

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

Sommario

I Large Language Model (LLM) addestrati per generare il codice sono oggi altamente efficaci e in grado di generare soluzioni di qualità. Tuttavia, poiché il loro addestramento si basa su dataset generici, tali modelli non sono in grado di elaborare soluzioni personalizzate per specifiche esigenze, utilizzando codice già creato dal programmatore o dalla propria azienda per contesti analoghi. 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. Servirebbe quindi una nuova fase di fine-tuning per adattare il modello alle proprie esigenze, ma questa soluzione è un processo molto costoso che richiede particolari competenze tecniche difficilmente presenti in molte aziende. Inoltre il fine-tuning non permette di aggiornare il modello in maniera rapida e dinamica, richiedendo un nuovo addestramento per ogni modifica. 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 arricchendo il prompt della query in 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 risposte personalizzate che solo con la conoscenza del LLM, anche se estremamente performante e completo, sarebbero state impossibili da ottenere.

*A Giulia e ai miei figli, il dono più incredibile di questa vita.
Alla mia grande famiglia.*

Grazie

Indice

Sommario	iii
1 Introduzione	1
2 Addestrare un LLM per la Generazione del Codice	3
2.1 Scelta Modello	3
2.2 Raccolta e Preparazione dei Dataset	4
2.2.1 Pulizia e Pre-Processo	4
2.3 Pre-Addestramento	6
2.4 Fine-Tuning	6
2.4.1 Tecniche di Apprendimento nel Fine-Tuning	6
2.5 Pre-Addestramento vs Fine-Tuning	7
2.6 Valutazione e Ottimizzazione	8
2.6.1 Metriche di Valutazione	9
2.6.2 Tecniche di Ottimizzazione	9
3 Retrieval Augmented Generation	11
3.1 Introduzione	11
3.2 Funzionamento	11
3.2.1 Creazione Vector Database	12
3.2.2 Fase 1: User query e function calling	13
3.2.3 Fase 2: Recupero delle Informazioni	14
3.2.4 Fase 3: Aumento del Prompt	15
3.3 Perché RAG	16
4 Implementazione di un sistema RAG per lo sviluppo di codice per il linguaggio Java	19
4.1 Obiettivo	19
4.2 Architettura del Sistema	20
4.2.1 Text Processor (Chunking)	20
4.2.2 Embedder (BGE-M3)	20

4.2.3	Vector DB (FAISS)	21
4.2.4	Retriever	21
4.2.5	LLM Interface	21
4.3	Software Utilizzati	22
4.3.1	Ollama 🐙	22
4.3.2	Llama 3.2	22
4.3.3	Codeqwen 1.5	22
4.3.4	LangChain	22
4.3.5	BGE-M3	23
4.3.6	FAISS	23
4.4	Dataset	23
4.5	Risultato atteso caso d'uso RAG	24
4.5.1	Codice di riferimento per rispondere alla query	24
4.5.2	Output	24
4.6	Implementazione	25
4.6.1	Chunking del Codice Java	25
4.6.2	Generazione degli Embedding	29
4.6.3	Esecuzione di query sul Database FAISS	30
4.6.4	Creazione della Pipeline RAG	33
4.7	Test Sistema RAG	37
4.7.1	Query base senza riferimenti al metodo utilizzato all'interno di segnaleWow	37
4.7.2	Risultati con query che descrive accuratamente il contesto	38
4.7.3	Commento risultati ottenuti	39
4.8	Generazione codice Java specifico con RAG	40
4.8.1	Test 1: Estensione di una Classe	41
4.8.2	Test 2: Refactoring di un Metodo	42
4.8.3	Test 3: Implementazione di nuove funzionalità usando metodi di classi esistenti	42
4.8.4	Conclusioni	44
4.9	Valutazione "llm as a judge"	44
4.9.1	Creazione domande	44
4.9.2	Metrica del punteggio	45
4.9.3	llm as a judge	46
4.9.4	Valutazione risposte	47
4.9.5	Analisi risultati	48
5	Conclusioni	51
6	Ringraziamenti	55

INDICE

	57
Bibliografia	57

Capitolo 1

Introduzione

Il mondo della programmazione è un settore in continua evoluzione e negli ultimi anni ha visto un'esplosione nel campo dell'Artificial intelligence (AI). Con l'avvento di questa nuova tecnologia, per molti programmatori è cambiato il modo di lavorare: utilizzano assistenti basati sull'AI durante la produzione del codice software. Questi strumenti sono in grado di completare il codice, effettuare debugging, suggerire correzioni e creare documentazione pertinente. Aiutano inoltre i programmatori nei compiti più ripetitivi e meccanici, aumentando la produttività e riducendo i tempi di sviluppo. Ad esempio in **Github Copilot**, un assistente per la scrittura del codice basato su LLM, include il comando: *“Generate Commit Message with Copilot”* che propone testi per i commit basandosi sulle modifiche apportate al codice, come mostrato in fig. 1.1. Software come Copilot utilizzano

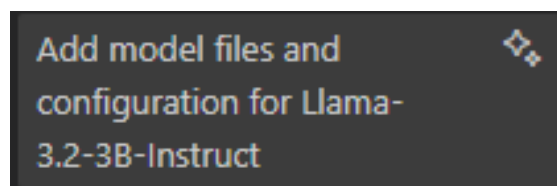


Figura 1.1: Esempio di commit autogenerato da Copilot

LLM per generare codice e testo, scansionando in pochi istanti il contesto nel codice in sviluppo. In progetti complessi, questo non riduce il ruolo del programmatore, che rimane responsabile dei compiti ad alto valore aggiunto, delegando invece la generazione di parti semplici e ripetitive a questi strumenti. È quindi fonamen-

tale comprenderne il funzionamento, saper formulare correttamente le domande, esplicitando nel dettaglio con parole chiave mirate come deve essere realizzato il codice, per indirizzare correttamente il ragionamento del LLM.



Figura 1.2: Copilot risolve gli enigmi del programmatore

Un limite significativo di questi strumenti risiede nel loro addestramento su dataset generici: forniscono risposte standardizzate, ma non sono in grado di adattarsi a esigenze specifiche di un'azienda o di un programmatore. Proprio per questo, la personalizzazione delle risposte rappresenta l'ultimo miglio da percorrere, affinché il LLM apprenda lo stile del programmatore e produca codice coerente con quanto già realizzato e conosciuto. In questo contesto si inseriscono il **Fine-Tuning** e i sistemi **RAG**, argomento centrale di questa tesi.

Capitolo 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 è utile per poter poi comprendere al meglio la successiva implementazione con le tecniche di **RAG**. La procedura si divide nelle seguenti fasi:

- Scelta del Modello
- Raccolta e Preparazione dei Dataset
- Pre-Addestramento
- Fine-Tuning
- Valutazione e Ottimizzazione

Analizziamo ora nel dettaglio ogni fase.

2.1 Scelta Modello

I LLM utilizzano tipicamente architetture basate su transformers, che sono particolarmente efficaci nell'elaborazione di sequenze di dati, come il testo e il codice. I transformers 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 transformers 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 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 questo motivo per tutelare il costo speso 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. 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 contenenti tecniche, api e librerie deprecate o non più disponibili**. Per questo motivo i dati raccolti devono essere puliti e pre-processati per rimuovere errori e informazioni non pertinenti, garantendo così un dataset di alta qualità per l'addestramento.

Tokenizzazione

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 ha lo scopo di fornire al modello una conoscenza generale della sintassi e delle strutture logiche del linguaggio di programmazione. Durante questa fase il modello impara 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.

2.4.1 Tecniche di Apprendimento nel Fine-Tuning

Nella fase di fine-tuning, il modello può utilizzare diverse tecniche di apprendimento per migliorare le sue capacità di generazione del codice. Le tecniche di apprendimento più comuni sono le seguenti.

- **Supervisionato:** Addestramento basato su coppie input-output predefinite, dove il modello impara a mappare descrizioni in linguaggio naturale al codice corrispondente.
 - *Esempio:* Utilizzo di un dataset contenente descrizioni come 'Scrivi una funzione in Python che calcoli la media di una lista' abbinate al relativo codice Python. In questo modo, il modello impara a generare il codice corretto partendo dalla descrizione fornita.

- **Per Rinforzo:** Ottimizzazione basata su un sistema di feedback, dove il modello riceve una ricompensa in base alla qualità del codice generato, come correttezza, efficienza e aderenza a specifiche metriche.
 - *Esempio:* Un modello genera una funzione di ordinamento. Il codice viene eseguito e sottoposto a una serie di test (ad esempio, verificando l'ordinamento corretto e l'efficienza computazionale). Se il codice supera i test e rispetta i criteri di prestazione, il modello riceve una ricompensa che ne rafforza le scelte, migliorando così la qualità delle future generazioni.
- **Few-shot Learning:** Capacità di adattarsi a nuovi compiti o contesti con pochissimi esempi di addestramento.
 - *Esempio:* Dopo aver osservato solo alcuni esempi di come tradurre una descrizione in linguaggio naturale al codice in un nuovo linguaggio di programmazione (ad esempio, Python), il modello è in grado di generare codice in Python anche per nuove descrizioni simili, senza necessità di un vasto dataset specifico per quel linguaggio.

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 in cui il modello:

- Acquisisce una comprensione **generale** del linguaggio di programmazione
 - *Esempio:* Il modello analizza milioni di righe di codice open-source, apprendendo le regole base e la struttura sintattica di linguaggi come Python, Java e C++.
- Viene addestrato su **grandi quantità** di codice sorgente generico

- *Esempio*: Utilizzando un vasto insieme di dati proveniente da repository pubblici (ad esempio, GitHub), il modello impara le convenzioni e le pratiche comuni adottate dalla comunità di sviluppo.
- Impara le strutture base e la sintassi del linguaggio
 - *Esempio*: Durante questa fase, il modello apprende come si definiscono funzioni, variabili, cicli e condizioni, senza però concentrarsi su particolari logiche applicative.
- Non è ancora specializzato per compiti specifici
 - *Esempio*: Pur essendo in grado di generare codice sintatticamente corretto, il modello non ha ancora imparato a ottimizzare o personalizzare il codice per particolari applicazioni, come la sicurezza o le performance.

Fine-Tuning

Il fine-tuning è la fase di specializzazione in cui il modello:

- Si adatta a un **dominio specifico** o a compiti particolari
 - *Esempio*: Un modello pre-addestrato può essere ulteriormente raffinato per generare codice dedicato allo sviluppo di applicazioni web, concentrandosi su framework come Django o Flask.
- Utilizza dataset specifici e composti da dati strutturati
 - *Esempio*: Il fine-tuning può avvenire su un dataset che contiene esempi di codice per la gestione dell'autenticazione, la validazione degli input e la gestione degli errori, rendendo il modello più efficace nel risolvere problemi tipici di un dominio applicativo specifico.

2.6 Valutazione e Ottimizzazione

Dopo l'addestramento, è fondamentale sottoporre il modello a una fase di valutazione per verificare la qualità del codice generato. Questa valutazione non si limita

a controllare la correttezza sintattica, ma si estende anche alla funzionalità e all'efficienza del codice. I risultati ottenuti offrono spunti preziosi per intervenire con ottimizzazioni mirate, come l'aggiustamento dei pesi, modifiche all'architettura o l'integrazione di ulteriori dati di addestramento.

2.6.1 Metriche di Valutazione

Per garantire che il modello produca codice di qualità, vengono utilizzate diverse metriche, tra cui:

- **Correttezza Sintattica:** Verifica che il codice generato sia privo di errori di sintassi e possa essere compilato o interpretato senza problemi.
- **Funzionalità:** Assicura che il codice realizzi effettivamente la funzionalità desiderata, testando se il comportamento del programma rispetti le specifiche iniziali.
- **Efficienza:** Valuta le prestazioni del codice in termini di tempo di esecuzione e utilizzo delle risorse, garantendo un'operatività ottimale.

2.6.2 Tecniche di Ottimizzazione

Una volta completata la valutazione, i risultati ottenuti possono guidare il processo di ottimizzazione del modello. Tra le tecniche adottabili troviamo:

- **Aggiustamento dei Pesi:** Modifica dei parametri interni del modello per migliorare la precisione e l'affidabilità del codice generato.
- **Modifiche all'Architettura:** Introduzione di nuove componenti o revisione di quelle esistenti per aumentare la capacità del modello di apprendere e generalizzare.
- **Integrazione di Dati Aggiuntivi:** Ampliamento del dataset di addestramento con ulteriori esempi, mirati a colmare le lacune individuate durante la valutazione.

Capitolo 3

Retrieval Augmented Generation

3.1 Introduzione

La sigla RAG, acronimo di **Retrieval Augmented Generation** (in italiano, *Generazione Aumentata tramite Recupero*), indica un approccio innovativo volto a potenziare le capacità di un LLM. Questo sistema amplia la base di conoscenza del modello arricchendola con informazioni esterne, al di fuori dal dataset di addestramento originale. Il prompt del LLM contenente la query in input dell'utente, prima di essere elaborato, viene arricchito con contenuti aggiuntivi chiamati chunk, ottenuti attraverso tecniche di recupero che identificano le corrispondenze rilevanti rispetto ad un proprio database vettoriale. Tale integrazione consente al modello di generare risposte più accurate, contestualizzate e aggiornate, migliorando significativamente la qualità complessiva dell'output.

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 illustrate in fig. 3.1 che analizziamo ora nel dettaglio.

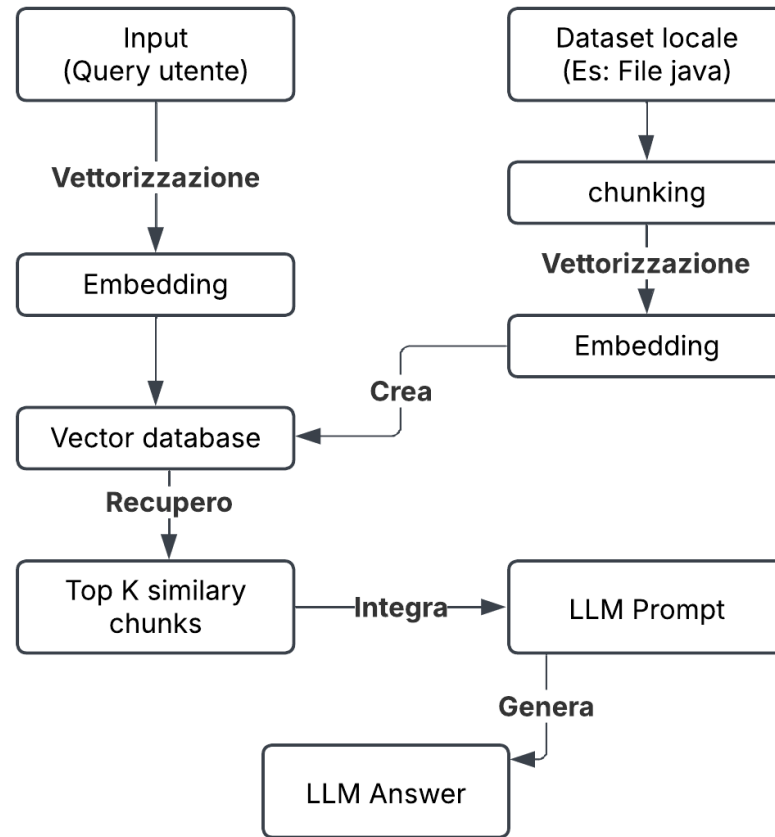


Figura 3.1: Funzionamento del sistema RAG

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 fase più importante di tutto il processo, dividere il codice in chunk correttamente etichettando ogni elemento con i corretti metadati è fondamentale per la successiva fase di interrogazione. Il processo di creazione del Vector Database segue la seguente pipeline:

- **Chunking:** Per garantire che i modelli di embedding possano lavorare efficacemente, è necessario suddividere il dataset in chunk di dimensioni controllate. Questi chunk devono essere abbastanza piccoli da mantenere la focalizzazione semantica, ma sufficientemente grandi da fornire un contesto utile. Tipicamente, i chunk variano tra 512 e 4096 token.
- **Embedding:** Conversione dei chunk in vettori numerici ad alta dimensionalità, utilizzando tecniche di embedding come Word2Vec, GloVe o BERT. Questi vettori rappresentano il contenuto semantico dei chunk, consentendo al sistema di confrontare e recuperare le informazioni in base alla similarità tra i vettori.
- **Vector Store:** Memorizzazione degli embedding in un database vettoriale per consentire una rapida interrogazione e recupero delle informazioni. Questo database deve essere progettato per garantire un'efficienza computazionale ottimale, in modo da ridurre i tempi di risposta del sistema.

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'occorrenza quando 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 necessario. 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 e sempre contestualizzati può essere configurato il sistema in modo che chiami 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 ad alta dimensionalità
- Il sistema cerca nel database vettoriale i chunk più simili alla query, calcolando la distanza tra i vettori utilizzando tecniche di confronto come la similarità coseno o la distanza euclidea. In fig. 3.2 è rappresentata, riducendo a due dimensioni lo spazio vettoriale, la distanza dei vettori più simili trovati nel database e la query.
- Se trovate corrispondenze con un determinato *score_threshold*, i chunk vengono recuperati.

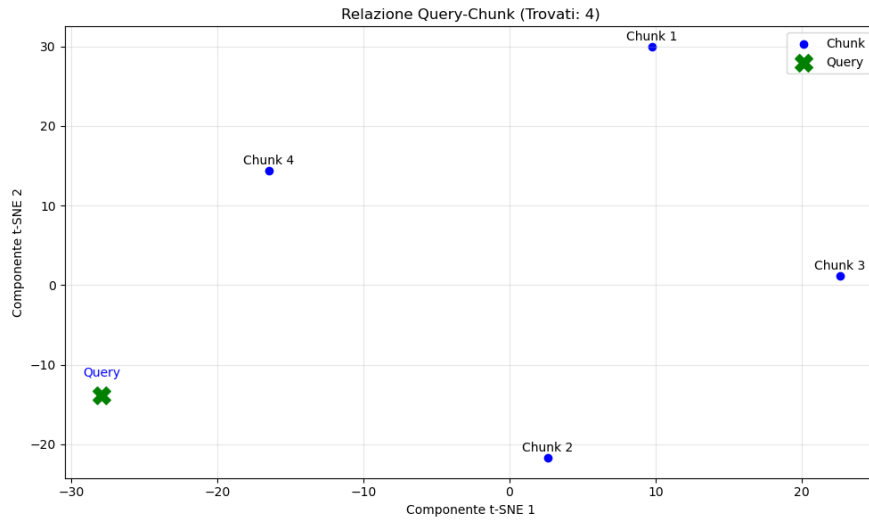


Figura 3.2: Esempio di distanze tra chunk e query

Calcolo dello *score_threshold*

In questo contesto, il parametro `score_threshold` [Lan24c] viene utilizzato per filtrare i chunk restituiti in base alla loro somiglianza con la query. La somiglianza viene tipicamente calcolata mediante la *cosine similarity* tra le embedding della query e quelle dei documenti presenti nel database. Ogni chunk viene associato

a un punteggio che rappresenta il grado di somiglianza con la query, e solo i chunk con un determinato `score_threshold` vengono restituiti. Per leggere questo parametro va prestata particolare attenzione al tipo di somiglianza calcolata, nella *cosine similarity* il valore va da 0 è 1.

3.2.4 Fase 3: Aumento del Prompt

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. In questo modo, il LLM riceve un input contenente le informazioni relative alla risposta che dovrà generare. Unendo la sua base di conoscenza a queste informazioni esterne, riesce a rispondere in maniera molto più accurata e contestualizzata. Un Prompt aumentato con l'inserimento di Chunk è formato dai seguenti elementi:

- **Query Utente:** 'Scrivi una funzione in Java che valuti la precisione dei tiri a Basket'
- **Chunk Recuperato:** 'chunk contenente una funzione che calcola la precisione dei tiri a Basket'

Inserendo all'interno del Prompt i chunk contenenti le informazioni necessarie per generare la risposta, è possibile guidare il LLM verso la soluzione. In questo modo si evita che il LLM parta da zero per arrivare alla risposta, ma parta già da una base che gli permette di generare una risposta più precisa e coerente.

Aggiunta di informazioni di sistema

In aggiunta ai chunk recuperati, è possibile inserire nel prompt delle informazioni di sistema, finalizzate a indirizzare le risposte del LLM. Queste informazioni diventano parte integrante del flusso standard del RAG e sono sempre uguali, non dipendendo dalla query inserita, e hanno lo scopo di fornire al modello ulteriori dettagli, migliorando e personalizzando la qualità dell'output. Un esempio potrebbe essere la richiesta al LLM di rispondere sempre utilizzando la valuta Dollaro.

Scrivendo le query in italiano, il LLM avrebbe proposto come valuta l'Euro, essendo la moneta utilizzata in Italia; ma con l'aggiunta di questa informazione, il modello risponderà sempre utilizzando come valuta il Dollaro.

3.3 Perchè RAG

Il più grande vantaggio dato dalla creazione di un sistema RAG è la possibilità di personalizzare le risposte del LLM senza dover intervenire direttamente sulla sua conoscenza. Questo permette di eseguire rapidamente aggiornamenti al Vector database del RAG, in modo da mantenere sempre aggiornata la base di conoscenza del modello.

Numerose applicazioni moderne dimostrano l'efficacia di questo approccio:

- I **GPTs** personalizzati di ChatGPT, che integrano documenti specifici per creare assistenti specializzati
- **Bing Chat** utilizza l'integrazione di risultati di ricerca in tempo reale per fornire risposte aggiornate e contestualizzate.
- **Framework come LangChain** permettono agli sviluppatori di costruire applicazioni basate su RAG, combinando LLM e sistemi di recupero delle informazioni per creare soluzioni personalizzate.

Le performance di un LLM addestrato su misura per le proprie esigenze sono migliori dei risultati ottenuti da un sistema RAG ma per quasi tutte le aziende questo è impossibile perchè richiede costi difficilmente sostenibili ed è per questo che i RAG sono un ottimo compromesso tra costi e benefici. Il RAG non modifica il LLM ma lo integra, permettendo di ottenere risposte personalizzate e contestualizzate senza dover intervenire direttamente sul modello. Riassumendo i vantaggi principali di un sistema RAG sono:

- permette di ottenere risposte mirate e personalizzate contenuti knowledge relativa a librerie e codice custom senza dover intervenire direttamente sul LLM

- possibilità di aggiornare rapidamente il Database con la base di conoscenza interna
- facilita l'assistenza da parte del modello nella fase di debugging migliorando la sua comprensione di sistemi complessi
- supporta la creazione di documentazione aggiornata
- permette all'interno di un Team di migliorare la coerenza del codice scritto da diversi programmatori proponendo librerie e standard comuni

Capitolo 4

Implementazione di un sistema RAG per lo sviluppo di codice per il linguaggio Java

4.1 Obiettivo

Questo caso studio si propone l'obiettivo di verificare il livello di personalizzazione e qualità delle risposte di un LLM integrato con un sistema RAG specializzato per lo sviluppo di codice Java. Potenziando la query nel Prompt di input del LLM attraverso la creazione di un sistema RAG di supporto, verranno costruite e analizzate singolarmente tutte le fasi che compongono il processo. Il sistema RAG è stato testato con delle classi JAVA custom create appositamente per il caso studio.

Problema da affrontare:

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

4.2 Architettura del Sistema

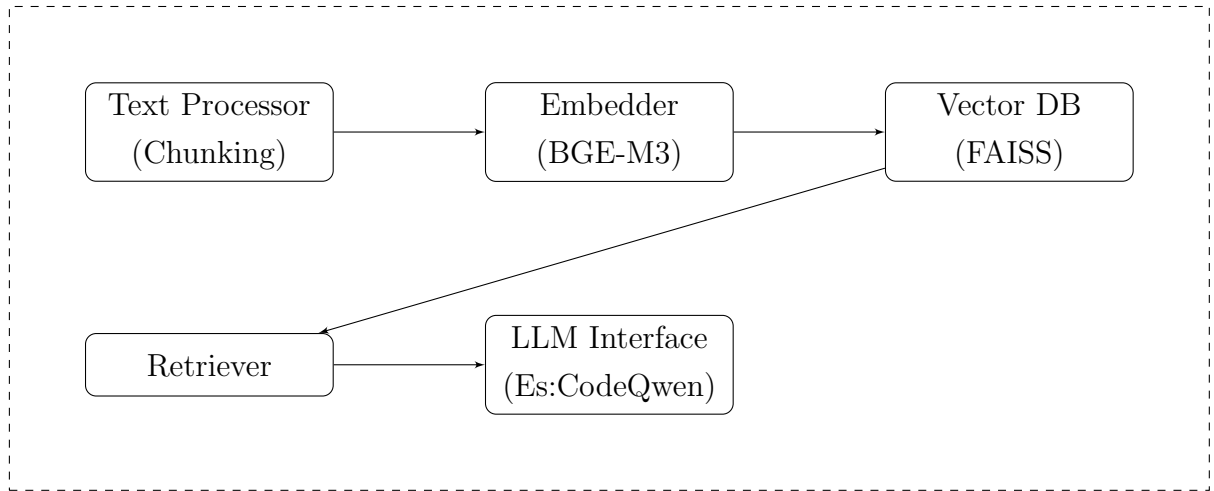


Figura 4.1: Architettura del sistema RAG

Il sistema RAG è stato progettato con un'architettura modulare che si compone di cinque componenti principali, ognuna delle quali svolge un ruolo fondamentale nel trasformare dati grezzi in risposte coerenti e contestualizzate.

4.2.1 Text Processor (Chunking)

Il primo modulo, il **Text Processor (Chunking)**, si occupa di suddividere i file Java in chunk costituiti da un numero definito di token. Questa operazione non si limita a dividere il testo in parti uguali, ma è studiata per mantenere intatto il contesto del codice. Infatti, il sistema gestisce con attenzione la sovrapposizione dei token tra i chunk adiacenti, assicurandosi che nessuna informazione rilevante venga persa durante il processo di frammentazione.

4.2.2 Embedder (BGE-M3)

La fase successiva coinvolge l'**Embedder** basato sul modello BGE-M3. Questo componente trasforma i frammenti testuali in rappresentazioni vettoriali dense attraverso un processo di embedding multivettoriale. Questi vettori vengono poi nor-

malizzati, una procedura essenziale per ottimizzare le future ricerche di similarità nel database vettoriale.

4.2.3 Vector DB (FAISS)

L'archiviazione e la ricerca efficiente sono delegate al **Vector DB FAISS** (Facebook AI Similarity Search). Questo modulo memorizza gli embedding generati, organizzandoli in un database vettoriale che permette di eseguire ricerche basate sulla similarità. Grazie a questa struttura, il sistema può recuperare in modo rapido ed efficiente i chunk più rilevanti in risposta a specifiche query, garantendo prestazioni elevate anche in presenza di grandi quantità di dati.

4.2.4 Retriever

A completare il flusso operativo, il modulo **Retriever** esegue query semantiche sul database vettoriale. Analizzando i vettori e individuando quelli che meglio rispondono ai criteri di rilevanza, il Retriever estrae i k chunk più pertinenti e li organizza per creare un contesto ricco e strutturato. Questo contesto viene poi fornito al modello di linguaggio per permettergli di generare risposte il più possibile accurate e specifiche.

4.2.5 LLM Interface

Infine, l'**LLM Interface (CodeQwen e Llama 3.2)** rappresenta il punto di interazione diretto con l'utente. Attraverso un'interfaccia gestita da Ollama, questo modulo comunica con i modelli di linguaggio CodeQwen e Llama 3.2, utilizzando il contesto prelevato dal Retriever per generare risposte personalizzate e contestualizzate. In questo modo, il sistema RAG integra e potenzia le capacità del LLM senza modificare direttamente la sua base di conoscenza. Nel complesso, l'architettura modulare del sistema RAG permette di combinare la flessibilità della generazione basata su LLM con la precisione e la rapidità dei sistemi di recupero delle informazioni, offrendo così una soluzione altamente efficace per la personalizzazione e l'aggiornamento continuo delle risposte generate.

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 Llama 3.2

Llama 3.2 3B [AI24b], un language model 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.

4.3.3 Codeqwen 1.5

Codeqwen [Tea24b] è un language model 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.

4.3.4 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.

4.3.5 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.

4.3.6 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 è stato creato appositamente per testare il sistema RAG ed è composto da diciannove classi Java. È possibile scaricare il dataset in https://github.com/ilBollo/Tesi/tree/main/my_project/classi_java_custom Il dataset è composto dalle seguenti classi Java:

DateUtilCustom.java Classe personalizzata per gestire le date

GiorniMagici.java Classe per calcolare in maniera particolare dei giorni

BasketballStats.java Classe abstract per statistiche di basket

AdvancedBasketballStats Classe che estende BasketballStats

BasketballTest Classe per testare le statistiche di basket implementate in AdvancedBasketballStats

Altre classi java Non strettamente correlate con le prime due utili per aumentare la base dati sul quale effettuare le ricerche e per testare la capacità di generalizzazione del sistema.

4.5 Risultato atteso caso d'uso RAG

DateUtilCustom.java e GiorniMagici.java sono strettamente correlate infatti GiorniMagici.java richiama metodi definiti in DateUtilCustom.java. Inizialmente andremo a testare il sistema RAG impostando come primo input 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

```
1 public static String segnaleWow(LocalDate data) {  
2     String wow = "il tuo segnale Wow e': " + DateUtilCustom.getMessaggioMagico(  
3         data);  
4     return wow;  
5 }
```

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  
2     DateTimeParseException {  
3     DayOfWeek giornoSettimana = datamagica.getDayOfWeek();  
4     switch(giornoSettimana) {  
5         case MONDAY: return "La magia inizia nel silenzio...";  
6         case TUESDAY: return "I sussurri degli antichi si fanno sentire.";  
7         case WEDNESDAY: return "Il velo tra i mondi e' sottile oggi.";  
8         case THURSDAY: return "L'energia magica e' potente e chiara.";  
9         case FRIDAY: return "Attenzione agli incantesimi del crepuscolo.";  
10        case SATURDAY: return "Il giorno perfetto per scoprire segreti nascosti.";  
11        case SUNDAY: return "Riposa e rigenera il tuo potere magico.";  
12        default: return "Il giorno e' avvolto nel mistero...";  
13    }  
14 }
```

4.5.2 Output

Il sistema RAG dovrebbe essere in grado di recuperare i chunk relativi ai metodi `segnaleWow` e `getMessaggioMagico` e di integrarli nel Prompt del LLM. Il model-

lo, basandosi su queste informazioni, dovrebbe generare una risposta coerente e contestualizzata alla query iniziale. Nel caso specifico essendo il 14 Febbraio 2025 un venerdì, la risposta corretta è:

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

4.6 Implementazione

4.6.1 Chunking del Codice Java

Per segmentare il codice Java in chunk, è stata utilizzata la libreria `langchain_text_splitters`. Durante la progettazione del sistema di chunking, sono state analizzate le caratteristiche del codice Java e sono state adottate specifiche strategie per garantire una suddivisione efficace e accurata:

- La dimensione dei chunk è stata impostata a 512 token. Questa scelta è stata fatta per evitare che metodi diversi vengano fusi nello stesso chunk, garantendo al contempo un contesto sufficientemente utile.
- È stato introdotto un overlap di 128 token tra i chunk. Questo assicura una continuità tra i chunk adiacenti, evitando la perdita di informazioni rilevanti.
- Sono stati utilizzati separatori specifici per il linguaggio Java, come `\n}\n\npublic`, `\n\nclass`, e `\n/**`. Questi separatori aiutano a preservare la struttura logica del codice.
- Sono state implementate espressioni regolari per estrarre i nomi dei metodi e delle classi dai chunk. Queste espressioni permettono di identificare i metodi e le classi presenti nel codice, arricchendo i chunk con informazioni contestuali.

Funzione process_file_Java parte 1: Inizializzazione e Configurazione

Listing 4.3: Configurazione dello splitter

```

1  def process_file_Java(file_path):
2      with open(file_path, "r", encoding="utf-8") as f:
3          lines = f.readlines()
4          text = ''.join(lines)
5          splitter = RecursiveCharacterTextSplitter(
6              chunk_size=512,
7              chunk_overlap=128,
8              separators=[      # Separatori basati sulla sintassi Java
9                  "\n}\n\npublic ",
10                 "\n}\n\nprivate ",
11                 "\n}\n\nprotected ",
12                 "\n}\n\nstatic ",
13                 "\n}\n\n// End of method",
14                 "\n\nclass ", # Inizio nuove classi
15                 "\n@",      # Annotazioni
16                 "\n/**",     # Javadoc
17                 "\n * ",
18                 "\n"
19             ],
20             keep_separator=True, # Mantiene i separatori nei chunk
21             is_separator_regex=False # Separatori letterali (non regex)
22         )

```

Nella prima parte la funzione:

- Legge il file Java e unisce il contenuto in un unico blocco di testo
- Configura RecursiveCharacterTextSplitter con:
 - Separatori specifici per costrutti Java (metodi, classi, annotazioni)
 - Token di dimensione 512 e overlap di 128
 - Mantenimento dei separatori nei chunk e uso di separatori letterali

Funzione process_file_Java parte 2: Generazione dei Chunk e Metadati

Listing 4.4: Generazione chunk e metadati

```

1  # Suddivisione del testo e calcolo metadati
2  chunks = splitter.split_text(text)
3  chunk_metadata = []
4  cursor = 0

```

```

5
6     for chunk in chunks:
7         # Calcolo righe di inizio/fine
8         start_line = text.count('\n', 0, cursor) + 1
9         chunk_length = len(chunk)
10        end_line = text.count('\n', 0, cursor + chunk_length) + 1
11
12        # Estrazione contesto semantico
13        method_name = extract_method_name(chunk)
14
15        # Registrazione metadati
16        chunk_metadata.append({
17            "start_line": start_line,
18            "end_line": end_line,
19            "text": chunk,
20            "methods": [method_name]
21        })
22        cursor += chunk_length
23
24    return chunk_metadata

```

Nella seconda parte la funzione:

- Calcola le righe di inizio/fine con logica a cursore
- Arricchisce ogni chunk con i metadati sfruttando anche la funzione `extract_method_name` per estrarre i nomi dei metodi
- Ritorna una lista di chunk con i relativi metadati

Funzione `extract_method_name`

Per arricchire ulteriormente i chunk è stata creata la funzione `extract_method_name`. Questa funzione identifica i nomi dei metodi e delle classi all'interno dei chunk utilizzando espressioni regolari (regex). In questo modo, ogni chunk può essere associato a un contesto semantico specifico, migliorando la ricerca in fase di embedding e arricchendo il Prompt del LLM con informazioni contestuali.

Listing 4.5: Estrazione contesto dai chunk

```

1     def extract_method_name(text):
2         # Pattern per la firma di un metodo in Java
3         method_pattern = r'(?:(public|private|protected|static|final|synchronized|
4         abstract|native)\s+[\w<>[\]]+\s+(\w+)\s*([^\s]*)\s*)'
5
6         # Pattern per i costruttori

```

```
5     constructor_pattern = r'(?:(public|private|protected)\s+(\w+)\s*\([^)]*\))'
6
7     # Cerca la firma di un metodo
8     matches = re.findall(method_pattern, text)
9     if matches:
10         return matches[0] # Restituisce il primo metodo trovato
11
12     # Cerca costruttori
13     constr_matches = re.findall(constructor_pattern, text)
14     if constr_matches:
15         return constr_matches[0] + " (costruttore)"
16
17     # Cerca chiamate a metodi
18     method_calls = re.findall(r'\.(\w+)\s*\(', text)
19     if method_calls:
20         return f"Chiamata a: {method_calls[-1]}"
21     return "unknown_method"
```

Metadato Class

Durante l'elaborazione dei file, è importante mantenere traccia del contesto delle classi. Questo viene fatto aggiornando dinamicamente il nome della classe corrente mentre si processano i chunk. In questo modo, ogni chunk può essere associato alla classe corretta.

Listing 4.6: Aggiornamento contesto classe

```
1 current_class = ""
2 for file_path in files:
3     chunks_info = process_file_Java(file_path)
4     for chunk_info in chunks_info:
5         if "class" in chunk_info["text"]:
6             class_name = chunk_info["text"].split("class ")[1].split("{")[0].strip()
7             current_class = class_name
8     all_chunks.append(..., "class": current_class)
```

Creazione file json

Il risultato finale del processo di chunking viene salvato in un file json chiamato `chunks.json`. Questo file contiene la lista dei chunk, ognuno dei quali comprende i metadati come il percorso del file originale, la relativa classe, i metodi identificati e le linee di codice di inizio e fine. È stato inserito di default il campo `type` per identificare il tipo di chunk, in questo caso `code`. Questo dato apre la possibilità

di estendere il sistema per supportare altri tipi di chunk, come testo libero o documentazione.

Listing 4.7: Esempio di chunk generato

```
[
  {
    "id": 1,
    "text": "package classi_java_custom;\nimport java.time.\n    LocalDate;...",
    "source": "my_project/.../AdvancedBasketballStats.java",
    "type": "code",
    "start_line": 1,
    "end_line": 14,
    "class": "AdvancedBasketballStats extends BasketballStats"
    ,
    "methods": ["calcolaEfficienzaGiocatore"]
  }
]
```

4.6.2 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 complementarità, la rappresentazione densa 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 4.8: Generazione degli embedding e creazione di un database FAISS

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

```

9      # 2. Carica il modello BGE-M3 e genera gli embedding
10     embedder = SentenceTransformer('BAAI/bge-m3')
11     embeddings = embedder.encode(
12         [
13             f"METHODS:{', '.join(c['methods']) if c['methods'] else 'unknown'}"
14             "
15             f"CLASS:{c['class']}"
16             f"LINE:{c['start_line']}-{c['end_line']}"
17             f"CONTENT:{c['text']}"
18             for c in chunks_data
19         ],
20         show_progress_bar=True
21     )
22
23     # 3. Crea un database FAISS
24     vector_store = FAISS.from_embeddings(
25         text_embeddings=list(zip(chunks, embeddings)), # Abbina testi e
26         embedding=embedding
27     )
28
29     # 4. Salva il database
30     vector_store.save_local("./faiss_db")
31     print("Database FAISS creato e salvato in ./faiss_db.")

```

Il metodo *encode* del modello BGE-M3 genera gli embedding per ogni chunk, arricchendoli con i metadati creati durante il processo di chunking. Questi embedding vengono poi utilizzati per creare un database FAISS, che permette di eseguire ricerche di similarità in modo efficiente.

4.6.3 Esecuzione di query sul Database FAISS

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

Listing 4.9: Esecuzione di una query sul database FAISS

```

1      # 1. Carica il modello di embedding nel formato corretto
2      embedder = HuggingFaceEmbeddings(
3          model_name="BAAI/bge-m3",
4          model_kwargs={'device': 'cpu'},
5          encode_kwargs={'normalize_embeddings': True}
6      )
7
8      # 2. Carica il database FAISS esistente
9      vector_store = FAISS.load_local(

```



```
10     folder_path="./faiss_db",
11     embeddings=embedder,
12     allow_dangerous_deserialization=True
13 )
14
15 # 3. Query di esempio
16 query = "Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 1, 10))?"
17
18 # 4. Cerca i chunk piu' simili
19 docs = vector_store.similarity_search_with_score(
20     query,
21     k=5,
22     score_threshold=0.90, # bassa similarita'
23     search_type="similarity", # Piu' efficace per il codice
24     lambda_mult=0.5 # Bilancia diversita'/rilevanza
25 )
26
27 # 5. Stampa i risultati con relativo score
28 for i, (doc, score) in enumerate(docs):
29     print(f"Risultato {i+1} (Score: {score:.4f}):")
30     print(doc.page_content)
31     print("-" * 40)
```

LangChain fornisce un'interfaccia semplice per eseguire query semantiche sul database FAISS.

Parametri di ricerca

- **k**: Questo parametro definisce il numero di chunk da restituire in risposta a una query. In questo caso, **k=5** restituisce, se trovati, i 5 chunk più rilevanti.
- **score_threshold**: Questo parametro definisce il valore minimo di similarità richiesto per considerare un chunk come rilevante. Il parametro **score_threshold** ripartisce i valori con una scala inversa rispetto alla cosine similarity. In questo esempio, filtrando chunk con un valore minore o uguale a 0.90 si ottengono chunk con un grado di somiglianza medio-bassa.
- **search_type**: Questo parametro definisce il tipo di ricerca da eseguire sul database FAISS. La scelta **search_type="similarity"** (similarità del coseno) è efficace per il codice per tre motivi chiave:
 - **Cerca somiglianze nella logica**, non nella forma: Ignora differenze come:

- * Nomi variabili diversi (es. `data` vs `date`)
- * Commenti o spaziature extra
- * Lunghezza del codice
- **Trova funzioni equivalenti:** Riconosce che due metodi sono simili anche se:
 - * Usano librerie diverse ma fanno la stessa cosa
 - * Hanno parametri leggermente diversi
 - * Sono scritti in stili differenti
- **Migliora la ricerca semantica:** Capisce che:
 - * `calcolaMedia()` e `getAverage()` possono essere equivalenti
 - * Un costruttore e un metodo `factory` sono correlati
 - * Un loop `for` e un loop `while` implementano la stessa logica

Chunk estratti con query base

Testiamo il risultato con una query sintetica, senza alcun riferimento esplicito 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 **0.6457**. Questo valore non è particolarmente basso 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.

Query che descrive il contesto in maniera più completa

Alla query iniziale vien aggiunto il riferimento al metodo utilizzato all'interno di segnale Wow.

- **Query:** ‘Cosa ritorna il metodo `segnaleWow(LocalDate.of(2025, 1, 10))` che utilizza la funzione `getMessaggioMagico()` della libreria `DateUtilCustom`?’
- **Output:** Fornisce i seguenti cinque chunk.
 - Primo chunk (**score: 0.5082**): contiene la funzione `segnaleWow`
 - Secondo, terzo e quarto chunk (**scores: 0.7132, 0.7611, 0.8084**): contengono la funzione `getMessaggioMagico`
 - Quinto chunk (**score: 0.8239**): funzione non rilevante relativa alle date

Conclusione

Sono stati riscontrati due problemi molto rilevanti, il primo riguarda la mancanza di estrazione di metodi da librerie esterne se non esplicitate nella query. Mentre il secondo è causato da i chunk estratti, lo score impostato non è particolarmente basso e questo può causare l'estrazione di chunk relativi a funzioni con terminologie e meccanismi simili ma non coerenti con il contesto cercato. Per questo secondo punto questa analisi ha portato alla decisione di abbassare `score_threshold` da 0.90 a 0.82, questa piccola correzione risolve in parte il problema o almeno evita di propagarlo ulteriormente, preferendo non ottenere in certi casi risultati dal RAG piuttosto che ricevere risposte non coerenti.

4.6.4 Creazione della Pipeline RAG

La pipeline RAG combina i componenti precedentemente descritti per creare un sistema completo di generazione di risposte.

Listing 4.10: Caricamento database FAISS

```
1  # Configurazione embedding
2  embedder = HuggingFaceEmbeddings(
3      model_name="BAAI/bge-m3",
4      model_kwargs={'device': 'cpu'},
5      encode_kwargs={'normalize_embeddings': True}
6  )
7  # Caricamento database FAISS
8  vector_store = FAISS.load_local(
9      folder_path="./faiss_db",
```

```

10     embeddings=embedder,
11     allow_dangerous_deserialization=True
12 )

```

Inizialmente per eseguire l'embedder della query in input viene utilizzato il modello BAAI/bge-m3 e caricato il database FAISS precedentemente creato.

Listing 4.11: Configurazione Parametri

```

1  # Creazione del retriever
2  retriever=vector_store.as_retriever(
3      search_kwargs={
4          "k": 5, # Piu' documenti per contesto
5          "score_threshold": 0.82, # medio-bassa similarita' inizialmente era
6              0.90
7          "search_type": "similarity", # Piu' efficace per il codice
8          "lambda_mult": 0.5 # Bilancia diversita'/rilevanza
9      }
10 )
11 # Comando di sistema
12 varStileLLM = "Sei un assistente che combina codice Java e contesto
13     strutturale per risposte precise."
14
15 # Configurazione Template del prompt specifici per i modelli
16 LLAMA_TEMPLATE = """<|begin_of_text|>
17 <|start_header_id|>system"" + varStileLLM + """<|end_header_id|>
18 Contesto: {context}<|eot_id|>
19 <|start_header_id|>user<|end_header_id|>
20 Domanda: {input}<|eot_id|>
21 <|start_header_id|>assistant<|end_header_id|>""
22
23 CODEQWEN_TEMPLATE = """<|im_start|>system "" + varStileLLM + ""
24 {context}<|im_end|>
25 {{ if .Functions }}<|im_start|>functions
26 {{ .Functions }}<|im_end|>{{ end }}
27 <|im_start|>user
28 {input}<|im_end|>
29 <|im_start|>assistant
30 ""
31
32 COMMON_PARAMS = {
33     "temperature": 0.3, #lasciamo una bassa creativita' non vogliamo che
34         inventi risposte
35     "top_p": 0.85 # Bilancia creativita'/controllo nei token generati
36 }

```

Il retriever viene configurato con i parametri di ricerca precedentemente descritti. Vengono inoltre definiti i template del prompt per i due modelli LLM, Llama3.2 e

Codeqwen, con il parametro `system = varStileLLM` e il tag `context` dove verranno inseriti i chunk recuperati dal database FAISS. I `COMMON_PARAMS` sono stati impostati rispettivamente a:

- **temperature:** 0.3, in modo da garantire risposte coerenti e precise senza inventarle.
- **top_p:** 0.85, per bilanciare creatività e controllo nei token generati.

varStileLLM

Come parametro di sistema da passare al LLM è stata creata la variabile `varStileLLM` con il seguente valore: **“Sei un assistente che combina codice Java e contesto strutturale per risposte precise.”** questo parametro sarà passato ad ogni Prompt generato per il LLM e condizionerà tutte le risposte fornite.

Listing 4.12: Caricamento del modello

```
1  # Caricamento modello
2  def load_model(model_name):
3      models = {
4          "llama3.2": {
5              "template": LLAMA_TEMPLATE,
6              "params": COMMON_PARAMS
7          },
8          "codeqwen": {
9              "template": CODEQWEN_TEMPLATE,
10             "params": COMMON_PARAMS
11         }
12     }
13     if model_name not in models:
14         raise ValueError(f"Modello non supportato: {model_name}")
15     return OllamaLLM(
16         model=model_name,
17         **models[model_name]["params"]
18     ), PromptTemplate(
19         template=models[model_name]["template"],
20         input_variables=["input", "context"]
21     )
22
23 # Inizializza il modello
24 llm, prompt = load_model("codeqwen")
```

`load_model` carica il modello LLM e il template del Prompt in base al modello scelto sfruttando `OllamaLLM` e `PromptTemplate`.

Listing 4.13: Catena RAG

```

1  # Catena RAG
2  document_chain = create_stuff_documents_chain(llm, prompt)
3  rag_chain = create_retrieval_chain(
4      retriever,
5      document_chain
6  )
7
8  # Funzione query
9  def ask_ollama(question):
10     try:
11         result = rag_chain.invoke({"input": question})
12         print("DOMANDA:", question)
13         print("RISPOSTA:")
14         print(result["answer"])
15         print("FONTI:")
16         for i, doc in enumerate(result["context"], 1):
17             print(f"{i}. {doc.page_content[:150]}...")
18             if 'source' in doc.metadata:
19                 print(f"   Fonte: {doc.metadata['source']}")
20             print("-" * 80)
21     except Exception as e:
22         print(f"ERRORE: {str(e)}")
23
24  # Esempio d'uso
25  if __name__ == "__main__":
26     ask_ollama("Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 2, 14)) che
27     #ask_ollama("Cosa ritorna il metodo segnaleWow(LocalDate.of(2025, 2, 14))?)")

```

La chiamata iniziale alla funzione *ask_ollama* richiede come parametro la query in input per poi essere processata dalla pipeline RAG. Sfruttando le funzionalità della libreria LangChain [Lan24b], **result** è un array contenente la risposta e il contesto 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.

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.”**

Modifica Temperature

I LLM provavano 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 i modelli rispondessero in maniera più precisa. Il risultato è stato raggiunto.

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’ prende un parametro di tipo ‘LocalDate’ e lo utilizza per richiamare la funzione ‘getMessaggioMagico’ di ‘DateUtilCustom’. Tuttavia, non è chiaro che il metodo ‘getMessaggioMagico’ restituisca sempre una stringa. Supponendo che ‘getMessaggioMagico’ ritorni sempre una stringa, il risultato del metodo ‘segnaleWow’ con l’input ‘LocalDate.of(2025, 2, 14)’ sarebbe: ”il tuo segnale Wow è :messaggio magico” Tuttavia, senza sapere cosa esattamente ritorni ‘getMessaggioMagico’, non possiamo essere certi del risultato.”

Output con LLM CodeQwen

CodeQwen continua a cercare di dare risposta senza conoscerla anche con la temperatura a 0.2, è stata quindi ulteriormente abbassata a 0.1 e `top_p` è stato abbassato a 0.6. Il modello continuava a dare risposte incomplete è stato quindi aggiunto alla variabile `varStileLLM` la stringa ‘**‘Rispondi solo se hai certezza sicura della risposta.’’**. La risposta senza allucinazioni è stata:

```
“Il metodo ‘segnaleWow’ restituisce una stringa che contiene un messaggio di ”il tuo segnale Wow è :” seguito da un messaggio magico generato dal metodo ‘getMessaggioMagico’ della classe ‘DateUtilCustom’. Per ottenere il valore del messaggio magico associato a una data specifica, ad esempio ‘LocalDate.of(2025, 2, 14)’, dovresti richiamare il metodo ‘getMessaggioMagico’ direttamente su quella data: “java String wow = DateUtilCustom.getMessaggioMagico(LocalDate.of(2025, 2, 14)); System.out.println(wow); “”
```

4.7.2 Risultati con query che descrive accuratamente il contesto

Rieseguiamo il test ma con una query più dettagliata aggiungendo i riferimenti al metodo utilizzato all’interno della funzione `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 4 chunk, il primo che riporta la funzione `segnaleWow` e gli altri tre chunk sono 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. Tuttavia, il messaggio magico è generato dalla funzione ‘getMessaggioMagico’ della libreria ‘DateUtilCustom’, che richiede un oggetto ‘LocalDate’. Il problema è che il metodo ‘segnaleWow’ non restituisce direttamente il messaggio magico, ma piuttosto una stringa che contiene il messaggio magico. Quindi, per ottenere il messaggio magico effettivamente, dobbiamo chiamare entrambi i metodi. Ecco la soluzione: “java public static String segnaleWow(LocalDate date) return ”Il tuo segnale Wow è: ” + getMessaggioMagico(date); “ Quindi, se chiamiamo ‘segnaleWow’ con una data specifica, otteniamo il messaggio magico come risultato: “java LocalDate date = LocalDate.of(2025, 2, 14); String messaggioMagico = segnaleWow(date); System.out.println(messaggioMagico); // ”Il tuo segnale Wow è: Il giorno perfetto per scoprire segreti nascosti.” “ In questo modo, otteniamo il messaggio magico effettivamente.”

Output con LLM CodeQwen

“Il metodo ‘segnaleWow’ restituisce una stringa che contiene il messaggio magico associato alla data specificata. In questo caso, se si utilizza la funzione ‘getMessaggioMagico(LocalDate.of(2025, 2, 14))’, la data è un lunedì (14 febbraio 2025). Il metodo ‘getMessaggioMagico’ restituisce il messaggio magico associato al giorno della settimana del parametro ‘datamagica’. Nel caso di una data lunedì, il messaggio restituito sarà ”La magia inizia nel silenzio...”. Quindi, la risposta del metodo ‘segnaleWow(LocalDate.of(2025, 2, 14))’ sarà: “ il tuo segnale Wow è : La magia inizia nel silenzio...”

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 correttamente soprattutto se si scrive la richiesta in maniera più dettagliata possibile. I LLM hanno sbagliato a calcolare il giorno della settimana a causa di un loro errore nel calcolare il giorno della settimana corrispondente alla data fornita come

parametro. Ma le risposte fornite utilizzano correttamente il metodo `getMessaggioMagico`. Sono stati migliorati i parametri di ricerca per evitare allucinazioni da parte dei modelli. Eseguendo questo test utilizzando versioni di questi modelli con più parametri, si è ottenuto il risultato corretto.

Esempio utilizzando qwen con 70 bilioni di parametri

Testato su <https://deepinfra.com/Qwen/Qwen2.5-Coder-7B> dove è stato possibile settare anche gli stessi parametri al modello, ovvero temperature 0.1 e top_p 0.6.

```
“ Il metodo ‘segnaleWow(LocalDate.of(2025, 2, 14))’ ritorna
una stringa che contiene il messaggio magico corrispondente
alla data fornita. In questo caso, la data è il 14 febbraio
2025, che è un venerdì. Quindi, il metodo ‘getMessaggioMagico(LocalDate
datamagica)’ restituirà il messaggio "Attenzione agli incantesimi
del crepuscolo." Quindi, il metodo ‘segnaleWow(LocalDate.of(2025,
2, 14))’ restituirà la stringa: “ il tuo segnale Wow è :Attenzione
agli incantesimi del crepuscolo. “ Sicuro della risposta.”
```

La risposta è corretta, è possibile concludere che il sistema RAG creato funzioni complessivamente in modo corretto.

4.8 Generazione codice Java specifico con RAG

Valutiamo ora la capacità del sistema RAG di generare codice Java, verificando se il sistema RAG è in grado di aiutare i modelli LLM a personalizzare le risposte, per fare in modo che il LLM produca codice coerente con quanto già realizzato e conosciuto. Per questi test è stato utilizzato il LLM *CodeQwen 7B* essendo tra i due presi in considerazione quello maggiormente addestrato per lo sviluppo di codice e dal quale ci si aspetta risultati migliori. Sono state eseguite le seguenti tipologie di test specifici:

- Estensione di classi esistenti
- Refactoring di metodi
- Implementazione di nuove funzionalità usando metodi di classi esistenti

4.8.1 Test 1: Estensione di una Classe

Query:

“Aggiungi alla classe `AdvancedBasketballStats` un metodo `calcolaEfficienzaSquadra()` che valuti l’efficienza della squadra in base alle statistiche della squadra e alle partite vinte”

Codice generato con RAG: Il LLM ha riportato anche parte della classe `AdvancedBasketballStats`.

```
1 package classi_java_custom;
2 import java.time.LocalDate;
3 public class AdvancedBasketballStats extends BasketballStats {
4
5     /** Override con calcolo avanzato
6     */
7     @Override
8     public double calcolaEfficienzaGiocatore(int punti, int rimbalzi, int assist,
9                                             int pallePerse, int tiriSbagliati) {
10         return (punti * 1.2 + rimbalzi * 1.1 + assist * 1.3) -
11             (pallePerse * 0.9 + tiriSbagliati * 0.7);
12     }
13
14     /** Aggiunto metodo per calcolo efficienza squadra
15     */
16     public double calcolaEfficienzaSquadra(int[] statisticheGiocatori) {
17         double totalPunti = 0;
18         double totalRimbalzi = 0;
19         double totalAssist = 0;
20         double totalPallePerse = 0;
21         double totalTiriSbagliati = 0;
22         for (int stat : statisticheGiocatori) {
23             totalPunti += stat / 5; // Assumendo che ogni statistica e' un punto,
24                 rimbalzo, assist, palle perse o tiri sbagliati
25             totalRimbalzi += stat / 4;
26             totalAssist += stat / 3;
27             totalPallePerse += stat / 2;
28             totalTiriSbagliati += stat / 1.5;
29         }
30         return calcolaEfficienzaGiocatore(totalPunti, totalRimbalzi, totalAssist,
31             totalPallePerse, totalTiriSbagliati);
32     }
33 }
```

Analisi: Il RAG fornisce al Prompt la classe specifica che permette al LLM di generare codice Java coerente con lo stile e che sfrutta i metodi già presenti nella

classe. Infatti nel metodo `calcolaEfficienzaSquadra` generato viene utilizzato il metodo `calcolaEfficienzaGiocatore` già presente nella classe. Le variabili sono coerenti sia per significato che tipologia unico errore è stato creare variabili di tipo `double` invece che `int`. Complessivamente la risposta è coerente con quanto atteso e facilmente integrabile con il codice già presente.

4.8.2 Test 2: Refactoring di un Metodo

Query:

“Sostituisci nella classe `DateUtilCustom.java` il costrutto `if-else` in `getMessaggioMagico()` con uno `switch expression`”

Codice generato con RAG:

```
1      /**
2      * Ottieni un messaggio magico in base al giorno della settimana di una data.
3      * @param datamagica
4      * @return Il messaggio magico.
5      * @throws DateTimeParseException Se la data non e' valida.
6      */
7      public static String getMessaggioMagico(LocalDate datamagica) throws
8          DateTimeParseException {
9          DayOfWeek giornoSettimana = datamagica.getDayOfWeek();
10
11         return switch (giornoSettimana) {
12             case MONDAY -> "Lunedì, il mondo e' in pausa per un lunedì";
13             case TUESDAY -> "Martedì, la settimana comincia!";
14             ...continua correttamente
15         };
16     }
```

Analisi: Anche in questo caso il RAG fornisce al prompt il metodo specifico da modificare. Il refactoring è stato eseguito correttamente e il codice generato è facilmente integrabile con il codice già presente.

4.8.3 Test 3: Implementazione di nuove funzionalità usando metodi di classi esistenti

Query:

“Sto costruendo il seguente metodo: `int calcolaGiorniLavorativi(Date dataA, Date dataB) int numeroGiorni...` return numeroGiorni che trova la differenza in giorni lavorativi(escludendo sabato e domenica). Come posso costruirlo utilizzando la classe `DateUtilCustom`?”

Risposta e codice generato con RAG: “Per calcolare i giorni lavorativi tra due date, puoi utilizzare il metodo ‘`getDifferenceInWorkingDays`’ della classe ‘`DateUtilCustom`’. Questo metodo restituisce il numero di giorni lavorativi compresi tra le due date specificate, escludendo sabati e domeniche. Ecco come implementare il metodo ‘`calcolaGiorniLavorativi`’ utilizzando la classe ‘`DateUtilCustom`’:

```
1 import java.util.Date;
2
3 public class Main {
4     public static void main(String[] args) throws ParseException {
5         // Creazione di due date
6         SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy");
7         Date dataA = formatter.parse("01/01/2023");
8         Date dataB = formatter.parse("31/12/2023");
9
10        // Calcolo dei giorni lavorativi tra le due date
11        int numeroGiorniLavorativi = calcolaGiorniLavorativi(dataA, dataB);
12        System.out.println("Numero di giorni lavorativi: " +
13            numeroGiorniLavorativi);
14    }
15    public static int calcolaGiorniLavorativi(Date dataA, Date dataB) {
16        return (int) DateUtilCustom.getDifferenceInWorkingDays(dataA, dataB);
17    }
18 }
```

Nell’esempio di codice qui sopra, viene utilizzato il metodo ‘`getDifferenceInWorkingDays`’ della classe ‘`DateUtilCustom`’ per calcolare la differenza in giorni lavorativi tra le due date specificate. Il risultato viene convertito in un intero e restituito dal metodo ‘`calcolaGiorniLavorativi`’. ” **Analisi:** Nel Prompt sono state fornite le informazioni necessarie per implementare il metodo richiesto, passando i vari metodi della classe `DateUtilCustom`. Ma il LLM non è riuscito a generare un codice corretto, infatti il metodo `calcolaGiorniLavorativi` non è stato implementato correttamente avendo utilizzato una funzione non esistente `getDifferenceInWorkingDays`.

4.8.4 Conclusioni

Il LLM dimostra notevoli capacità di contestualizzare le risposte se passate adeguate informazioni dal RAG al Prompt. Tuttavia le conoscenze passate non sono sempre sufficienti per dare al modello il giusto contesto per generare codice coerente, il RAG per fornire il giusto contesto ha bisogno di richieste mirate e di piccole dove riesce estrarre i chunk più rilevanti. Non è invece in grado di generare codice parti di codice Java articolate per le quali avrebbe bisogno di un contesto più ampio. Nel RAG in fase di costruzione del Template da passare al Prompt come *system* si potrebbe creare un testo bene strutturato con le best practice più importanti per il programmatore da passare che saranno ogni volta fornite al LLM. Soluzione migliore potrebbe essere costruire un LLM specializzato nella gestione dei RAG direttamente in fase di fine-tuning, preparandolo a gestire le richieste del RAG. Inoltre il RAG dovrebbe fornire un contesto molto più ampio al Prompt al LLM, questo al giorno d'oggi sta già avvenendo in modelli come GitHub Copilot che permette di avere un contesto di classi molto ampio al LLM per generare le sue risposte.

4.9 Valutazione “llm as a judge”

Per eseguire una valutazione più ampia del sistema viene ora eseguito l'approccio “*llm as a judge*” per valutare automaticamente quanto prodotto dal sistema RAG. Sono state generate **30 domande** sulle quali sarà richiesta risposta al sistema RAG e successivamente eseguita una valutazione automatizzata delle risposte prodotte da parte di un altro LLM.

4.9.1 Creazione domande

Passando un file contenente tutte le librerie a **NotebookLM**, sono state generate 30 domande per valutare il sistema RAG. La richiesta 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 risultato è strutturato nel seguente modo:

Listing 4.14: Domande/Risposte generate da NotebookLM

```
{
  "question1": "Cosa ritorna il metodo 'segnaleWow(LocalDate
.of(2025, 2, 14))' che utilizza la funzione '
getMessaggioMagico()' della libreria 'DateUtilCustom'?",
  "answer": "Ritorna la stringa \"il tuo segnale Wow e':
Attenzione agli incantesimi del crepuscolo.\""
},
{
  "question2": "La classe 'AnalizzatoreRilascio' contiene un
metodo chiamato 'stimaDataRilascio'. Quali sono i due
parametri di input richiesti da questo metodo?",
  "answer": "Il metodo 'stimaDataRilascio' richiede un array
di interi ('int[] taskCompletati') e un valore double ('
double velocitaSviluppo') come input."
},
}
```

4.9.2 Metrica del punteggio

Per valutare le domande generate da **LMNotebook**, 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. Su una scala da 1 a 5, dove 1 significa che la domanda è risolvibile anche senza conoscere il contesto specifico, mentre 5 quando la domanda è chiaramente e inequivocabilmente risolvibile solo conoscendo il contesto.

Risultati

I modelli hanno valutato le domande generate da **LMNotebook** con punteggi diversi:

- **Mistral**: 145 su 150

- **GPT4o**: 121 su 150

GPT4o ha fornito anche una tabella riepilogativa con i punteggi di ogni domanda nel dettaglio hanno questa corrispondenza:

- Domande con punteggio 5: Non risolvibili senza accesso al codice/documentazione.
- Domande con punteggio 4: Difficili, ma in alcuni casi il modello può ipotizzare una risposta sensata.
- Domande con punteggio 3: Risolvibili parzialmente con conoscenze standard, ma con margini di errore.
- Domande con punteggio 2: Generalmente risolvibili perché rientrano in pattern comuni di programmazione.

4.9.3 llm as a judge

Per poter valutare massivamente le domande alla pipeline rag sono state aggiunte le seguenti funzioni:

Listing 4.15: funzione load_questions

```
1 def load_questions(file_path: str) -> List[Dict]:
2     try:
3         actual_path = Path(__file__).parent / file_path
4         with actual_path.open('r', encoding='utf-8') as f:
5             data = json.load(f)
6
7         # Validazione della struttura
8         required_keys = {'id', 'question', 'answer', 'punteggio'}
9         for item in data:
10             if not required_keys.issubset(item.keys()):
11                 raise ValueError("Struttura JSON non valida")
12         return data
13
14     except FileNotFoundError:
15         raise Exception(f"File {file_path} non trovato")
16     except json.JSONDecodeError:
17         raise Exception("Errore nel parsing del JSON")
```

load_question carica le domande dal file JSON e ne verifica la struttura.

Listing 4.16: Funzione:process_questions

```

1 def process_questions(questions: List[Dict]) -> List[Dict]:
2     results = []
3     for q in questions:
4         try:
5             result = rag_chain.invoke({"input": q["question"]})
6             entry = {
7                 "id": q["id"],
8                 "question": q["question"],
9                 "answerOK": q["answer"], # Mantiene il contesto originale
10                "answerRAG": result["answer"],
11                "punteggio": q["punteggio"],
12                "sources": [
13                    {
14                        "content": doc.page_content[:50]
15                    }
16                    for doc in result["context"]
17                ]
18            }
19            results.append(entry)
20            print(f"Processata {q['id']}")
21
22        except Exception as e:
23            print(f"Errore su {q['id']}: {str(e)}")
24            results.append({
25                "id": q["id"],
26                "punteggio": q["punteggio"],
27                "error": str(e),
28                "question": q["question"]
29            })
30    return results

```

`process_questions` esegue la valutazione delle domande e restituisce i risultati.

4.9.4 Valutazione risposte

Il sistema RAG ha elaborato le domande e fornito risposta salvando il risultato in un file JSON: Questi file sono stati valutati da **DeepSeek**, il quale ha assegnato un punteggio specifico per ogni domanda confrontando le risposte `answerOK` e `answerRAG`. Sono state effettuate quattro elaborazioni con configurazioni differenti ottenendo i seguenti punteggi:

- CodeQwen con `score_threshold` 1.0 : 109/124
- Llama3.2 con `score_threshold` 0.8 : 87/124

- Llama3.2 con score_threshold 1.0 : 97/124
- Llama3.2 con score_threshold 0.4 : 28/124

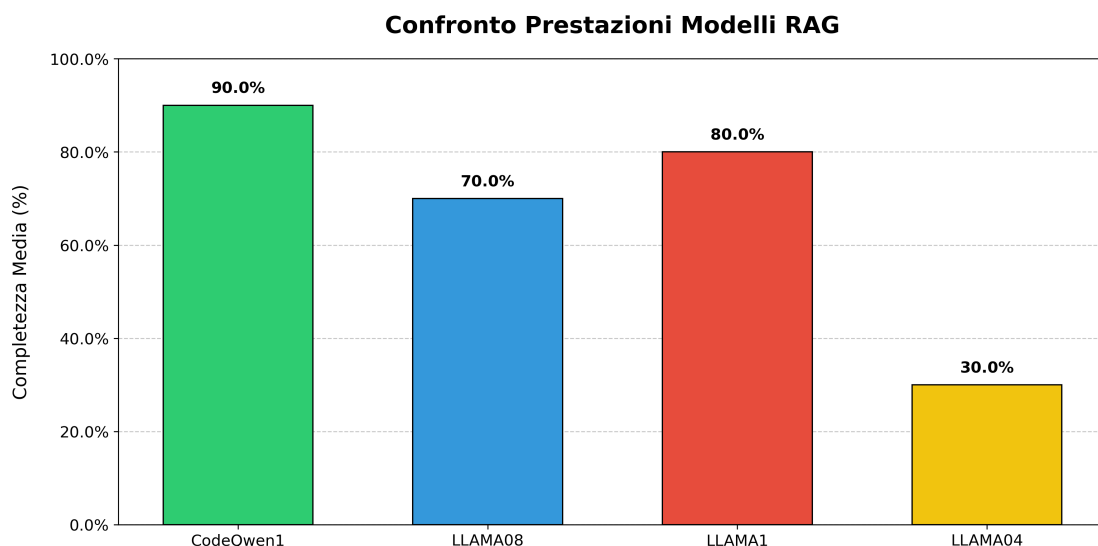


Figura 4.2: Confronto risposte modelli RAG

4.9.5 Analisi risultati

La configurazione migliore è stata quella del LLM **CodeQwen** con **score threshold 1.0**, i test effettuati hanno dimostrato che è meglio aumentare la tolleranza di score dei chunk estratti nella fase di Retriever. Questo score relativamente alto, come già analizzato nei capitoli precedenti, può causare l’inserimento di alcuni chunk non coerenti ma allo stesso tempo evita di escludere chunk validi e fondamentali per permettere al LLM di rispondere correttamente. Quando il RAG trova i chunk corretti abbiamo dimostrato che `score_threshold` ha valore molto basso solo nei casi in cui la query in input è scritta in maniera molto accurata cosa che normalmente non avviene. Il risultato di questa minore rigidità nel filtrare i chunk può essere utile per ridurre l’overfitting creato nella prima fase di configurazione dei parametri dell RAG. Per validare la soluzione adottata e prevenire il plorificarsi di chunk non coerenti viene sempre impostato il limite a 5 Chunk. In questo

modo il Prompt non viene mai eccessivamente appesantito e il modello riesce ad elaborare nella maggior parte dei casi risposte coerenti. I test fatti su Llama3.2 con score threshold 1.0 e con threshold 0.8 esplicitano queste conclusioni infatti il risultato migliore è stato con score_threshold più alto perchè in questo modo non sono stati estromessi nella fase di Retriever chunk validi e fondamentali per la risposta. I risultati sono stati nel complesso soddisfacenti ed è possibile concludere che il sistema RAG funziona correttamente aumentando le conoscenze dei LLM in modo vincente. Ulteriore test è stato fatto impostando uno score_threshold 0.4 con il quale vengono esclusi di fatto quasi tutti i chunk, in questo modo il LLM risponde solo utilizzando le sue conoscenze riportando risultati estremamente scarsi non conoscendo i contesti e le librerie specificate nelle domande.

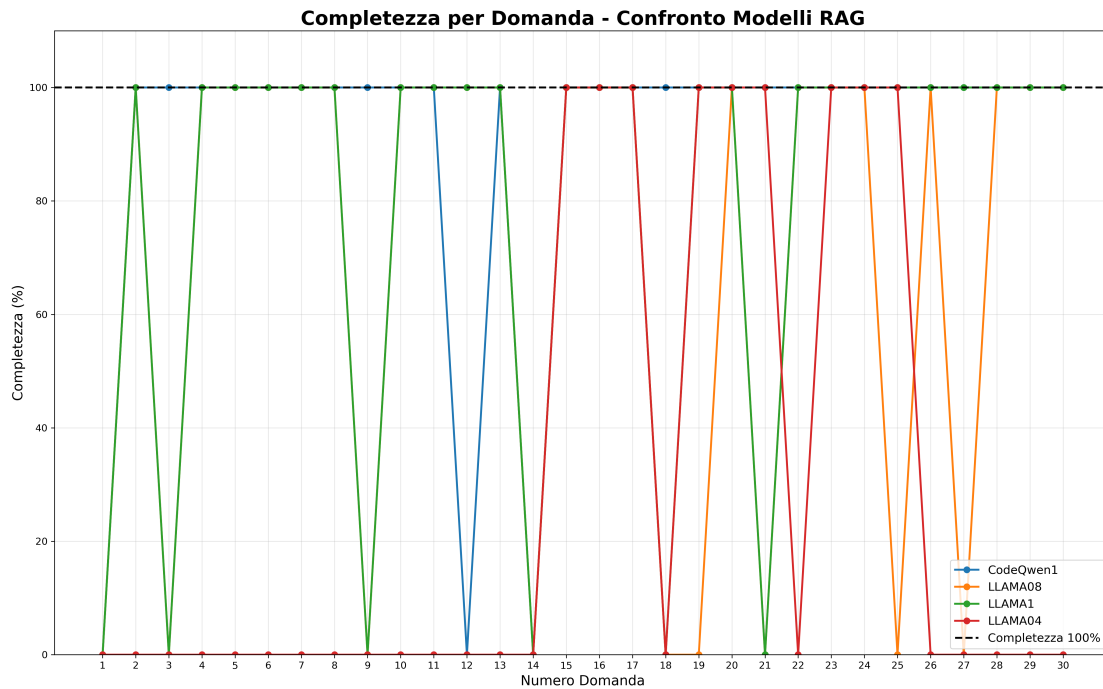


Figura 4.3: Completezza per domanda

Nota: È interessante ricordare che tutti i modelli hanno sbagliato la prima domanda, non per mancanza di contesto nei chunk forniti al Prompt, ma come precedentemente descritto a causa di un errore nel calcolo del giorno della settimana.

Questo errore non si sarebbe verificato con le versioni con più parametri degli stessi modelli.

Capitolo 5

Conclusioni

“Let an ultraintelligent machine be defined as a machine that can far surpass all the intellectual activities of any man however clever. Since the design of machines is one of these intellectual activities, an ultraintelligent machine could design even better machines; there would then unquestionably be an ‘intelligence explosion,’ and the intelligence of man would be left far behind. Thus the first ultraintelligent machine is the last invention that man need ever make.”

— *I.J. Good (1965)*[Goo65]

Questi mesi di lavoro mi hanno portato a cercare di conoscere e capire il più possibile come funzionano realmente i Large Language Model. Nella fase di ricerca ho sempre trovato nei vari articoli e analisi stupore e meraviglia nel commentare i risultati e le capacità dimostrate da questi modelli. Il saggio di Leopold Aschenbrenner [Fac24c] “SITUATIONAL AWARENESS: The Decade Ahead”, da me incontrato quasi al termine della scrittura di questa tesi, spiega e approfondisce lo stesso giudizio che oggi ho sul futuro dell’Artificial intelligence. Non voglio addentrarmi ora in ulteriori giudizi generali sul futuro dell’AI preferendo tornare nel contesto di questa tesi e affermare che **l’integrazione di RAG e LLM nello sviluppo del Software** è oggi già piena realtà. Il sistema RAG d’esempio implementato funziona correttamente nonostante i limiti dei modelli utilizzati e con un’ulteriore implementazione delle tecniche Chunking potrebbe essere realmente utilizzato a livello aziendale con buoni risultati. Implementare sistemi RAG per le

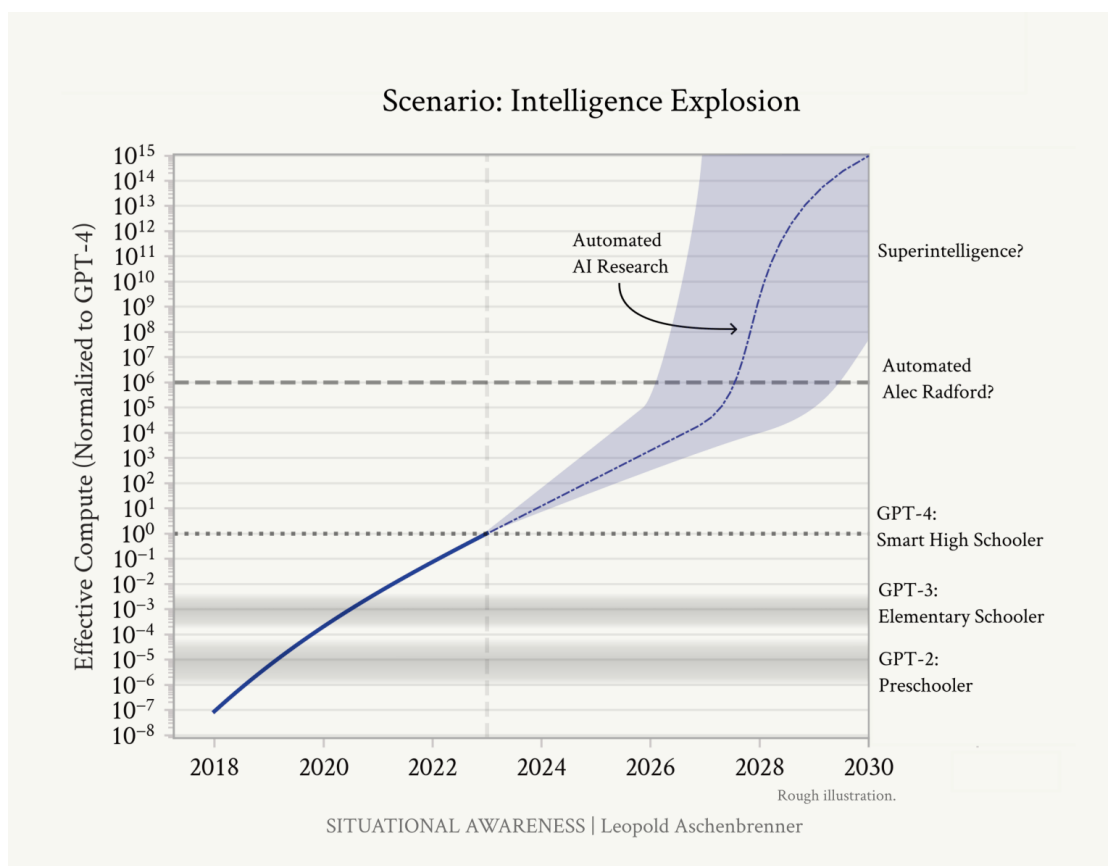


Figura 5.1: fig: tratta da situational-awareness.ai

aziende oggi è un ottima soluzione, ma resta sempre il peso e la consapevolezza che in questo momento, le soluzioni costruite, sono sempre un passo indietro rispetto ai miglioramenti globali che giornalmente vengono rilasciati. Come programmatore userò questi strumenti nel mio lavoro marginalmente perchè è mio desiderio continuare ad avere il controllo dei progetti realizzati che sento spesso di perdere utilizzando questi strumenti. Altro motivo e per continuare ad avere soddisfazione e orgoglio nel trovare soluzioni in autonomia, certo sono consapevole che permettendo di velocizzare e migliorare la qualità del mio codice facendomi conoscere e ragionare su nuove tecniche e soluzioni aumentando la mia base di conoscenza. L'espressione *“Se la ruota è già stata inventata, perchè reinventarla”* invita a non ripetere un lavoro già svolto con successo, ma è anche vero che il codice è stato

inventato e migliorato da tanti programmatori che hanno cercato di renderlo più adatta ai loro scopi. Proprio per questo mi piace concludere pensando che noi programmatori (semplicemente) stiamo imparando da altri programmatori che non incontreremo e conosceremo mai di persona e l'AI sia, in questo campo, (solo) un grandissimo trasmettitore di conoscenza.

“De nihilo nihil”

— *Lucrezio* (55 a.c)

Capitolo 6

Ringraziamenti

Ringrazio il mio relatore il Prof. Viroli Mirko e il Dott. Aguzzi Gianluca per l'interessantissimo argomento di tesi proposto e per la disponibilità e professionalità dimostrata. Ringrazio tutta la comunità di ricercatori che forniscono materiale open source e documentazione per permettere a tutti di apprendere e migliorare questa incredibile tecnologia. Grazie a tutti gli studenti che ho conosciuto in questo percorso, è stato molto costruttivo e divertente studiare con voi. Infine vorrei trovare le parole giuste per ringraziare Giulia, in questi anni sei stata eccezionale grazie per tutto, questo risultato è in gran parte merito tuo.

Bibliografia

- [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.

- [Fac24c] Hugging Face. Situational awareness:the decade ahead, 2024. URL: <https://situational-awareness.ai/>.
- [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/>.
- [Goo65] Irving John Good. *Speculations Concerning the First Ultraintelligent Machine*, page 31–88. Elsevier, 1965. URL: [http://dx.doi.org/10.1016/S0065-2458\(08\)60418-0](http://dx.doi.org/10.1016/S0065-2458(08)60418-0), doi: 10.1016/s0065-2458(08)60418-0.
- [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.
- [Lan24c] LangChain. Langchain retriever documentation, 2024. Documentation for LangChain retriever module. URL: [https://v03.api.js.langchain.com/classes/langchain.retrievers_score_threshold.ScoreThresholdRetriever.html](https://v03.api.js.langchain.com/classes/langchain.retrievers.score_threshold.ScoreThresholdRetriever.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.