ZUCCHETTI

Specifica Tecnica

Gini

Integrazione di LLM in Git lato versionamento

Leonardo Trolese

Indice

1 Introduzione

1.1 Scopo del documento

Questo documento descrive l'architettura del software sviluppato nell'ambito del progetto di stage presso Zucchetti S.p.A., svolto dallo studente Leonardo Trolese. La specifica tecnica è rivolta a fornire una visione strutturata e dettagliata delle componenti del sistema, delle interazioni tra i moduli e delle scelte tecnologiche adottate. Essa costituisce un riferimento per lo sviluppo, il testing, la manutenzione e l'estensione futura del plugin realizzato.

1.2 Obiettivi del progetto

L'obiettivo principale del progetto è la realizzazione di un plugin per Git capace di interpretare richieste espresse in linguaggio naturale e generare comandi Git corrispondenti tramite l'uso di modelli LLM (Large Language Model). Il sistema integrerà tecniche di Retrieval-Augmented Generation (RAG) e ricerca ibrida basata su dense vector embedding e ranking BM25. Inoltre, saranno sviluppate funzionalità per:

- la generazione automatica di commenti alle commit;
- l'analisi dell'impatto delle modifiche sul repository;
- la gestione assistita dei conflitti di merge.

Le attività saranno accompagnate da sessioni di test e dalla produzione di documentazione tecnica, al fine di assicurare l'affidabilità e la riusabilità della soluzione finale.

1.3 Glossario

- **LLM**:: Large Language Model – modelli linguistici di grandi dimensioni utilizzati per comprendere e generare testo in linguaggio naturale.
- **Git**:: Sistema di controllo versione distribuito utilizzato per tracciare modifiche nel codice sorgente.
- **Plugin Git**:: Estensione di funzionalità per Git, integrata come modulo aggiuntivo.
- **RAG**:: Retrieval-Augmented Generation – tecnica per potenziare le risposte di un LLM mediante l'accesso a documenti rilevanti.
- **Embedding**:: Rappresentazione vettoriale di dati (es. parole, frasi) usata per il confronto semantico tra contenuti.
- **BM25**:: Algoritmo di ranking basato su term frequency per l'indicizzazione e il recupero di informazioni testuali.
- **CLI**:: Command-Line Interface – interfaccia testuale per l'interazione con il software.
- **Commit**:: Salvataggio di uno stato del codice in un repository Git.
- **Merge conflict**:: Conflitto tra modifiche concorrenti in Git che devono essere risolte manualmente o automaticamente.

2 Tecnologie

Questa sezione descrive le tecnologie impiegate nello sviluppo del progetto, con particolare attenzione agli strumenti software, ai linguaggi di programmazione e alle librerie che compongono l'ecosistema

applicativo. La scelta di queste tecnologie è stata guidata da criteri di compatibilità, efficienza, supporto alla modularità e disponibilità di risorse open source.

2.1 Stack tecnologico

Il progetto è stato sviluppato all'interno di un ambiente Python, con il supporto di strumenti e librerie open source orientati all'elaborazione del linguaggio naturale, alla generazione di codice e alla manipolazione di repository Git. L'architettura del sistema prevede l'integrazione di modelli LLM (Large Language Models), l'impiego di tecniche di Retrieval-Augmented Generation (RAG) e l'uso di database vettoriali per la ricerca semantica dei contenuti.

Oltre a Python come base di sviluppo, il sistema si affida a componenti modulari per la gestione della configurazione, il testing, la documentazione e l'interfaccia a riga di comando. In particolare, l'interazione con Git avviene tramite interfacce programmatiche, mentre i modelli LLM operano in locale attraverso strumenti containerizzati. Le fasi di testing e documentazione sono integrate nel flusso di lavoro del progetto per garantire qualità e tracciabilità.

2.2 Linguaggi utilizzati

Nel corso del progetto sono stati utilizzati i seguenti linguaggi:

- **Python**: linguaggio principale impiegato per lo sviluppo del plugin e delle sue componenti interne. La scelta è motivata dalla vasta disponibilità di librerie per l'IA, l'elaborazione del linguaggio naturale e l'interazione con Git.
- **JSON**: formato di serializzazione leggero utilizzato per la configurazione dell'applicazione e la definizione di dati strutturati, come input di esempio o risposte attese.

2.3 Framework e librerie

Il sistema integra numerosi pacchetti Python e strumenti software open source, ciascuno con un ruolo specifico all'interno del flusso operativo:

- **chromadb**: database vettoriale utilizzato per la memorizzazione e la ricerca di rappresentazioni semantiche (embedding) di documenti testuali. È un componente chiave per la fase di retrieval nel contesto RAG.
- **rank_bm25**: implementazione dell'algoritmo BM25, impiegata per il ranking dei documenti recuperati sulla base della rilevanza testuale. Viene utilizzata in combinazione con le ricerche vettoriali per aumentare la precisione.
- **ollama**: strumento che fornisce un'interfaccia semplice per l'utilizzo di modelli LLM localmente. Consente di eseguire inferenze senza dipendere da servizi cloud esterni, garantendo maggiore controllo e privacy.
- **GitPython**: libreria che permette di interagire in modo programmatico con repository Git. È fondamentale per l'automazione delle operazioni di commit, merge e generazione di commenti.
- **attrs**: utilizzata per la definizione concisa e robusta di classi dati in Python, facilitando la scrittura di codice manutenibile e ben strutturato.
- **PyMuPDF**: libreria opzionale utilizzata per il parsing e la gestione di file PDF, utile in eventuali casi di estrazione o analisi di contenuti documentali.

- **colorama**: impiegata per la gestione della colorazione dell'output su terminale, migliorando l'esperienza utente durante l'utilizzo del plugin in modalità CLI.
- **pytest**, **pytest-cov**: strumenti dedicati al testing automatico delle componenti software. Consentono l'esecuzione di test unitari e la valutazione della copertura del codice, a supporto della qualità del progetto.

L'adozione di queste tecnologie ha permesso di costruire un sistema robusto, flessibile e facilmente estendibile, in linea con gli obiettivi di progetto e le pratiche moderne di sviluppo software.

3 Architettura di Sistema

3.1 Panoramica del Sistema

Il sistema è strutturato come un plugin Git che opera attraverso un'interfaccia a riga di comando (CLI), implementando un'architettura modulare basata su componenti specializzati. Il flusso di interazione principale si articola nelle seguenti fasi:

1. **Input Processing**: l'utente inserisce una query in linguaggio naturale attraverso la CLI. Il Plugin

Manager interpreta il comando e lo instradata al componente appropriato.

2. **Query Understanding**: il Sistema di Query analizza la richiesta dell'utente e la prepara per

l'elaborazione RAG, identificando il contesto e gli elementi chiave della query.

3. **Knowledge Retrieval**: la Pipeline RAG:
   - converte i documenti in embedding attraverso l'Embedding Handler
   - recupera le informazioni rilevanti dal database vettoriale
   - applica tecniche di ranking ibrido (BM25 + similitudine vettoriale)
4. **Response Generation**: il modello LLM genera una risposta strutturata combinando:
   - il contesto recuperato dalla knowledge base
   - la query dell'utente
   - le informazioni sullo stato corrente del repository Git
5. **Command Execution**: il sistema traduce la risposta generata in comandi Git eseguibili, gestendo:
   - la validazione dei comandi
   - l'esecuzione sicura
   - la gestione degli errori
   - il feedback all'utente

3.2 Componenti Principali

3.2.1 Plugin Manager

Il Plugin Manager rappresenta il punto di ingresso principale del sistema e svolge le seguenti funzioni chiave:

- **Validazione dell'Ambiente**: verifica che il comando venga eseguito all'interno di un repository Git valido

- **Gestione Input CLI**: processa gli argomenti della riga di comando per determinare l'azione richiesta
- **Orchestrazione**: indirizza le richieste al componente Query appropriato quando viene invocato il comando 'cmd'
- **Gestione Errori**: fornisce feedback all'utente attraverso messaggi colorati in console per una migliore esperienza utente

Il componente agisce come intermediario tra l'interfaccia CLI e il core del sistema, garantendo che tutte le precondizioni necessarie siano soddisfatte prima di procedere con l'elaborazione delle richieste.

3.2.2 Sistema di Query

Il Sistema di Query rappresenta il core dell'applicazione, responsabile dell'elaborazione delle richieste dell'utente e della generazione delle risposte appropriate. Implementa un pattern State per gestire il flusso di elaborazione e supporta diverse modalità di interazione attraverso una struttura flessibile e modulare.

Il sistema si articola in tre stati principali che gestiscono l'intero ciclo di vita di una query: InitialState per l'avvio del processo RAG e l'inizializzazione del contesto, ExecutionState per la gestione dei comandi Git generati, e RefinementState per eventuali correzioni basate sul feedback dell'utente. Questa architettura a stati permette una gestione fluida delle interazioni e garantisce la coerenza delle operazioni durante l'intero processo di elaborazione.

La pipeline di elaborazione opera attraverso diverse fasi sequenziali: inizia con l'analisi della query in linguaggio naturale, procede con il recupero del contesto rilevante tramite RAG, e culmina nella generazione di una risposta strutturata tramite LLM. Durante questo processo, il sistema mantiene una stretta integrazione con il repository Git, consentendo l'accesso in tempo reale allo stato del progetto e alla sua storia.

Le risposte generate vengono strutturate in formato JSON, includendo sia una spiegazione dettagliata che il comando Git proposto. Prima dell'esecuzione, ogni comando attraversa una fase di validazione sintattica e semantica per garantire operazioni sicure sul repository.

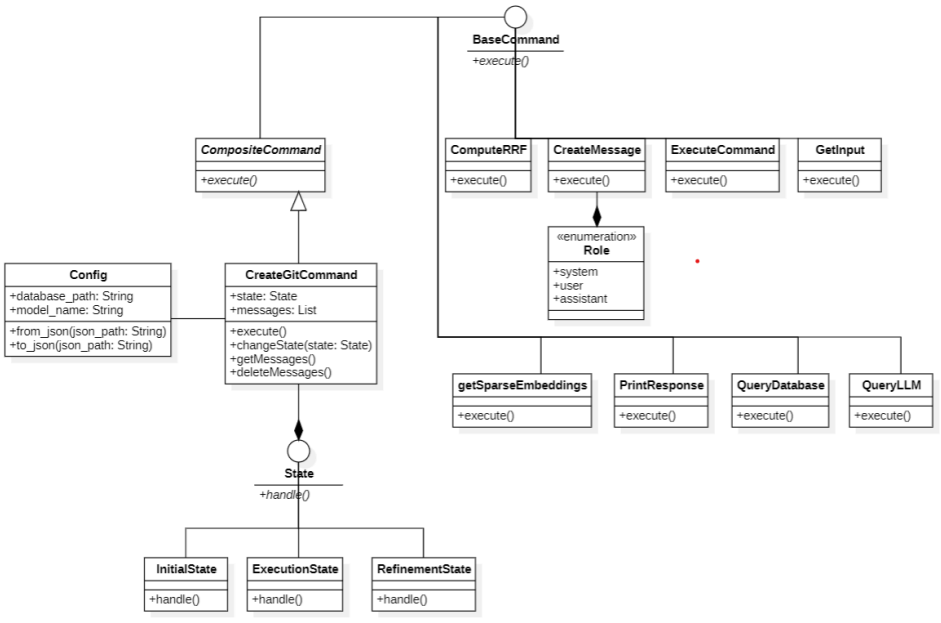

Figure 1: Diagramma delle classi del Sistema di Query

L'integrazione con il modello LLM avviene attraverso un'interfaccia standardizzata che supporta diverse implementazioni, permettendo la facile sostituzione o l'aggiornamento del modello sottostante senza impattare il resto del sistema. Questa flessibilità architettturale consente di adattare il sistema all'evoluzione delle tecnologie LLM mantenendo invariata la logica di business.

3.2.3 Pipeline RAG

La Pipeline RAG rappresenta il componente centrale del sistema di recupero e augmentation delle informazioni, implementata seguendo il pattern Chain of Responsibility per garantire un'elaborazione sequenziale e modulare dei dati. La sua costruzione è gestita attraverso un Builder pattern che assicura una corretta inizializzazione e configurazione di tutti i componenti della catena.

Il flusso di elaborazione inizia con la ricezione di una query o di un documento, che viene processato attraverso una serie di handler specializzati. Ogni handler nella catena può elaborare i dati, arricchirli con informazioni contestuali e passarli al successivo, oppure terminare la catena se necessario. Questa struttura permette di aggiungere o rimuovere fasi di elaborazione senza modificare la logica esistente.

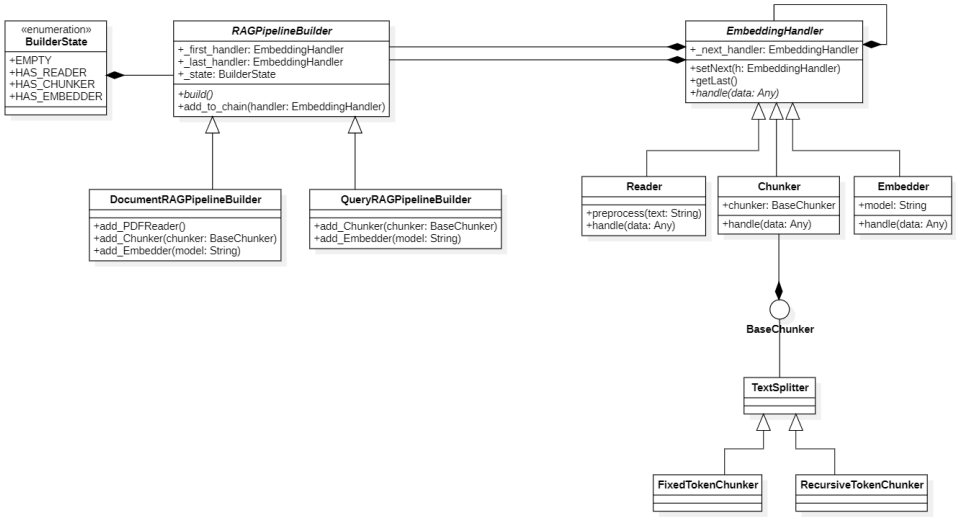

Figure 2: Diagramma delle classi della Pipeline RAG

Il Builder della pipeline offre un'interfaccia fluida per la configurazione, permettendo di specificare:

- la strategia di chunking da utilizzare
- il tipo di embedding da generare
- le soglie di similarità per il retrieval
- i parametri di ranking per la fusione dei risultati

3.2.3.1 Embedding Handler

L'Embedding Handler, integrato nella pipeline come componente specializzato, gestisce la trasformazione dei testi in rappresentazioni vettoriali, interfacciandosi con diversi modelli di embedding attraverso un'astrazione comune che ne semplifica la sostituzione o l'aggiornamento.

3.2.4 Librerie di chunking

Le componenti di chunking utilizzate nel progetto provengono da una libreria scritta da uno sviluppatore di Chroma, e disponibile al link https://github.com/brandonstarxel/chunking_evaluation. Queste

librerie sono una versione modificata di quelle presenti all'interno della popolare libreria LangChain per Python.

4 Architettura di Deployment

4.1 Ambiente di sviluppo

Lo sviluppo del progetto è avvenuto su sistema operativo **Windows 11**, utilizzando l'editor **Visual Studio Code** come ambiente principale. L'editor è stato configurato con estensioni per il supporto a Python, controllo versione Git e formattazione automatica del codice, offrendo un ambiente integrato per la scrittura, l'esecuzione e il debug.

Per la gestione delle dipendenze e dell'ambiente di esecuzione è stato utilizzato **venv**, che ha permesso di isolare le librerie del progetto e mantenere un ambiente di sviluppo riproducibile. Tutte le dipendenze sono state tracciate tramite file `requirements.txt`.

L'interazione con i modelli linguistici (LLM) è stata gestita in locale grazie all'uso di **Ollama**, che consente l'esecuzione di modelli open source su macchina personale, evitando la dipendenza da servizi esterni e garantendo maggiore controllo sul flusso dei dati.

Il codice sorgente è strutturato in moduli Python separati per ciascuna delle funzionalità principali: gestione delle query, pipeline RAG, componenti CLI e test. I test automatici sono stati implementati con `pytest` e la loro esecuzione è stata integrata in una pipeline CI basata su **GitHub Actions**, per garantire la qualità del codice durante il ciclo di vita del progetto.

Questo setup ha permesso un ciclo di sviluppo flessibile, locale e iterativo, mantenendo allo stesso tempo un buon livello di controllo sul codice, sulle dipendenze e sull'infrastruttura.

4.2 Ambiente di produzione

Il sistema è concepito per operare in ambienti locali, sfruttando un'interfaccia CLI e modelli LLM eseguiti tramite Ollama. Tuttavia, al fine di garantire un ciclo di vita del software moderno, robusto e scalabile, il progetto adotta un'infrastruttura di Continuous Integration e versionamento automatizzato, che consente l'esecuzione sistematica dei test e la generazione di release a partire da commit versionati. La produzione è quindi intesa non solo come ambiente di esecuzione finale, ma come risultato di una catena strutturata di validazione e distribuzione del software.

4.2.1 Continuous Integration

4.2.1.1 Descrizione e approccio

Il progetto implementa una pipeline CI completa basata su **GitHub Actions**, pensata per automatizzare l'esecuzione dei test, la verifica della qualità del codice e la gestione del ciclo di rilascio. Ogni modifica apportata al repository viene sottoposta a un processo di verifica continua, assicurando che il codice mantenga elevati standard qualitativi e sia sempre in uno stato rilasciabile.

4.2.1.2 Workflow e automatizzazione

- La pipeline viene attivata automaticamente al verificarsi di eventi Git specifici, come **push** su `main` o l'apertura di una **pull request**.
- Le dipendenze del progetto vengono installate in ambienti puliti a partire dal file `requirements.txt`, garantendo coerenza tra ambiente di sviluppo e ambiente di test.
- I test vengono eseguiti tramite `pytest`, con possibilità di futura generazione automatizzata di report di copertura mediante `pytest-cov`.
- La presenza di un file di configurazione consente di usare il plugin con differenti modelli LLM tramite Ollama, anche se il default raccomandato in CI è `qwen3-4B`.

4.2.1.3 Versionamento e release automatica

- Le **release ufficiali** del plugin vengono generate automaticamente al push di un **tag semantico** (es. `v1.0.0`), seguendo le convenzioni **SemVer**.
- Il contenuto della release include:
  - codice sorgente completo;
  - database vettoriale precompilato;
  - istruzioni per l'installazione manuale.

4.2.1.4 Estensibilità e manutenzione

- Il sistema CI è progettato in maniera modulare: nuovi step, tool o test possono essere integrati facilmente modificando il file `ci.yml`.
- Il supporto a più modelli LLM è garantito da una logica di configurazione centralizzata, che consente di testare nuovi modelli con minimi cambiamenti al codice.
- Il design del sistema prevede l'adozione futura di tecniche di analisi statica (es. `pylint`, `mypy`) o strumenti di code quality, mantenendo la compatibilità con l'infrastruttura esistente.