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 logica e di sistema 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 il software Git, che introduca in quest'ultimo la possibilità di gestire e generare testo in linguaggio naturale, in accompagnamento ad alcune delle principali funzionalità del software. Le principali features implementate dal plugin sono:

- generazione di comandi Git per CLI a partire da query in linguaggio naturale che descrivano l'intenzione dell'utente;
- generazione automatica di commenti di commit a seguito di analisi del codice modificato;
- l'analisi dell'impatto delle modifiche correnti sul repository remoto;
- utilizzo di LLM per la risoluzione automatizzata di 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.

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.

Il software si affida al database vettoriale ChromaDB per la memorizzazione del corpo di intenti utilizzato in fase di Rag per la raccolta di esempi coerenti con la query utente, da fornire come contesto all'LLM.

Oltre a Python come base di sviluppo e ChromaDB come database, il sistema si affida a un software esterno per l'interazione con LLM. Trattasi di Ollama, utilizzato per l'interazione con LLM, e fruibile in python mediante l'apposita libreria Ollama Python.

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**: libreria Python che gestisce l'interazione con l'omonimo 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**: libreria Python utilizzata per interagire con l'omonimo software pensato per l'utilizzo locale di modelli LLM. 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 Logica

L'architettura logica del software nella cartella `src` segue un approccio modulare e a livelli, ispirato ai principi della Clean Architecture e del Domain-Driven Design. I principali moduli individuati sono:

3.1 1. CLI (Interfaccia a Riga di Comando)

La cartella `cli/` contiene i punti di ingresso dell'applicazione (`main.py`), la gestione dell'input/output utente (`user_io.py`) e il parser dei comandi (`command_parser.py`). Questo livello si occupa esclusivamente dell'interazione con l'utente e della raccolta delle richieste, delegando la logica applicativa ai moduli sottostanti.

3.2 2. Application Layer

La cartella `application/` implementa la logica di orchestrazione delle funzionalità principali tramite handler specializzati:

- `cmd_conversation_hanlder.py`, `commit_conversation_handler.py`, `commit_impact_handler.py`, `merge_conflict_handler.py`:

gestiscono i diversi flussi conversazionali e le operazioni core (generazione comandi, commit, analisi impatto, risoluzione conflitti).

- Sottocartelle come `rag/` contengono la logica per la costruzione del contesto tramite RAG (Retrieval-Augmented Generation).

3.3 3. Dominio

La cartella `domain/` definisce le strutture dati centrali e i prompt per l'interazione con il modello LLM:

- `chat.py`, `prompts.py`, `response_structure.py`: rappresentano le entità, i messaggi e i formati di risposta usati in tutto

il sistema.

3.4 4. Infrastructure

La cartella `infrastructure/` fornisce servizi di basso livello e componenti tecnici:

- `embedding/`: pipeline di embedding, chunking e gestione RAG.
- `git_service/`: astrazioni per l'interazione con Git (diff, repository, annotazioni).
- `llm/`: client per la comunicazione con modelli LLM (es. Ollama).
- `database/`: gestione del database vettoriale per il retrieval.
- File contenente gli esempi che costituscono il corpo del sistema di RAG (`examples.json`).

3.5 5. Configurazione

La cartella `config/` contiene la configurazione centralizzata del sistema (`config.py`, `config.json`), permettendo la parametrizzazione dei modelli e delle lingue.

3.6 6. Scripts

La cartella `scripts/` ospita utility e script di supporto che non fanno strettamente parte del software eseguibile, ma sono stati utilizzati nel corso della produzione di tale software.

3.7 Relazioni tra i moduli

- Il livello CLI riceve input dall'utente e invoca gli handler dell'application layer.

- Gli handler orchestrano le operazioni, interagendo con il dominio per la gestione dei dati e con l'infrastructure per

i servizi tecnici (Git, embedding, LLM).
- La configurazione è accessibile trasversalmente da tutti i moduli.
- L'infrastruttura è progettata per essere facilmente sostituibile o estendibile (es. nuovi modelli LLM, database, chunker).

3.8 Diagramma logico

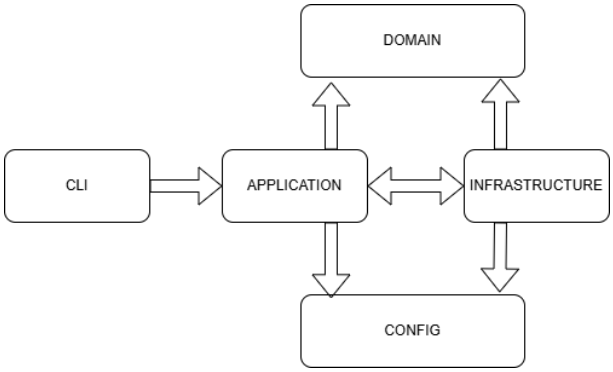

Figure 1: Diagramma logico dell'architettura del software

Questa architettura garantisce separazione delle responsabilità, testabilità e facilità di estensione, rendendo il sistema robusto e manutenibile.

4 Architettura di Sistema

4.1 Panoramica Generale

Il sistema descritto è un plugin progettato per assistere gli utenti nell'utilizzo avanzato di Git tramite un'interfaccia a riga di comando intelligente, potenziata da modelli di linguaggio (LLM) e tecniche di ricerca ibrida realizzata mediante Retrieval-Augmented Generation (RAG) e algoritmo di ranking BM25. L'architettura è pensata per garantire modularità, estendibilità e facilità di manutenzione, ed è stata realizzata applicando best practices come dependency injection, e pattern specifici per la risoluzione di problemi ricorrenti riscontrato durante la realizzazione del software.

Il plugin implementa quattro funzionalità principali, ciascuna orchestrata da moduli dedicati, che coprono la generazione di comandi Git, la creazione di messaggi di commit, l'analisi dell'impatto delle modifiche e la risoluzione automatica dei conflitti.

4.2 Modulo 1 – Generazione ed esecuzione di comandi Git

Il modulo di generazione comandi è progettato secondo un'architettura orientata ai pattern Command e Adapter, con particolare attenzione all'estendibilità e alla separazione delle responsabilità.

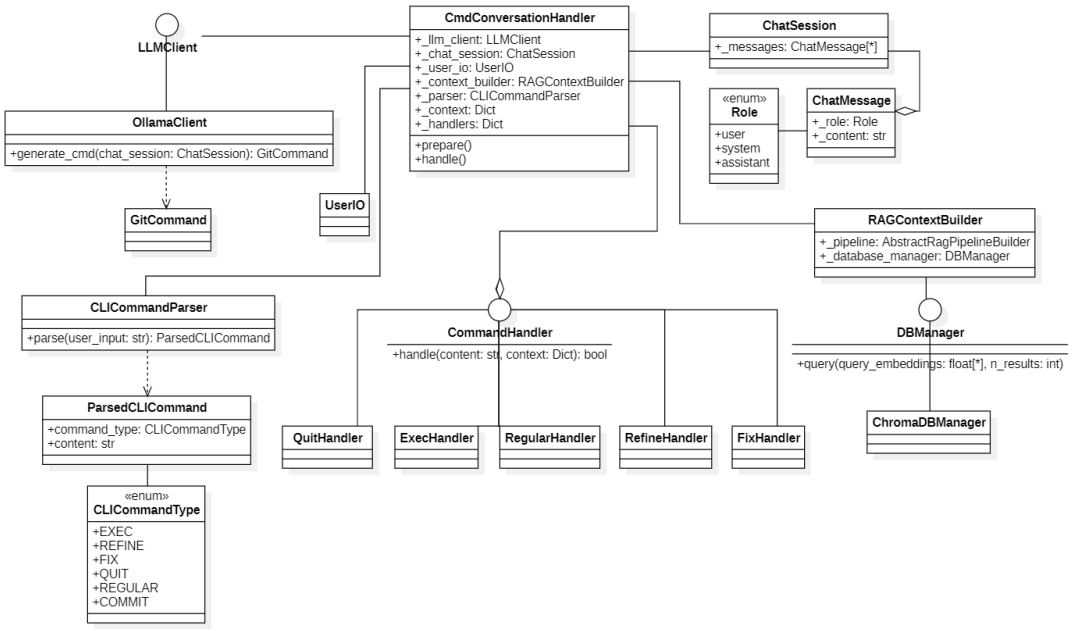

Figure 2: Diagramma logico dell'architettura del software

4.2.1 Struttura e responsabilità

Il cuore del modulo è rappresentato dalla classe `CmdConversationHandler`, che funge da *orchestratore*: riceve l'input dell'utente tramite `UserIO`, costruisce il contesto necessario (con `RAGContextBuilder`), e delega la gestione del comando al parser (`CLICommandParser`). Quest'ultimo trasforma l'input in un oggetto `ParsedCLICommand`, identificando il tipo di comando richiesto.

La gestione effettiva dei comandi è affidata a una gerarchia di handler che implementano il pattern *Command*: la classe astratta `CommandHandler` definisce l'interfaccia comune (`handle(content, context)`), mentre le sottoclassi concrete (`ExecHandler`, `RegularHandler`, `RefineHandler`, `FixHandler`, `QuitHandler`) implementano la logica specifica per ciascun tipo di comando. Questo approccio consente di aggiungere facilmente nuovi tipi di comando semplicemente introducendo nuovi handler, senza modificare la logica esistente.

Il recupero del contesto è invece demandato alla classe `RAGContextBuilder` che attraverso l'attributo `_pipeline: AbstractRagPipelineBuilder`, estrae dal database gli intenti più coerenti con la query dell'utente, andando poi a raffinare la ricerca attraverso l'applicazione, su di essi, dell'algoritmo di ranking BM25. Nell'insieme questa componente realizza una ricerca ibrida.

4.2.1.1 Architettura del modulo RAG

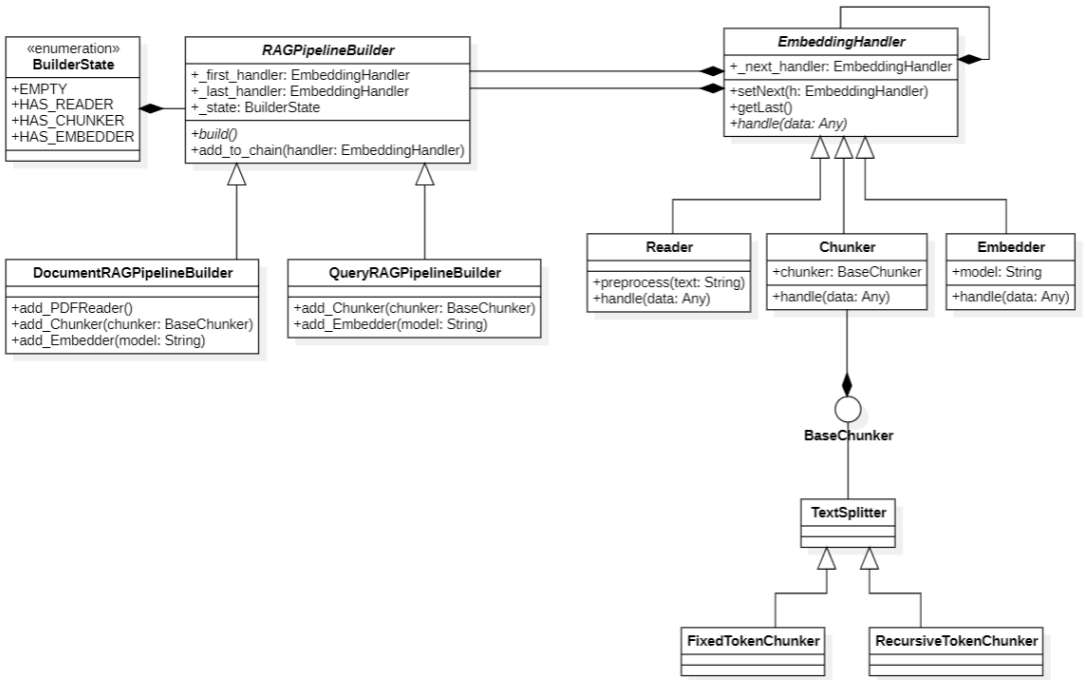

Figure 3: Diagramma logico dell'architettura del software

Il modulo RAG (Retrieval-Augmented Generation) è un componente centrale per la costruzione del contesto semantico, utilizzato dalla classe `CmdConversationHandler` per recuperare esempi coerenti rispetto alla query dell'utente. Il modulo è costruito seguendo un pattern a catena di responsabilità (**Chain of Responsibility**), che consente l'aggiunta dinamica e componibile di handler per la lettura, la segmentazione (**chunking**) e l'embedding del contenuto.

La pipeline è costruita a partire da `RAGPipelineBuilder`, classe astratta che gestisce il concatenamento di `EmbeddingHandler`. Due specializzazioni sono fornite:

- `DocumentRAGPipelineBuilder`: per l'elaborazione di documenti statici (es. PDF);
- `QueryRAGPipelineBuilder`: per l'elaborazione di query in input (modulo 1).

Ogni `EmbeddingHandler` incapsula un'operazione specifica:

- `Reader`: legge e preprocessa il contenuto;
- `Chunker`: segmenta il testo in token, frasi o paragrafi;
- `Embedder`: genera embedding semantici tramite un modello configurabile (es. `"nomic-embed-text"`).

La componente di chunking composta da `BaseChunker` e sottoclassi è una versione modificata rispetto a quella offerta da *LangChain*, e proviene dal repository GitHub associato al report tecnico [Evaluating Chunking Strategies for Retrieval](https://research.trychroma.com/evaluating-chunking) redatto da Chroma.

La classe `RAGContextBuilder`, utilizzata dal modulo principale, istanzia internamente questa pipeline per costruire dinamicamente il contesto semantico sulla base della query utente.

4.2.2 Pattern utilizzati

- **Command Pattern:** Ogni comando dell'utente viene incapsulato in un oggetto handler dedicato, che può essere eseguito in modo indipendente. Questo favorisce l'estendibilità e la separazione delle responsabilità.

- **Adapter Pattern:** L'interazione con componenti esterni o potenzialmente variabili (come il modello LLM) avviene tramite interfacce astratte (`LLMClient`). Ad esempio, `OllamaClient` implementa l'interfaccia `LLMClient`, o `ChromaDBManager` che implementa `DBManager`, permettendo di sostituire facilmente il backend LLM senza impattare il resto del sistema.

4.2.3 Estendibilità

L'architettura è pensata per facilitare l'aggiunta di nuove funzionalità: per supportare un nuovo tipo di comando, è sufficiente implementare una nuova sottoclasse di `CommandHandler` e registrarla nell' orchestratore. Analogamente, l'uso di interfacce astratte per i servizi esterni (LLM, database, ecc.) permette di adattare il sistema a nuove tecnologie o provider con modifiche minime.

4.2.4 Sintesi del flusso

1. L'utente inserisce una query tramite CLI.
2. `CmdConversationHandler` raccoglie l'input, e lo passa al parser.
3. `CLICommandParser` restituisce un oggetto `ParsedCLICommand` che identifica il tipo di comando.
4. L'orchestratore seleziona l'handler appropriato e ne invoca il metodo `handle`.
5. Gli handler possono interagire con il modello LLM (tramite l'interfaccia `LLMClient`), con la pipeline di ricerca ibrida e con altri servizi secondo necessità.
6. Il risultato viene restituito all'utente tramite `UserIO`.

Questa organizzazione garantisce modularità, testabilità e facilità di evoluzione del modulo di generazione

4.3 Modulo 2 – Generazione di messaggi di commit

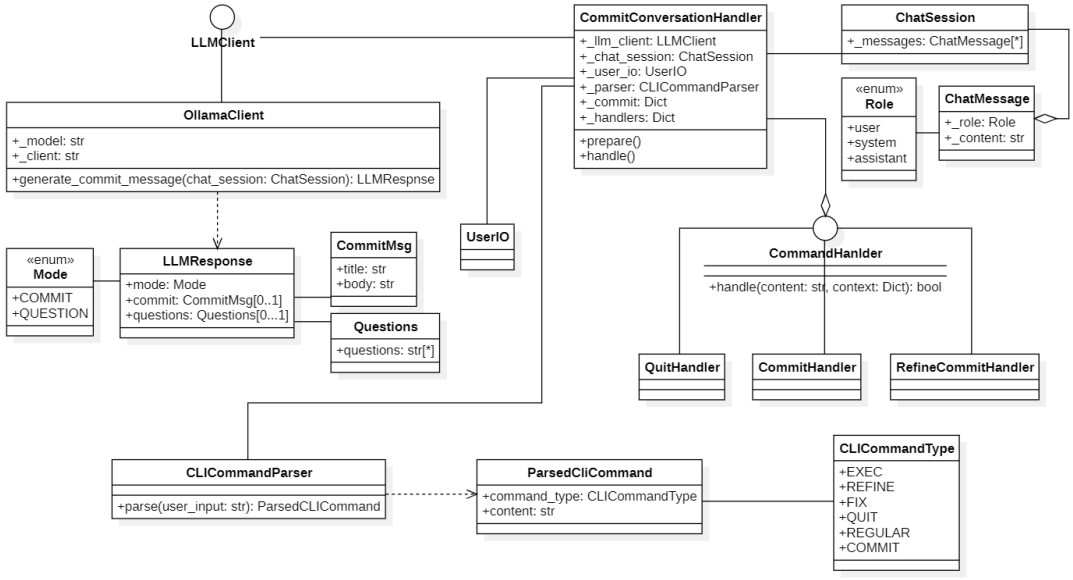

Figure 4: Diagramma logico dell'architettura del software

4.3.1 Struttura e responsabilità

Il modulo per la generazione dei messaggi di commit è centrato sulla classe `CommitConversationHandler`, che svolge il ruolo di orchestratore del flusso conversazionale. Essa

riceve l'input utente tramite `UserIO`, mantiene lo stato della conversazione (`ChatSession`) e utilizza il parser (`CLICommandParser`) per interpretare i comandi. L'interazione con il modello LLM avviene tramite l'interfaccia `LLMClient`, la cui implementazione concreta (`OllamaClient`) si occupa di generare la risposta (`LLMResponse`). La risposta del modello può essere di due tipi, distinti dal campo `mode`:

- **COMMIT:** contiene un oggetto `CommitMsg` con titolo e corpo del messaggio di commit generato.
- **QUESTION:** contiene una lista di domande (`Questions`) che il sistema pone all'utente per chiarire o

arricchire il contesto prima di generare il messaggio di commit definitivo.

La gestione delle azioni conseguenti al comando è affidata a una gerarchia di handler che estendono `CommandHandler`, tra cui `CommitHandler` e `RefineCommitHandler`, responsabili rispettivamente della generazione e della raffinazione dei messaggi di commit.

4.3.2 Pattern utilizzati

- **Command Pattern:** ogni tipo di comando (commit, refine, quit, ecc.) è gestito da un handler dedicato

che implementa una logica specifica, facilitando la separazione delle responsabilità e l'estendibilità del sistema.

4.3.3 Estendibilità

La struttura modulare consente di aggiungere facilmente nuovi tipi di handler per gestire ulteriori modalità di interazione o nuove tipologie di comandi, senza modificare la logica esistente. La separazione tra orchestratore, parser, handler e interfaccia verso il modello LLM garantisce una facile manutenibilità e possibilità di evoluzione del modulo.

4.3.4 Sintesi del flusso

1. L'utente inserisce un comando che avvia la generazione del messaggio di commit tramite CLI.
2. `CommitConversationHandler` raccoglie l'input, aggiorna la sessione e lo passa al parser.
3. Il comando viene interpretato e gestito dall'handler appropriato (`CommitHandler`, `RefineCommitHandler`, ecc.).
4. Il `CommitConversationHandler` interagisce con il modello LLM per generare una risposta (`LLMResponse`).
5. Se la risposta è di tipo *COMMIT*, viene proposto un messaggio di commit all'utente.
6. Se la risposta è di tipo *QUESTION*, vengono poste domande di chiarimento all'utente e il ciclo riprende fino a ottenere tutte le informazioni necessarie.
7. Il risultato finale viene restituito all'utente tramite `UserIO`.

4.4 Modulo 3 - Analisi dell'impatto delle modifiche

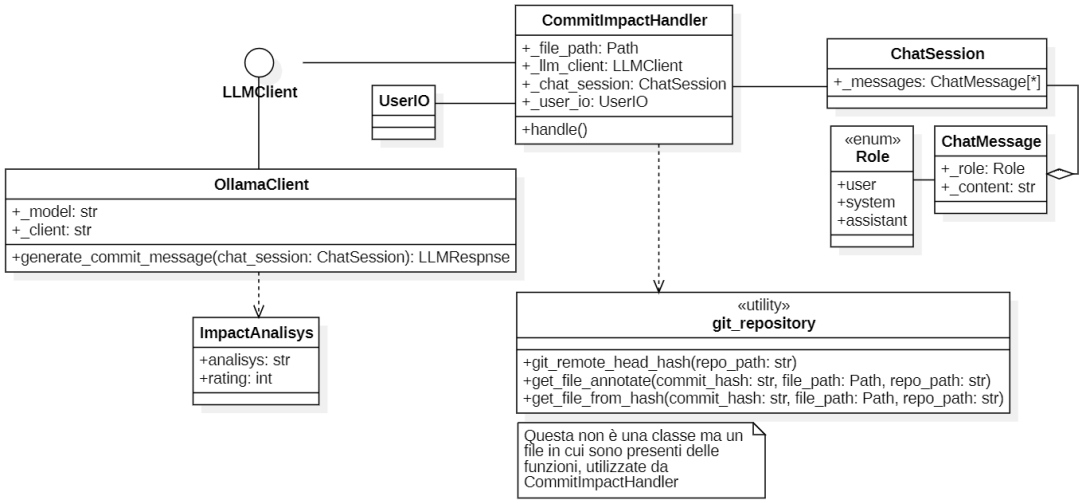

Figure 5: Diagramma logico dell'architettura del software

4.4.1 Struttura e responsabilità

Il modulo per l'analisi dell'impatto delle modifiche è incentrato sulla classe `CommitImpactHandler`, che si occupa di orchestrare l'intero flusso di raccolta dati, interazione con il modello LLM e restituzione del risultato all'utente. La classe mantiene i riferimenti ai componenti principali necessari: il percorso del file da analizzare (`_file_path`), il client LLM (`_llm_client`), la sessione conversazionale (`_chat_session`) e l'interfaccia utente (`_user_io`).

Un aspetto centrale di questo modulo è la raccolta e la preparazione dei dati necessari all'analisi. `CommitImpactHandler` utilizza direttamente alcune funzioni di utilità definite nel modulo `git_repository` (evidenziato come «`utility`» nell'UML), tra cui:

- `get_remote_head_hash(repo_path: str)`: per ottenere l'hash della testa remota del repository.
- `get_file_annotate(commit_hash: str, file_path: Path, repo_path: str)`: per recuperare le annotazioni

(blame) di un file a uno specifico commit.

- `get_file_from_hash(commit_hash: str, file_path: Path, repo_path: str)`: per ottenere il contenuto di un

file a uno specifico commit.

Queste funzioni non appartengono a una classe, ma sono funzioni di modulo importate e utilizzate direttamente da `CommitImpactHandler` per raccogliere tutte le informazioni necessarie a descrivere le modifiche e il loro contesto.

Il risultato dell'analisi è rappresentato dalla classe `ImpactAnalisys`, che funge da semplice contenitore per la risposta generata dal modello LLM (testo dell'analisi e rating numerico).

4.4.2 Pattern utilizzati

In questo modulo non sono stati adottati pattern di progettazione specifici oltre alla separazione delle responsabilità e alla modularizzazione del codice.

4.4.3 Estendibilità

Sebbene il modulo non sia stato progettato esplicitamente per una facile estensione, la struttura attuale permette di aggiungere nuove funzioni di raccolta dati o di modificare la logica di analisi senza impattare significativamente il resto del sistema. L'uso di funzioni di utilità esterne facilita l'integrazione di nuove fonti di dati o metriche di analisi, qualora necessario.

4.4.4 Sintesi del flusso

1. L'utente avvia la richiesta di analisi dell'impatto tramite CLI.
2. `CommitImpactHandler` raccoglie i dati necessari utilizzando le funzioni di utilità di `git_repository` (recupero hash, blame, contenuto file).
3. I dati raccolti vengono formattati e inseriti nella sessione conversazionale (`ChatSession`).
4. Il client LLM elabora la richiesta e restituisce una risposta strutturata (`ImpactAnalisys`).
5. Il risultato viene presentato all'utente.

4.5 Modulo 4 – Risoluzione automatica dei conflitti di merge

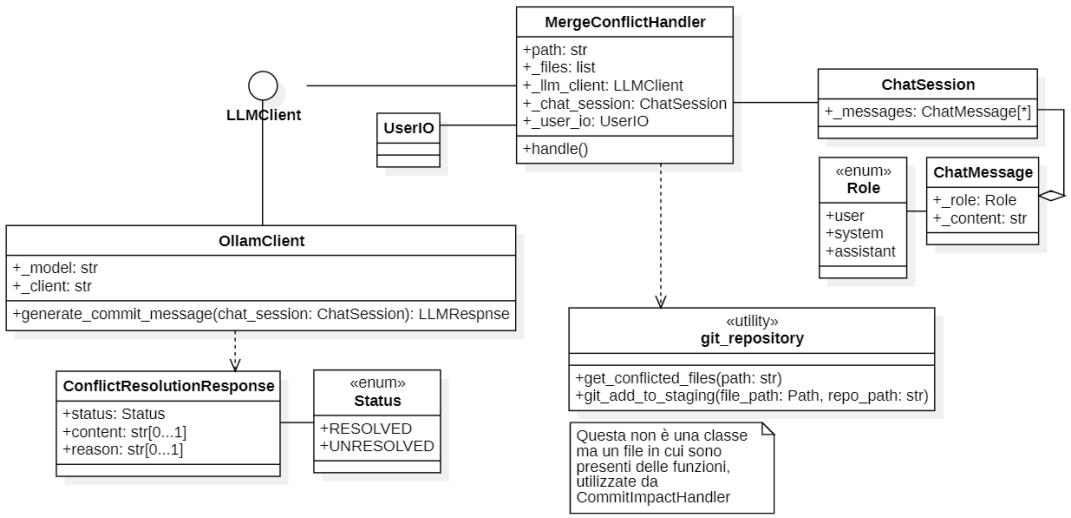

Figure 6: Diagramma logico dell'architettura del software

4.5.1 Struttura e responsabilità

Il modulo per la risoluzione dei conflitti di merge è incentrato sulla classe `MergeConflictHandler`, che si occupa di orchestrare l'intero processo di individuazione e risoluzione dei conflitti. Quando viene invocato, il software verifica se il percorso fornito corrisponde a una directory o a un singolo file:

- Se è una directory, itera su tutti i file presenti per individuare quelli in stato di conflitto.
- Se è un file, si concentra esclusivamente su quello.

L'individuazione dei file in conflitto avviene tramite la funzione di utilità `get_conflicted_files(path: str)` del modulo `git_repository`, che restituisce la lista dei file problematici. Per ciascun file in conflitto, il modulo interagisce con il modello LLM (tramite `LLMClient` e la sua implementazione `OllamaClient`) per tentare una risoluzione automatica. La risposta del modello (`ConflictResolutionResponse`) può essere di due tipi, identificati dal campo `status`:

- **RESOLVED:** il conflitto è stato risolto e viene fornito il contenuto aggiornato.

- **UNRESOLVED:** il conflitto non può essere risolto automaticamente; viene fornita una spiegazione tramite il campo `reason`.

Se la risoluzione ha successo, il file viene aggiunto all'area di staging tramite la funzione utility `git_add_to_staging(file_path: Path, repo_path: str)`.

4.5.2 Pattern utilizzati

In questo modulo non sono stati adottati pattern di progettazione specifici oltre alla separazione delle responsabilità e alla modularizzazione del codice.

4.5.3 Estendibilità

Sebbene il modulo non sia stato progettato esplicitamente per una facile estensione, la struttura attuale consente di aggiungere nuove strategie di risoluzione o di integrare ulteriori controlli senza modificare in modo sostanziale la logica esistente.

4.5.4 Sintesi del flusso

1. L'utente avvia la procedura di risoluzione dei conflitti specificando un file o una directory.
2. `MergeConflictHandler` utilizza la funzione utility `get_conflicted_files` per individuare i file in conflitto.
3. Per ogni file individuato, viene avviata una sessione di risoluzione tramite il modello LLM.
4. Se il conflitto viene risolto (`RESOLVED`), il file aggiornato viene aggiunto all'area di staging tramite `git_add_to_staging`.
5. Se il conflitto non può essere risolto (`UNRESOLVED`), viene fornita all'utente una spiegazione del motivo.
6. Il processo si ripete per tutti i file in conflitto nella directory.