilizza la funzione getMessaggioMagico() della libreria  
DateUtilCustom?”
```

Fonti aggiunte al prompt

In questo caso il RAG aggiunge 4 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 a quanto aspettato.

Output con LLM Llama3.2

RISPOSTA: Il metodo ‘segnaleWow(LocalDate.of(2025, 1, 10))’ ritorna un segnale Wow specifico in base al giorno indicato.

Per determinare il segnale Wow, il metodo ‘getMessaggioMagico()’ della libreria ‘DateUtilCustom’ viene chiamato con la data ‘LocalDate.of(2025, 1, 10)’ come argomento.

La funzione ‘getMessaggioMagico()’ restituisce un messaggio magico in base al giorno della settimana di una data. In questo caso, il giorno è sabato (giorno 7), quindi il messaggio ritornato sarà:

”Il giorno perfetto per scoprire segreti nascosti.”

Quindi, il metodo ‘segnaleWow(LocalDate.of(2025, 1, 10))’ restituirà la stringa ”il tuo segnale Wow è : Il giorno perfetto per scoprire segreti nascosti.”.

Output con LLM CodeQwen

Se si chiama il metodo ‘segnaleWow’ con una data di 10 gennaio 2025, utilizzando la funzione ‘getMessaggioMagico()’ della libreria ‘DateUtilCustom’, il risultato sarà:

“ ”il tuo segnale Wow è :Il giorno perfetto per scoprire segreti nascosti.” “

Questo è poiché, inizialmente, viene recuperata la data di 10 gennaio 2025 utilizzando ‘LocalDate.of(2025, 1, 10)’. Quindi, viene richiamato il metodo ‘getMessaggioMagico()’ passando questa data come parametro.

Il metodo ‘getMessaggioMagico()’ restituisce un messaggio magico in base al giorno della settimana di tale data. In questo caso, la data 10 gennaio 2025 cade mercoledì, quindi viene restituito il messaggio ”Il giorno perfetto per scoprire segreti nascosti.”

4.5.7 Valutazione del RAG

Sfruttando l’approccio ”llm as a judge” per valutare automaticamente quanto prodotto dal sistema RAG operiamo una valutazione automatica delle risposte prodotte.

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 - Gennaio 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

- [AI24] Meta AI. Llama-3.2-3b: Open foundation and fine-tuned chat models, 2024.
- [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.
- [Doc24] Huggingface Docs. Lora, dec 2024.
- [Doc25] GitHub Docs. Asking github copilot questions in your ide, jan 2025.
- [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.
- [Git24] GitHub. Github copilot is more than a tool, it’s an ally, dec 2024.
- [Hug24a] Hugging Face. Llm judge: Automated evaluation cookbook, 2024. Guide for automated LLM evaluation using judge models.
- [Hug24b] Hugging Face. Rag evaluation cookbook, 2024. Guide for evaluating Retrieval Augmented Generation systems.
- [Lan24a] LangChain. Langchain integration: Ollama, 2024. Documentation for LangChain Ollama integration.
- [Lan24b] LangChain. Langchain retrieval chain documentation, 2024. Create Retrieval Chain API reference.

- [Met24a] Meta. Llama-3.3-70b-instruct, dec 2024.
- [Met24b] Meta AI. Faiss: A library for efficient similarity search, 2024. Facebook AI Similarity Search library documentation.
- [Oll24] Ollama. Ollama documentation, 2024. GitHub repository.
- [Res24] Restack. Understanding tokenization in machine learning, 2024. Guide to tokenization concepts and implementation in ML.
- [SBO23] Ahmed R. Sadik, Sebastian Brulin, and Markus Olhofer. Coding by design: Gpt-4 empowers agile model driven development, 2023.
- [Sta24] Stanford University. Code generation with large language models, 2024. CS224G Course Materials.
- [Tea24a] LangChain Team. Langchain documentation, 2024.
- [Tea24b] Qwen Team. Codeqwen1.5: A code-specialized language model, 2024.
- [Tea24c] Qwen Team. Qwen2.5-coder-3b: A code-specialized language model, 2024.
- [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.