Università di Catania

Ingegneria dei Sistemi Distribuiti– 9 CFU

A.A. 2024/2025

Corso di Laurea Magistrale in Informatica

Dipartimento di Matematica e Informatica

Relazione Progetto

Chatbot per Generazione ed Esecuzione automatica di Unit Test

Mannino Daniele – 1000015345

1. Introduzione all'architettura del Servizio di Generazione Test

Lo scopo di questa relazione è quello di analizzare l'applicazione creata, un Assistente AI per la Generazione e l'esecuzione automatica di Unit Test, focalizzandosi sulle sue caratteristiche architetturali e sull'adozione del design pattern **Remote Facade**.

L'obiettivo principale dell'applicazione è semplificare e automatizzare la creazione ed esecuzione di Unit Test utilizzando un Large Language Model (LLM). L'utente inserisce un blocco di codice sorgente e il relativo linguaggio (es. Python, Java), e l'applicazione si occupa di interrogare un modello AI avanzato per ottenere in risposta il codice di test pertinente e i risultati dei test generati.

L'applicazione è strutturata in un'architettura a due livelli ben definita, tipica dei sistemi distribuiti moderni:

- **Frontend** (Client): Implementato con Gradio (frontend_facade.py), offre l'interfaccia utente (UI) per la raccolta degli input (code e language) e la visualizzazione dell'output. La sua responsabilità è l'interazione utente e la comunicazione con il backend.
- **Backend (Server/Microservizio):** Implementato con FastAPI (backend_facade.py), funge da servizio API REST che gestisce l'integrazione con l'LLM di Groq (utilizzando il modello llama-3.1-8b - instant). La sua responsabilità primaria è l'elaborazione del linguaggio naturale e la logica di business legata all'interrogazione dell'AI.

Questa separazione netta tra i due strati è fondamentale per l'implementazione del pattern Remote Facade, analizzato in dettaglio a seguire.

2. Caratteristiche Funzionali e Tecniche dell'applicazione

L'applicazione espone un'unica, cruciale funzionalità: la generazione e l’esecuzione di Unit Test.

2.1 Flusso Operativo Dettagliato:

- **Input Utente** (Frontend): L'utente inserisce il codice sorgente in un'area di testo e seleziona il linguaggio da un menu a tendina (es. "Python").
- **Invocazione Remota**: Il frontend serializza questi due input in un oggetto JSON strutturato e invia una richiesta HTTP POST aII'endpoint specifico del backend (/generate-tests).
- **Elaborazione del Prompt** (Backend): Il backend riceve il JSON e, utilizzando i dati, costruisce un prompt complesso e altamente specifico. Questo prompt include istruzioni (definizione del ruolo dell'AI come "ingegnere del software esperto") e l’incorporazione del codice utente, definendo chiaramente il compito richiesto (scrivere test, coprire *edge cases,* fornire solo codice).
- **Chiamata LLM**: Il client Groq viene chiamato con il prompt costruito, utilizzando parametri ottimizzati per la generazione di codice (bassa temperatura oer determinismo, limite max di tokens).
- **Post-Elaborazione e Salvataggio**: Il backend esegue il post-processing critico e **salva** il codice sorgente e il codice di test in file temporanei sul disco (generated_files).
- **Esecuzione automatizzata**: La Facade avvia l'esecuzione del codice di test tramite subprocess, gestendo la struttura dei file system (pacchetti Python, struttura Maven Java). Questa

operazione è eseguita in modo **asincrono** (run_in_threadpool) per evitare di bloccare il server API.

- **Raccolta e Pulizia**: Il backend cattura l'output standard e gli errori generati dal processo di esecuzione e, al termine, **pulisce** i file temporanei creati (Python: __init__.py; Java: cartelle Maven e pom.xml).
- **Risposta e Pulizia**: La Facade aggrega il codice di test e il risultato dell'esecuzione in un unico payload JSON pulito da restituire al client.
- **Output Utente** (Frontend): Il frontend riceve il codice dei test e il risultato dell'esecuzione e lo visualizza nel componente Chatbot, mantenendo una cronologia delle richieste e risposte.

3. Il Design Pattern Remote Facade

Il Remote Facade (Facciata Remota) è un design pattern strutturale, estensione del classico pattern Facade, specificamente applicato nel contesto di sistemi distribuiti e architetture a servizi.

Il suo scopo principale è **fornire un'interfaccia** semplice, unificata e **coerente** a un sottosistema complesso di servizi remoti.

In un'architettura a microservizi o a livelli, un client (il nostro frontend Gradio) deve spesso interagire con un backend che, a sua volta, gestisce la complessità dell'interrogazione di servizi esterni o sottosistemi complessi (come un LLM).

Vantaggi Chiave del Remote Facade:

1. **Disaccoppiamento**: Il client non dipende direttamente dai dettagli interni (es. URL dell'LLM, chiave API, formato specifico del prompt, parametri del modello AI).

2. **Riduzione del Traffico di Rete**: Invece di richiedere al client di effettuare chiamate multiple o di inviare dati ridondanti, la Facade aggrega queste operazioni in un'unica richiesta remota.
3. **Sicurezza e Controllo**: La Facade incapsula e nasconde informazioni sensibili (come la chiave API di Groq) e impone regole di business (come la validazione degli input o la logica di retry).
4. **Semplicità del Client**: Il client necessita solo di conoscere l'interfaccia pubblica della Facade, ignorando totalmente la complessità interna.
5. **Gestione del Ciclo di Vita del Test**: La Facade incapsula l'intera complessità dell'esecuzione del codice su server remoto, che include: la gestione del **file system** (salvataggio, creazione di strutture come Maven o pacchetti Python), l'uso di **subprocessi** esterni, la gestione dell'**asincronicità** e la **pulizia** delle risorse, tutti dettagli che il client non deve assolutamente conoscere.

4. Implementazione Remote Facade nel Progetto

L'implementazione del Remote Facade è evidente nell'interazione tra i file backend_facade.py (la Facade) e frontend_facade.py (il Client).

4.1 backend_facade.py

Il file backend_facade.py incarna perfettamente il ruolo della Facade Remota attraverso il suo unico endpoint, /generate- tests.

- **Definizione dell'interfaccia semplificata**:

  Il backend definisce la sua interfaccia pubblica non come una stringa generica, ma come un modello di dati strutturato

```python
# --- Modello di Richiesta Strutturata ---
# Invece di una stringa generica, definiamo un modello di dati
# specifico per la nostra operazione. Questo rende l'API più chiara e robusta.
class TestGenerationRequest(BaseModel):
    code: str
    language: str
```

- **Incapsulamento della logica di Business e Tecnologia**

  Il cuore della Facade è la funzione ***generate_tests***. Qui avviene l'incapsulamento della logica complessa:

  - **Costruzione del Prompt Strategico e Post-Processing**: Il backend non solo ottimizza il prompt per l'LLM, ma gestisce anche la **correzione post-generazione** del codice Python (es. forzando l'importazione da from target_module import ...) per assicurare che il codice sia immediatamente eseguibile.

```
prompt = f"""
Sei un ingegnere del software esperto specializzato in testing. Il tuo compito è scrivere unit test chiari, concisi e completi per il codice fornito,
utilizzando il linguaggio **{request.language}**.

**REGOLE GENERALI:**
1. Fornisci SOLO il codice del test. Non includere spiegazioni, introduzioni, conclusioni, o delimitatori del blocco di codice (come ```java o ```python).
2. Il valore atteso nell'asserzione DEVE essere matematicamente CORRETTO.

**REGOLE SPECIFICHE PER LINGUAGGIO ({request.language}):**

[IF JAVA]:
- **JUnit 5:** Utilizza sempre JUnit 5. Includi l'importazione per l'annotazione @Test. Includi l'import statico completo:
    `import static org.junit.jupiter.api.Assertions.*;`.
- **Getter:** Assumi l'esistenza di un metodo **getter pubblico standard** (es. `getNome()`) per accedere ai campi privati.
- **Output:** Non creare asserzioni su metodi che scrivono solo su console (`System.out.println`).

[IF PYTHON]:
- **Unittest:** Utilizza il modulo standard `unittest`.
- **Importazione:** Devi importare il codice da testare usando l'esatta sintassi: `from target_module import {source_class_name}`.
  **È CRUCIALE usare 'target_module', non il nome della classe.**
- **Mocking:** Usa `unittest.mock.patch` per catturare l'output su console (`sys.stdout`) se necessario per testare metodi che usano `print()`.

Codice da testare:
```{request.language}
{request.code}
```
"""
```

  - **Gestione I/O e Asincronicità:** La Facade gestisce la scrittura dei file sorgente (target_module.py) e test sui percorsi temporanei. Soprattutto, incapsula l'intera funzione execute_tests (che usa subprocess) all'interno di un wrapper **asincrono**. Questo nasconde la necessità di eseguire codice bloccante in un ambiente non bloccante.

  - **Orchestrazione dell'Esecuzione:** La Facade è responsabile dell'intera sequenza di operazioni dopo la generazione AI: **Salvataggio → Esecuzione → Cattura Output → Pulizia**. Questi passaggi sono ridotti a un'unica risposta per il client.

- **Gestione dettagli tecnici**: Il backend gestisce la chiamata all'LLM (passando l'API key, selezionando il modello llama-3.1-8b-instant, e fissando i parametri di generazione come temperature=0.2). Questi dettagli sono vitali per la qualità dell'output, ma sono completamente irrilevanti per l'utente e per il frontend.
- **Filtraggio della Risposta**: L'API Groq restituisce una struttura dati complessa. La Facade estrae solo il valore di interesse e Io restituisce al client in un formato pulito:

In sintesi, l'endpoint /generate-tests è la singola interfaccia che nasconde un complesso sottosistema di *prompt engineering* e interazione con un'API di intelligenza artificiale.

4.2 Frontend_ facade.py

Il frontend non contiene un singolo riferimento al servizio Groq, alla sua chiave API, al nome del modello, né alla logica di costruzione del prompt.

La funzione ***get_ unit_ tests*** si riduce a un'operazione di comunicazione standard:

1. **Preparazione del Payload**: Crea un semplice oggetto JSON con i campi richiesti dalla Facade (code, language).
2. **Invocazione Singola**: Effettua una singola chiamata HTTP requests.post
3. **Gestione del Risultato**: Si aspetta un JSON pulito contenente **due campi cruciali**: il test_code generato e l'**execution_result** grezzo (l'output standard di unittest o Maven). Il frontend estrae entrambi i campi e li presenta all'utente in modo formattato.

Grazie alla Facade, il client è estremamente *indifferente* rispetto alla tecnologia sottostante. Se il backend decidesse di cambiare LLM da Groq a OpenAI, o di implementare una pipeline di moderazione del prompt, il codice del frontend non necessiterebbe di alcuna modifica.

5. Conclusione

L'implementazione del pattern Remote Facade in questa applicazione è un'ottima pratica di design, oltre che una necessità funzionale per gestire l'integrazione di servizi LLM e l'esecuzione remota di codice non attendibile.

Il pattern ha permesso di:

1. **Isolare i dettagli critici**: la chiave API e la complessa logica di prompt engineering e l'intera infrastruttura di esecuzione dei test (I/O, subprocess, Maven/unittest) risiedono esclusivamente nel backend, migliorando sicurezza e manutenibilità.
2. **Semplificare il Client**: Il frontend Gradio è leggero e disaccoppiato, concentrandosi solo sulla UI e sulla gestione della cronologia, rendendolo più facile da sviluppare e modificare.
3. **Unificare un'Operazione Complessa**: L'operazione di "generare ed eseguire test" è stata ridotta a un'unica e robusta chiamata API remota, conforme al principio di singola responsabilità nel contesto di un'architettura distribuita.

Il codice dei file di progetto sopra descritti, il file di requisiti utili da installare per il corretto funzionamento dell'applicazione, sono conservati in un repository GitHub al seguente link: https://github.com/DanyMannino22/Progetto_IDSD. All'interno del repository è inoltre presente il file README.md contenente tutte le istruzioni dettagliate per il corretto funzionamento del ChatBot.