Relazione di progetto di ”Paradigmi di programmazione e sviluppo”

LLM per la generazione di codice di produzione a partire da codice di test Junit

di Torroni Michele n.m.0001050680, anno 2022/2023

0. Indice

1. Introduzione

Il campo dell'informatica è in continua evoluzione, e la necessità di sviluppare software in modo efficiente e rapido è sempre più cruciale. In questo contesto, l'utilizzo dei LLM (Large Language Model) rappresenta una promettente frontiera, consentendo la generazione automatica di codice a partire da un prompt (testuale) fornito dall'utente, sia esso espresso in linguaggio naturale o sotto forma di codice (di test o di produzione). Questa relazione esplora l'applicazione dei LLM nel processo di sviluppo del software a partire da codice di Test.

Nello specifico, a partire da codice di test scritto in linguaggio Java, utilizzando la libreria Junit [JUNIT], si è richiesta, utilizzando ChatGpt3.5-turbo e ChatGpt4-turbo [OAIM], la produzione codice Java e Scala, analizzandone poi i risultati ottenuti.
Sempre nel contesto di questo progetto universitario, è stata sviluppata un'applicazione in linguaggio Scala per rendere più rapido e automatizzato il processo di ricerca sopra esposto.

1.1 I LLM

1.1.2 Definizione e Contestualizzazione

I LLM rappresentano una classe avanzata di modelli di linguaggio, capaci di interpretare e generare testo in modo coerente; più nello *specifico i LLM, Modelli linguistici di grandi dimensioni, sono un tipo di algoritmo di intelligenza artificiale (IA) che utilizza tecniche di apprendimento profondo e set di dati estremamente vasti per comprendere, riassumere, generare e prevedere nuovi contenuti* [DLLM].
Contestualizzati nel campo della programmazione, questi modelli si pongono come un ponte tra il linguaggio naturale e il codice sorgente, promettendo un cambiamento sostanziale nel modo in cui il software viene sviluppato.

1.1.3 Evoluzione e Sviluppi Recenti

Dall'avvento degli algoritmi di apprendimento automatico, i LLM hanno fatto progressi significativi. Modelli come GPT-3 hanno dimostrato la capacità di comprendere il contesto e rispondere in modo contestualmente appropriato, aprendo la strada a nuovi orizzonti nell'automazione della generazione di codice.
Nel contesto di questo progetto sono state misurate le performance dei modelli, forniti da OpenAi [OPAI] ChatGPT-3.5-turbo e ChatGPT-4.

1.1.4 I LLM e la Generazione di Codice

I Large Language Models sono modelli di linguaggio avanzati, capaci di comprendere e generare testo in modo coerente e contestualmente appropriato. L'applicazione di questa tecnologia al contesto della programmazione consente la creazione automatica di codice sorgente a partire da specifiche in linguaggio naturale. Questo approccio rivoluzionario ha il potenziale per accelerare notevolmente il processo di sviluppo del software, riducendo il gap tra la comprensione del dominio e l'implementazione pratica.

1.1.5 Vantaggi dell'utilizzo dei LLM

Riduzione dei tempi di sviluppo: I LLM possono interpretare e tradurre rapidamente le specifiche fornite in linguaggio naturale in codice eseguibile, riducendo significativamente i tempi di sviluppo del software.

Minimizzazione degli errori umani: La generazione automatica del codice riduce la dipendenza dagli sviluppatori umani, riducendo al contempo la probabilità di errori di implementazione.

Aumento della produttività: Gli sviluppatori possono concentrarsi su compiti più creativi e ad alto valore aggiunto (come, ad esempio, il contesto di questo progetto, dove il ruolo umano è prettamente quello di creare al meglio i test e il prompt fornito al modello, per poi analizzare il codice prodotto), mentre i LLM si occupano delle attività più ripetitive e routine.

Facilità nella manutenzione del codice: Il codice generato dai LLM è spesso più consistente e ben strutturato, semplificando le operazioni di manutenzione e aggiornamento.

1.1.6 Sfide e Considerazioni Etiche

Comprensione del contesto: I LLM devono essere in grado di comprendere il contesto delle specifiche fornite per generare un codice accurato. Questo rappresenta una sfida significativa, specialmente quando si affrontano concetti complessi o ambiguità nel linguaggio naturale, che spesso sottintende relazioni implicite tra le entità/operazioni da svolgere

Sicurezza: L'automazione della generazione del codice solleva preoccupazioni sulla sicurezza, poiché gli errori nella comprensione delle specifiche potrebbero portare a vulnerabilità nel software prodotto.

Accettazione della comunità: L'accettazione e l'adozione di questa tecnologia sono questioni cruciali, poiché gli sviluppatori devono essere disposti a fidarsi dei LLM nel processo di sviluppo del software.

Responsabilità: La generazione automatica di codice solleva questioni etiche sulla responsabilità degli sviluppatori. Chi è responsabile in caso di errori o vulnerabilità nel software prodotto?

Controllo e Trasparenza nella Generazione del Codice: Garantire un adeguato controllo e trasparenza nel processo di generazione del codice è essenziale. Gli sviluppatori devono comprendere come i LLM prendono decisioni e come influenzare il risultato.

Impatto Sociale ed Economico: L'adozione su larga scala dei LLM potrebbe influenzare il panorama sociale ed economico, creando nuovi posti di lavoro, ma anche generando interrogativi su come affrontare la potenziale automazione di compiti tradizionalmente umani.

1.2 Analisi Del Problema

Il presente lavoro si concentra sul tema dello sviluppo di codice di produzione a partire da codice di test. L'obiettivo è analizzare le potenzialità di creazione di codice Java e Scala utilizzando i LLM a partire da test JUnit. Questo approccio permette di automatizzare la generazione di codice di produzione a partire dai test, semplificando e accelerando il

processo di sviluppo; Questo approccio permette di assicurare che il codice prodotto soddisfi le specifiche e che sia privo di errori. Inoltre, consente di creare un ambiente di sviluppo più efficiente, in cui i test possono fungere sia da strumento di verifica che da guida per la scrittura del codice di produzione. All'interno di questo contesto, verranno esplorate le potenzialità offerte dai linguaggi di programmazione Java e Scala per la creazione di codice di produzione.
Tutti i test e i risultati prodotti saranno analizzati nei capitoli successivi (3. Test)

1.2.1 Importanza del codice di test

Il codice di test riveste un ruolo critico nello sviluppo software. Esso permette di verificare che il codice di produzione funzioni correttamente e si comporti come previsto nelle diverse condizioni di utilizzo. In tal modo, il codice di test contribuisce a garantire la qualità del software, identificando eventuali bug o malfunzionamenti. Inoltre, il codice di test fornisce una documentazione esauriente sul comportamento del codice di produzione, facilitando la manutenzione e l'aggiornamento del software nel tempo. Pertanto, la creazione di codice di produzione a partire dal codice di test rappresenta una pratica di sviluppo imprescindibile nel realizzare software affidabile e di qualità.

1.2.2 Perché Junit

JUnit è un framework di testing per il linguaggio di programmazione Java. È ampiamente utilizzato per scrivere ed eseguire test unitari, ovvero test mirati a singole unità di codice come metodi o classi. I test JUnit forniscono un modo sistematico e automatizzato per verificare che il codice funzioni come previsto.

Alcuni benefici che Junit può apportare al processo di sviluppo del codice :

- **Automazione dei Test:** I test JUnit possono essere facilmente automatizzati ed eseguiti in modo rapido. Questo permette di eseguire test frequenti e di identificare tempestivamente eventuali problemi nel codice.
- **Riduzione degli Errori:** Eseguire test automatici riduce la probabilità di errori nel codice. Ogni volta che viene apportata una modifica al codice, è possibile eseguire i test per assicurarsi che le modifiche non abbiano introdotto nuovi bug.
- **Miglioramento della Manutenibilità:** I test unitari fungono da documentazione esecutiva del codice. Possono essere utilizzati per comprendere rapidamente come il codice dovrebbe essere utilizzato e per verificare che le modifiche apportate non abbiano effetti indesiderati.
- **Facilità di Refactoring:** Quando si eseguono refactoring o modifiche al codice esistente, i test unitari forniscono una rete di sicurezza che aiuta a garantire che il comportamento atteso rimanga invariato.
- **Supporto per lo Sviluppo Agile:** I test JUnit sono cruciali per le pratiche di sviluppo agile, dove c'è una necessità di sviluppare rapidamente e apportare frequenti modifiche al codice. Garantiscono che le modifiche non abbiano impatti indesiderati.
- **Riduzione del Costo di Debugging:** Identificare e correggere bug durante lo sviluppo è spesso più economico che farlo in fase di produzione. I test unitari consentono di individuare e risolvere problemi prima che raggiungano l'ambiente di produzione.
- **Integrazione Continua:** I test JUnit sono spesso utilizzati all'interno di sistemi di continuous integration, che eseguono automaticamente i test ogni volta che viene apportato un commit al repository. Questo aiuta a garantire che il codice sorgente rimanga sempre funzionante.

In generale, l'implementazione di test JUnit è una pratica consigliata nel processo di sviluppo del software, contribuendo a garantire la qualità, la robustezza e la manutenibilità del codice.

1.2.3 Processo di Generazione di Codice con LLM

Per quanto riguarda l'utilizzo dei LLM per la produzione di codice di produzione

Raccolta delle Specifiche in Linguaggio Naturale: Il processo inizia con la raccolta di specifiche dettagliate in linguaggio naturale, fornite dagli utenti o dagli stakeholder del software. Nello specifico, all'interno del contesto di questo progetto, questa fase si riassume nella stesura del prompt iniziale e la successiva sottomissione dei test all'LLM.

Analisi del Contesto e Interpretazione: I LLM esaminano il contesto delle specifiche, interpretando in modo intelligente i dettagli e le relazioni impliciti per comprendere appieno le richieste.

Generazione del Codice Sorgente: Utilizzando le informazioni raccolte, i LLM generano il codice sorgente in un linguaggio di programmazione specificato, seguendo gli standard e le convenzioni del dominio applicativo.

Ottimizzazione e Validazione: Il codice generato viene successivamente ottimizzato per migliorare la leggibilità e le prestazioni. La validazione automatica e il testing sono parte integrante di questo processo per garantire la correttezza funzionale.
Nello specifico, vengono forniti all'LLM tutti gli errori (che siano essi "generici", come errori nell'interpretazione del problema, di compilazione o a runtime).

1.3 Riferimenti a progetti e prodotti esistenti

Per quanto riguarda i lavori reperibili in rete, gran parte dei relatori esplorano la possibilità di generare codice di test a partire dal codice di produzione (e non viceversa, come accade è accaduto nell'ambito di questo progetto) oppure le capacità dei LLM di scrivere codice a partire da specifiche in linguaggio naturale.

Tutte le fonti/pagine ufficiali sono reperibili al capitolo 5.Riferimenti, all'interno di questa sezione saranno solo esposte (per punti) le caratteristiche salienti:

- **[BCML]** : Si tratta di una classifica dei *"migliori"* (secondo i parametri scelti) modelli open (quindi gratuiti) creati appositamente per generare codice.
  Il post descrive un'approfondita valutazione dei modelli di linguaggio per il codice, focalizzandosi su Code LLMs, attraverso due comuni benchmark: *HumanEval*, una prova di correttezza funzionale su 164 problemi di programmazione Python, e *MultiPL-E*, una traduzione di HumanEval in 18 linguaggi di programmazione. Viene riportato anche l'utilizzo di memoria e i throughput dei modelli. Il post fornisce anche dettagli su come i prompt e le valutazioni sono stati gestiti per ogni modello.
- **[ICLL]** : Blog di Meta, dove viene presentato il modello CodeLLAMA, LLM specializzato per il codice, progettato per generare codice in risposta a prompt di testo. Code Llama, basato su Llama 2, è all'avanguardia tra i LLM pubblicamente disponibili per compiti di codifica, promettendo di rendere i flussi di lavoro dei programmatori più veloci ed efficienti. Con dimensioni di modello di 7B, 13B e 34B, Code Llama supporta varie lingue di programmazione e offre funzionalità di completamento del codice e debug. Il modello è stato valutato positivamente su

benchmark di codifica umana ed è rilasciato con licenza comunitaria per incoraggiare un approccio aperto e responsabile all'IA. Meta sottolinea l'importanza dell'uso responsabile di Code Llama e offre orientamenti su come valutarne le prestazioni, affrontare rischi di input e output, e promuovere la trasparenza nelle interazioni utente. Code Llama è destinato a sostenere ingegneri del software in vari settori, e Meta spera che ispiri la creazione di nuovi strumenti innovativi basati su Llama 2.

- **[ISCO]** : StarCoder e StarCoderBase sono modelli specifici per la generazione di codice (Code LLMs) addestrati su dati concessi in licenza da GitHub, includendo informazioni da oltre 80 linguaggi di programmazione, commit Git, problemi di GitHub e notebook Jupyter. Con una capacità di contesto di oltre 8.000 token, superano altri modelli LLM aperti, permettendo diverse applicazioni interessanti. StarCoderBase supera modelli esistenti su benchmark popolari e si confronta o supera modelli chiusi come code-cushman-001 di OpenAI. Le valutazioni umane mostrano che entrambi i modelli superano quelli più grandi, stabilendo nuovi risultati di riferimento. StarCoder si distingue per essere multilingue e funge anche da assistente tecnico, rispondendo a richieste di programmazione senza ulteriori prompt. Il modello è addestrato su un sottoinsieme di The Stack 1.2, e vengono forniti anche pesi del modello, codice di preelaborazione dati e strumenti di valutazione.
- **[HTUL]** : Il testo esplora l'uso dei LLM nella programmazione, evidenziando benefici e limitazioni. LLM come GPT-3, Codex e Copilot possono generare, completare e correggere il codice, accelerando lo sviluppo e riducendo errori. Possono anche fornire documentazione automatica e rispondere a domande di sviluppo. Tuttavia, presentano limitazioni nella comprensione logica del codice e richiedono revisioni umane. L'autore suggerisce miglioramenti continuativi e un uso responsabile, evidenziando il ruolo umano essenziale nell'integrazione di queste tecnologie nell'ambito dello sviluppo software.
- **[LLTE]** : In questo documento, l'autore esplora l'uso di Large Language Models (LLM), concentrandosi specificamente su GPT, per la generazione automatica di casi di test in un codice Java. L'indagine è divisa in tre scenari di crescente complessità.
  In sintesi, mentre gli LLM possono essere un aiuto prezioso nella generazione di casi di test, specialmente per scenari più semplici, le loro limitazioni emergono in codici più complessi e intricati. Il codice generato funge da utile punto di partenza, ma spesso richiede l'intervento umano per la rifinitura e la correttezza. Inoltre, per un'applicazione efficace, è essenziale garantire che il modello sia stato addestrato o affinato sul codice specifico in questione.
- **[ESUL]** : Questo studio esamina come tre modelli generativi di codice (Codex, GPT-3.5-Turbo e StarCoder) possono essere utilizzati per generare casi di unit test senza la necessità di un raffinamento specifico. Vengono utilizzati due benchmark, *HumanEval* ed *Evosuite SF110*, per esaminare l'effetto della generazione del contesto nel processo di generazione dei test unitari; l'articolo discute anche le implicazioni dell'utilizzo di modelli generativi di codice per la generazione di test unitari in un ambiente di sviluppo guidato dai test (Test Driven Development - TDD). Fornisce anche uno studio sistematico dei tre modelli su un ampio dataset di 47 progetti open-source e 160 classi da HumanEval.
- **[APLC]** : Questo articolo esplora le capacità dei LLM nella generazione automatica di codice, concentrandosi su modelli come Bard, ChatGPT-3.5, ChatGPT-4 e Claude-2.

L'obiettivo è valutare come i livelli di specificità delle richieste influenzino l'accuratezza, l'efficienza temporale e lo “spacing” del codice generato. L'analisi coinvolge un benchmark di 104 problemi di codifica con quattro tipi di richieste e sottolinea variazioni significative nelle prestazioni tra diversi LLM e tipi di richieste. I risultati suggeriscono che l'inclusione di test nelle richieste può migliorare le prestazioni, mentre l'ambiguità nelle richieste rappresenta una sfida. GPT-3.5 e GPT-4 dimostrano una notevole versatilità, mentre Claude ha una buona performance con richieste precise.

2. LLMTestApp

Per automatizzare il processo di testing che sarà illustrato al capitolo 3. Test è stata implementata una semplice applicazione, dotata di interfaccia grafica, scritta in linguaggio Scala 3, sfruttando le API fornite dalla libreria openai-scala-client [OASC], come indicato nella "documentazione ufficiale" di OpenAi [LOAA].

2.1 Requisiti

Si espone di seguito una rapida formulazione dei requisiti individuati durante l'analisi del problema.

2.1.1 Requisiti di business

L'obiettivo è quello di realizzare un'applicativo che possa facilitare l'interazione con i LLM che adottano la struttura delle API dei LLM OpenAi, nel contesto della produzione di codice di multipli linguaggi a partire da un prompt (sia esso formulato sia in linguaggio naturale che sotto forma di codice sorgente);

Nello specifico, l'applicazione dovrà permettere di:

- Interfacciarsi con i LLM OpenAi / OpenAi compatibili;
- Effettuare molteplici iterazioni di request-response all'interno dello stesso contesto/chat;
- Facilitare la definizione del contesto nel caso di molteplici istanze di richiesta con contesto comune.

2.1.2 Requisiti funzionali

2.1.2.1 Requisiti Utente

L'utente deve:

- Interagire con il sistema tramite un'iterfaccia grafica (GUI);
- Interfacciarsi con i LLM di OpenAi (previa indicazione di una APIkey valida, specificabile sia mediante un apposito campo della GUI che all'interno di un file di configurazione);
- Interfacciarsi con i LLM OpenAi compatibili, specificando l'indirizzo sul quale è in ascolto il LLM (specificabile sia mediante un apposito campo della GUI che all'interno di un file di configurazione);
- Poter definire una lista di LLM (mediante un file di configurazione) e scegliere per ogni contesto quale modello utilizzare;
- Poter definire una lista di linguaggi di programmazione (mediante un file di configurazione) e scegliere per ogni chiamata quale linguaggio utilizzare;
- Poter definire, all'interno di un file di configurazione, un prompt iniziale da sottoporre all'LLM;
- Poter effettuare chiamate successive mantenendo il contesto/chat;
- Poter utilizzare un pulsante della GUI per resetare la chat (che, dopo l'inizializzazione, consterà solo del prompt iniziale inserito all'interno del file di configurazione);
- Poter visionare lo storico della conversazione mediante un pulsante della GUI (che verrà mostrato all'interno del pannello di inserimento testuale della GUI);
- Poter eseguire un comando su CMD, specificabile sia mediante un apposito campo della GUI che all'interno di un file di configurazione;

- Disporre di un campo testuale dove inserire la richiesta da sottoporre all'LLM.
- Poter visionare la risposta prodotta dall'LLM all'interno di un campo della GUI.

2.1.2.2 Requisiti di Sistema

Il sistema deve:

- Leggere, all'apertura dell'applicativo, un file di configurazione avente estensione ".conf" e valorizzare i campi della GUI relativi alle configurazioni recuperate; nello specifico dovrà permettere di configurare:
  - Una Apikey di OpenAi;
  - Un indirizzo (su cui deve risiedere un LLM OpenAi compatibile);
  - Un comando CMD;
  - Un prompt iniziale;
  - Una lista di linguaggi di programmazione;
  - Una lista di LLM;
- Interagire con il LLM attraverso le API di tipo "ChatCompletion" (che permette di mantere il contesto/lo storico della conversazione);
- Mostrare all'utente, all'interno di un campo testuale della GUI, il contenuto dell'ultima risposta fornita dall'LLM.

2.1.3 Requisiti non funzionali

- Il sistema dovrà essere semplice ed intuivo; gli utenti dovranno poter interagire con esso senza una formazione specifica;
- L'utente dovrà essere in grado di interrogare i LLM effettuando il minor numero di interazioni con la GUI possibile;
- Il sistema dovrà essere stabile e reattivo.

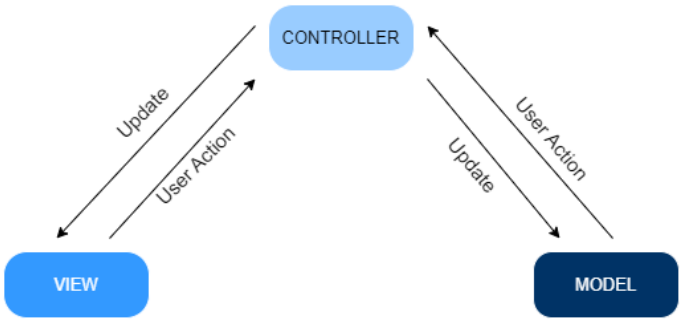
2.1.4 Requisiti d'implementazione

Il sistema dovrà essere implementato utilizzando:

- Scala 3.x
- JDK 17+
- Sbt
- Git

2.2 Design Architetturale

Il design dell'applicazione si basa sull'architettura MVC (Model-View-Controller), in modo da garantire una suddivisione chiara delle responsabilità delle sue componenti.



L'architettura dell'applicazione prevede la comunicazione tra i componenti seguendo il flusso tipico dell'MVC:

- L'utente interagisce con la vista (GUI), generando eventi, nello specifico: configurare ed effettuare le richieste da sottoporre all'LLM;
- Gli eventi sono catturati dalla vista, che notifica il controller.
- Il controller elabora gli eventi, aggiorna lo stato del model (ad esempio a seguito della modifica delle configurazioni della chiamata) e comunica con la vista per riflettere eventuali cambiamenti.
- Il model può generare eventi a seguito di cambiamenti di stato asincroni (ad esempio l'arrivo della risposta generata dall'LLM), notificando gli observer (in questo caso, il controller).
- La vista può quindi essere aggiornata in risposta ai cambiamenti nel modello.

2.3 Design di dettaglio

L'applicazione è organizzata all'interno di un package denominato "app" (percorso completo: *LLMTestApp**src**main**scala**app*\) , con un punto d'ingresso rappresentato dall'oggetto "Launch" (contenuto all'interno dell'omonimo file "Launch.scala").
Allo stesso livello troviamo tutti gli altri file, nello specifico:

- LlmTestController.scala
- LlmTestModel.scala
- LlmTestModelObserver.scala
- LlmTestView.scala
- LlmTestViewObserver.scala

Si evidenzia che, data la natura sperimentale del progetto, all'interno del package è presente anche una sottodirectory "PlayGround", contenente semplici file "temporanei", utili per testare rapidamente il funzionamento delle API/Libreria in caso di sviluppi futuri.

Come affermato nel paragrafo precedente (2.2 Design di dettaglio) il sistema consta principalmente di tre componenti (Model, View e Controller), di seguito descritti:

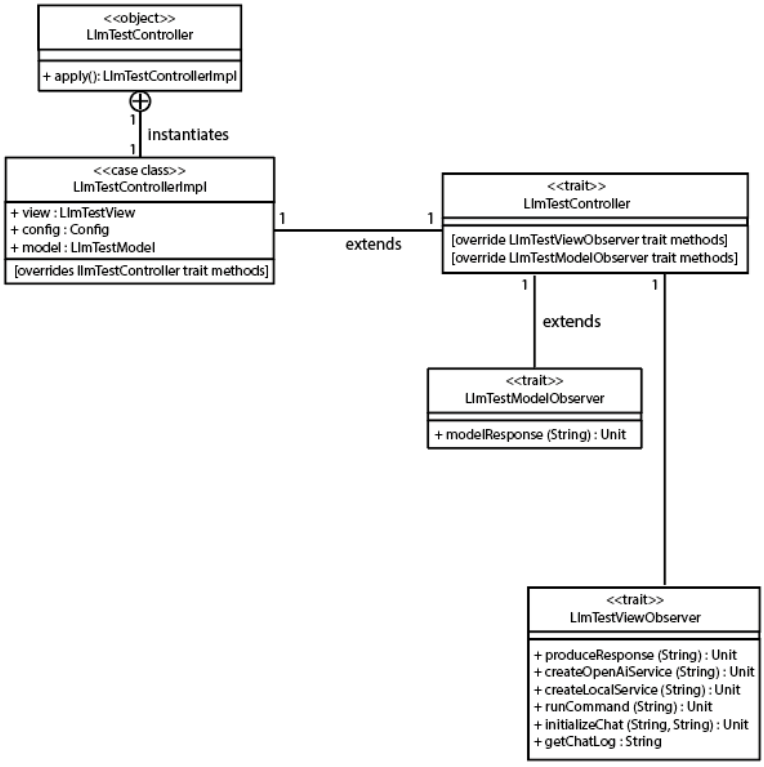
2.3.1 Controller

Il controller coordina le interazioni tra il modello e la vista.

Nel codice fornito, il componente è implementato come un trait *"LlmTesController"* (e relativo companion object) che include le funzionalità necessarie per gestire le azioni dell'utente e coordinare la comunicazione tra modello e vista, attraverso i metodi specificati all'interno dei trait '*LlmTestModelObserver'* e '*LlmTestViewObserver'*.
L'object è dotato di un solo metodo "apply()" che crea un'istanza della case class *"LlmTestControllerImpl"*, che a sua volta implementa i metodi del trait e al cui interno risiede la logica vera e propria del componente.

All'istanziamento della classe vengono create un'istanza del Model e una della View, e vengono passate ad esse le configurazioni necessarie (recuperate dal file 'config.conf').

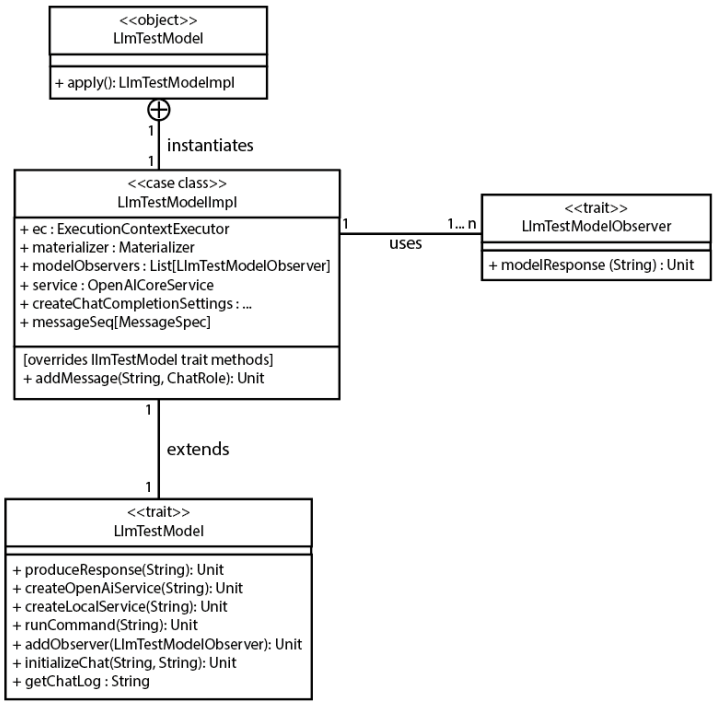
Si lascia di seguito il class diagram del componente Controller:



2.3.2 Model

Il modello, rappresentato dal trait "LlmTestModel" (e relativo companion object), incarna la logica di business e i dati dell'applicazione. Nel codice fornito, questo trait definisce le funzionalità principali del Model: il modello mantiene uno stato interno, gestisce la comunicazione con servizi esterni (come il servizio OpenAI) e fornisce metodi per rispondere agli eventi dell'utente.
Il model possiede una struttura simile al Controller, quindi la logica vera e propria e le implementazioni del trait sono contenute all'interno della case class "LlmTestModelImpl". Fulcro del modello è l'interfacciamento alle OpenAiAPI e permette quindi di interrogare i LLM (fornendo un prompt) e ricevere da essi una risposta (che sarà poi, per mezzo del trait 'LlmTestModelObserver', recapitata alla View dal Controller).

Si lascia di seguito il class diagram del componente Model:



2.3.3 View

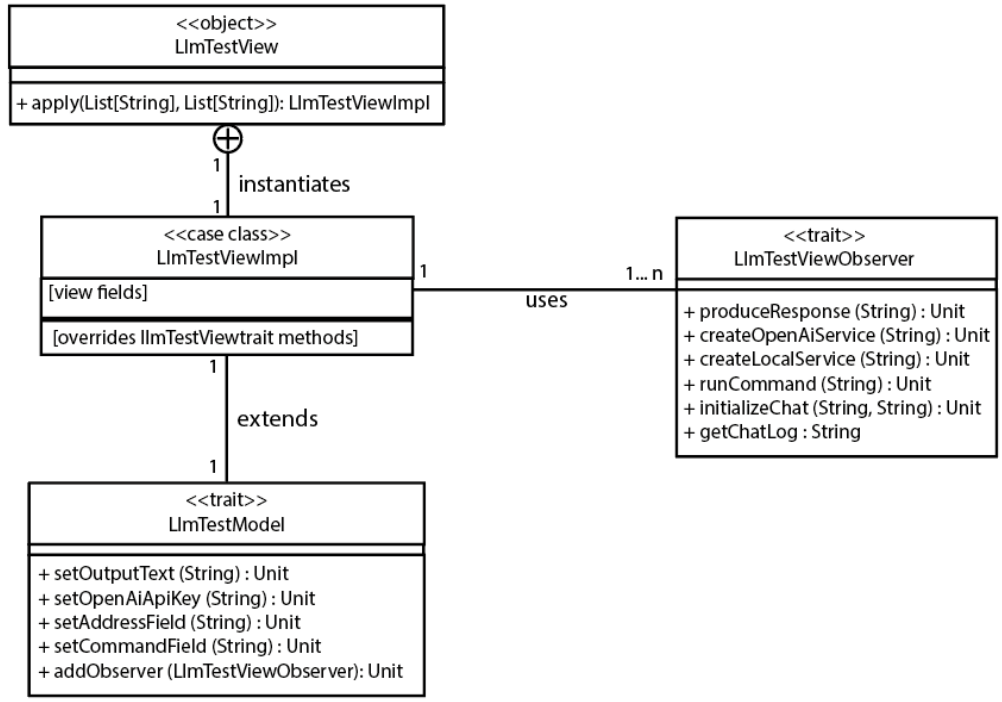
La vista, implementata all'interno del file "*LlmTestView.scala*", rappresenta l'interfaccia utente (UI) e mostra i dati provenienti dal modello e dal file di configurazione. Nel codice fornito, la case class "LlmTestViewImpl" definisce la struttura della GUI utilizzando la libreria Swing di Scala [DLSS]. La vista reagisce agli eventi dell'utente (innescati mediante la pressione dei pulsanti) e notifica il Controller corrispondente attraverso il trait "LlmTestViewObserver".

La vista consta di:

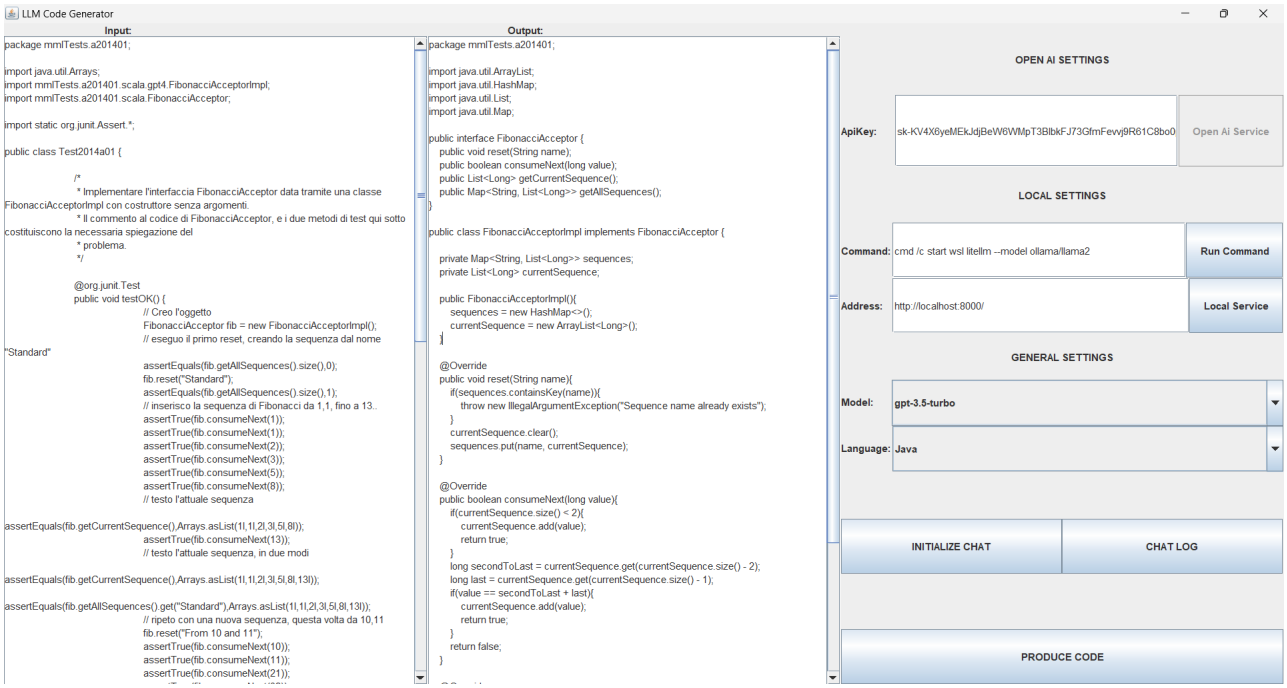
- Un campo 'Input', che permette all'utente di inserire il corpo della richiesta e all'interno del quale puó essere mostrato lo storico della conversazione;
- Un campo 'Output', utilizzato per mostrare all'utente il contenuto della risposta prodotta dall'LLM;
- Un campo 'Apikey', utilizzata da OpenAi per decurtare i fondi necessari dall'account dell'utente;
- Un pulsante 'OpenAiService', alla cui pressione viene istanziata una sessione con OpenAi;
- Un campo 'Command', dove è possibile specificare un comando CMD, eseguibile attraverso un pulsante 'Run Command';
- Un campo 'Address', che permette di specificare l'indirizzo dell'LLM (OpenAiCompatibile);
- Un campo 'Local Service', alla cui pressione viene istanziata una sessione con il LLM all'indirizzo specificato all'interno del campo 'Address';
- Un pulsante 'Initialize Chat', che permette di inizializzare la chat/contesto (che consterà solo del prompt iniziale inserito dall'utente all'interno del file di configurazione);

- Un pulsante ‘Chat Log’, alla cui pressione verrà mostrato, all’interno del campo ‘Input: ’, lo storico della conversazione
- Un pulsante ‘Produce Code’, che permette di inviare la richiesta di produzione di codice all’LLM selezionato.

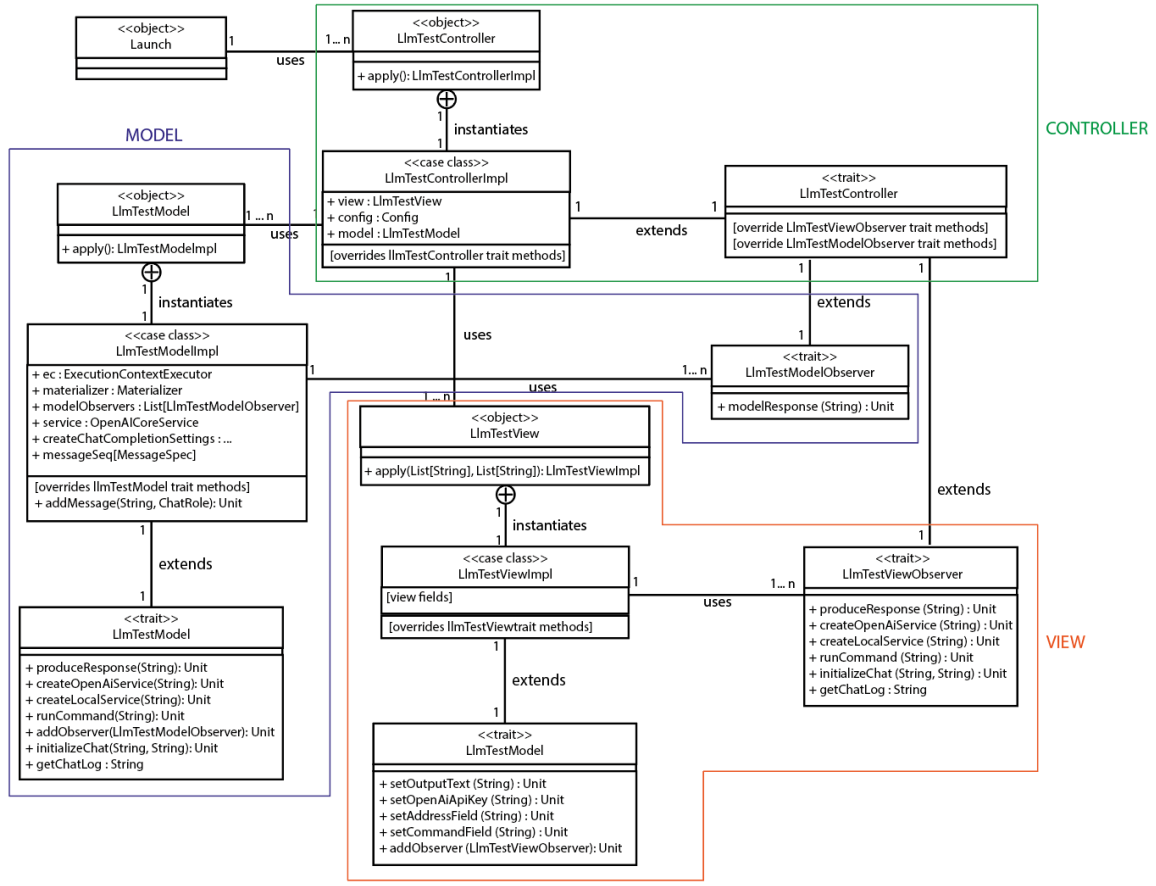
Si presenta il class diagram del componente View:



L’interfaccia grafica istanziabile attraverso questo componente è mostrata nella seguente figura:



Si riporta infine il diagramma delle classi dell'intero sistema, utile per schematizzare l'interazione dei vari elementi che lo compongono:



2.4 Implementazioni

2.4.1 Controller

Per permettere una configurazione più agevole dei parametri, si è fatto uso dell'interfaccia Config contenuta all'interno della libreria typesafe [DLCO]; si riporta un esempio di utilizzo della libreria nel contesto dell'istanziamento della View:

```
val config: Config = ConfigFactory.load("config")
view.setOpenAiApiKey(config.getString("openAiApiKey"))
view.setAddressField(config.getString("address"))
view.setCommandField(config.getString("cmdCommand"))
```

E le corrispetive risorse del file 'config.conf':

```
openAiApiKey = "sk-gkXZ9zFbDPUbfEwdtnq9T3BlbkFJ17dK5qCBe-
CBDd7Bza12I"
address = "http://localhost:8000/"
cmdCommand = "cmd /c start wsl litellm --model ollama/llama2"
```

La libreria permette quindi di istanziare un oggetto di tipo 'Config' che espone i metodi 'get…', richiedenti in ingresso il nome della risorsa desiderata.

2.4.2 Model

Si presentano di seguito l'implementazione dei due metodi utilizzati per istanziare l'oggetto 'service', di tipo 'OpenAICoreService' (trait fornito dalla libreria open-ai-scala-client [OASC]).

Nello specifico, il metodo 'createOpenAiService(…)' restituisce un oggetto in grado di interrogare le API di OpenAi (es: Gpt-3.5-tubro), mentre l'oggetto restituito da 'createLocalService(…)' permette di interagire con il LLM (avente API OpenAi-compatibile) in ascolto all'indirizzo indicato in input (all'interno del campo 'Address' della GUI).

```
override def createOpenAiService(myApiKey: String): Unit = ser-
vice = OpenAIServiceFactory(apiKey = myApiKey)
override def createLocalService(address: String): Unit = service
= OpenAICoreServiceFactory(address)
```

Si propone di seguito l'implementazione del metodo 'produceResponse' utilizzato per effettuare la richiesta vera e propria all'LLM, che fa uso del metodo 'createChatCompletion()'; si tratta di un metodo asincrono, quindi la risposta è contenuta (quando disponibile) all'interno di una *future* istanziata dal metodo 'map'.

```
override def produceResponse(inputText: String): Unit =
  addMessage(inputText, ChatRole.User)
  service.createChatCompletion(messages = message, settings =
createChatCompletionSettings
  ).map { chatCompletion => {
    val response = chatCompletion.choices.head.message.content
    addMessage(response, ChatRole.Assistant)
    modelObservers.foreach(obs => obs.modelResponse(re-
sponse))}}
```

Lo storico dei messaggi è contenuto all'interno della variabile "*message*" (di tipo Seq[MessageSpec]), che può essere consultata in ogni momento dall'utente dopo un'opportuna conversione in stringa, :

```
override def getChatLog: String =
  var chatLog : String = ""
  message.foreach(msg =>  msg.role match
    case ChatRole.System => chatLog = chatLog +
"_________BASEPROMPT_________" + "\n" + msg.content
    case ChatRole.User => chatLog = chatLog  + "\n\n" +
"_________USER_________" + "\n" +msg.content
    case ChatRole.Assistant => chatLog = chatLog + "\n\n" +
"_________ASSISTANT_________" + "\n" +msg.content
    case _ => chatLog = "an error occurred, tip : reset conversa-
tion")
  chatLog
```

Si noti come ogni MessageSpec possiede all'interno una variabile 'role', che puó assumere i valori:

- ChatRole.System - è il caso del prompt iniziale specificato all'interno del file 'config.conf';

- ChatRole.User – che rappresenta un prompt inserito dall'utente all'interno del campo 'Input:' della GUI;
- ChatRole.Assistant – che rappresenta il contenuto di una risposta fornita dall'LLM.

2.5 Casi d'uso

Esistono principalmente due casi d'uso dell'applicativo:

1. È possibile interfacciarsi ai i modelli forniti da OpenAi [OAIM] (previa impostazione dell'ApiKey reperibile sempre al link fornito) attraverso il pulsante "**Open Ai Service**";
2. È anche possibile effettuare richieste a tutti i modelli che presentino un'API OpenAi compatibile (es: Llama) attraverso la sezione "LOCAL SETTINGS", previa specifica dell'indirizzo su cui risiede il LLM.

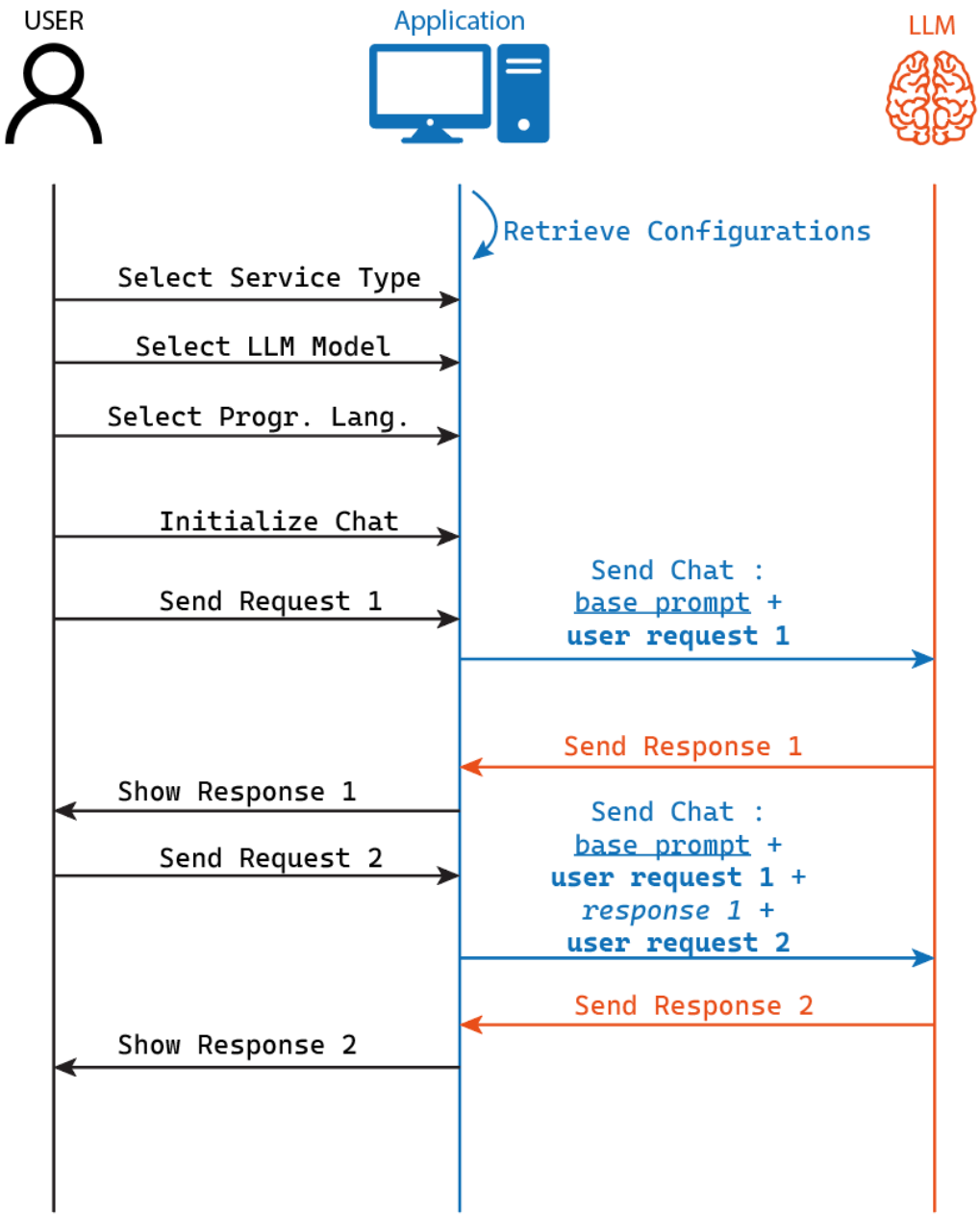
È necessario poi selezionare il LLM e il linguaggio su cui si vuole eseguire i test e premere il pulsante "**INITIALIZE CHAT**" (che crea un'istanza di chat "vuota", utilizzabile quindi per effettuare anche test successivi con input diversi, poiché esegue un "reset" della conversazione, inizializzandola con il prompt presente all'interno del file di configurazione "resources/config.conf").

Per effettuare la richiesta vera e propria è sufficiente inserire un prompt testuale all'interno del pannello "*Input:*", utilizzare il pulsante "**PRODUCE CODE**" ed attendere la risposta, che sarà mostrata all'interno del pannello "*Output: "*. A seguito della ricezione di una risposta, l'utente potrà specificare un nuovo prompt, sempre all'interno del campo "*Input:*" da sottoporre all'LLM.

È importante specificare la possibilità di eseguire un comando sul processo in esecuzione, ad esempio il comando "cmd /c start wsl litellm --model ollama/llama2" istanzierà su WSL (Windows Subsystem Linux) il modello llama2 (previa installazione sulla macchina in uso) attraverso il pulsante "Run Command", posto nella sezione LOCAL SETTINGS; si lascia un esempio dei comandi possibili per effettuare la sopracitata installazione:

- curl https://ollama.ai/install.sh | sh
- ollama pull llama2
- sudo apt install python3-pip
- pip install litellm
- litellm --model ollama/llama2 (optional: --debug)

Si lascia di seguito uno schema riassuntivo delle possibili interazioni dell'utente con il sistema, nel quale si mostra un'istanza composta da due iterazioni; si ricorda che possono essere effettuate molteplici interazioni all'interno dello stesso contesto e vi è la possibilità di resettare il contesto (chat) attraverso il comando 'Initialize Chat':


USER
Application
LLM
Retrieve Configurations
Select Service Type
Select LLM Model
Select Progr. Lang.
Initialize Chat
Send Request 1
Send Chat :
base prompt +
user request 1
Send Response 1
Show Response 1
Send Request 2
Send Chat :
base prompt +
user request 1 +
response 1 +
user request 2
Send Response 2
Show Response 2

3. Test

Come introdotto al capitolo 1. Introduzione, nel contesto di questo progetto sono stati eseguiti alcuni test per misurare le performance dei modelli ChatGPT-3.5-Turbo e ChatGPT-4 per quanto riguarda la capacità di produrre codice di produzione a partire da test Junit.

All'interno di questo capitolo si presenta quindi la metodologia utilizzata e i risultati ottenuti.

Il foglio di calcolo contenente la descrizione estesa degli errori e dei risultati ottenuti è reperibile all'indirizzo [TEST].

3.1 Metodologia

Sono stati sottomessi 13 test differenti, provenienti dagli esami passati del corso di Programmazione Ad Oggetti [POOP] tenuto dal professore Mirko Viroli [PIMV]; per ognuno di essi si è richiesta la produzione di codice in linguaggio Scala e Java utilizzando sia ChatGPT-3.5-Turbo che ChatGPT-4 (per un totale di 4 istanze per ogni test Junit).

Per ogni istanza di test, dopo la prima richiesta, contenente il codice sorgente, sono state effettuate molteplici iterazioni (contenenti gli errori rinvenuti nel codice prodotto), il cui numero è variato dipendente da una di queste condizioni di terminazione:

- 5 richieste di correzione (per un totale di 6 iterazioni);
- response error (ad esempio, nessuna risposta dopo 10 minuti o risposta incompleta);
- il codice fornito non è variato dall'iterazione precedente;
- il codice soddisfa i requisiti (tutti i Test Junit passati con successo).

3.1.1 Prompt

Per quanto riguarda il prompt "iniziale" sottomesso ai LLM, esso è reperibile all'interno del file di configurazione LLMTestApp\src\main\resources\config.conf ed è strutturato come segue:

"Can you provide the code to pass theese tests succesfully?

Provide only the code, no more information (like descriptions, apologies, etc.).

The code must pass all the tests.

All the code provided by you must be contained in a single file.

If some tests fail, I will send you the reported errors in subsequent requests;

you will correct the code sent in your last response.

Don't change or repeat the code I provide you.

The programming language is "

Il testo fornito è stato raffinato attraverso la somministrazione di 15 test ai LLM sopracitati e contiene tutte le istruzioni utili per ottenere una risposta quanto più possibile coerente in relazione al compito in oggetto, nello specifico:

- viene richiesta la produzione di codice che possa eseguire tutti i test forniti con successo;

- si richiede di omettere tutte le informazioni accessorie (come spiegazioni, richieste di scuse, ecc…);
- si richiede di limitare (per quanto possibile) i risultati ad un singolo file omnicomprensivo;
- viene specificato che possono essere effettuate richieste successive, contenenti gli errori eventuali errori rilevati nel codice prodotto;
- L'ultima riga specifica il linguaggio desiderato, il quale è stato omesso per permettere all'utente di selezionarlo all'interno della GUI dell'applicativo, senza la necessità di modificare il file di configurazione in caso di test differenti.

Per quanto riguarda invece il prompt inserito dall'utente all'interno del campo di pannello "Input: " (vedi 2.4.3 View), esso si suddivide principalmente in 2 tipologie:

La prima richiesta di produzione di codice è composta da 3 elementi:

1. Il codice del test vero e proprio;
2. (opzionalmente) la frase "i also provide theese x y";
    a. "x" -> numero di sorgenti aggiuntivi;
    b. "y" -> tipologia dei sorgenti (es: "interfaces", "traits", "classes", ecc…).
3. (opzionalmente) il codice dei sorgenti aggiuntivi.

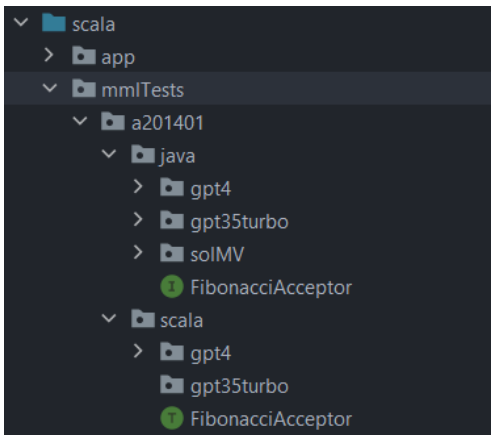
Le successive richieste contengono gli errori rinvenuti in sede di testing del codice prodotto, che siano essi di compilazione, runtime o errori nelle "assertion" contenute all'interno dei test Junit (all'interno del file).

3.1.2 Packages

Il repository del progetto [REPO] contiene sia i sorgenti dell'applicativo LLMTestApp che i test e il codice prodotto dai LLM, nello specifico:

- I test sono reperibili al path LLMTestApp\src\test\scala\mmlTests\
- Specularmente, il codice prodotto è situato all'interno della directory *LLMTestApp**src**main**scala**mmlTests*\

Per quanto riguarda la struttura del codice di produzione, esso è strutturato, per ogni test effettuato, come illustrato nell'immagine che segue:



Schematizzabile in :

- nomeTest

- java
  - gpt4
  - gpt35turbo
  - solMV - soluzione prof Mirko Viroli [PIMV]
  - (eventuali) sorgenti aggiuntivi (es: interfacce/classi)
- Scala
  - Gpt4
  - Gpt35turbo
  - SolMV – soluzione prof Mirko Viroli [PIMV]
  - (eventuali) sorgenti aggiuntivi (es: traits/classi)

3.2 Risultati

I risultati ottenuti a seguito dei sopracitati test sono consultabili all'interno del file "LLMTestApp\doc\tests.xlsx" del repo di progetto [TEST].

La **colonna D** "Attempt N." rappresenta il numero dell'iterazione (successiva alla prima richiesta, contenete il codice sorgente).

Per calcolare lo "Score" (**colonna E**) è stato attributo il valore

- 1 - per i test eseguiti con successo.
- 0.5 - per i test risultanti in "AssertionError".
- 0 - per i test che hanno presentato errori (colonna H) di compilazione, a runtime o di tipo response.

Il campo "n. Junit Tests" (**colonna F**) rappresenta il numero di test Junit contenuti all'interno dei sorgenti e la colonna successiva "Score %" è calcolata come *(Score/n.JunitTests)*100*; è possibile quindi dedurre che se la **colonna G** contiene il valore "100", il codice prodotto dall'LLM passa tutti i test con successo.

La **colonna H** "Error Type" descrive, se presente, la tipologia di errore del codice prodotto dall'LLM, e puó assumere i valori:

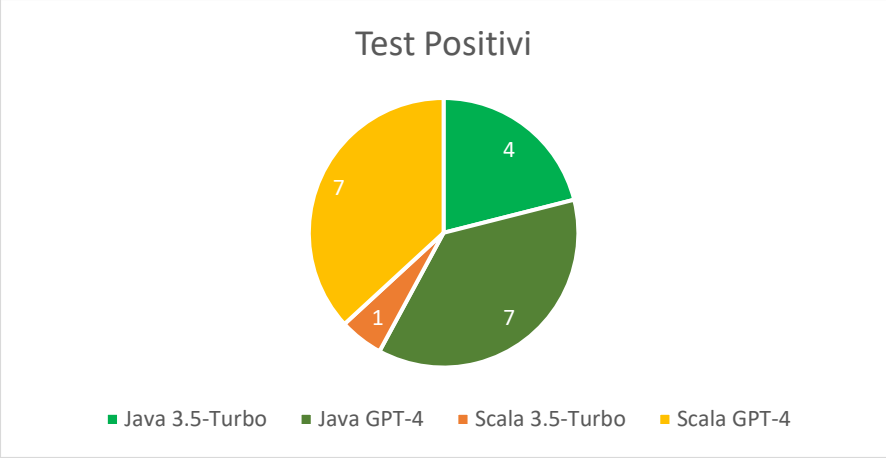
- Assertion – il codice prodotto genera errori di tipo "AssertionError" all'esecuzione del codice di Test;
- Runtime – il codice provoca errori in fase di esecuzione (ad esempio "NullPointerException" o "IndexOutOfBoundsException");
- Compile – i sorgenti non presentano errori di compilazione;
- Response – la risposta dell'LLM non è attinente alle specifiche (es: risposta parziale).

Il campo "Errors" (**colonna I**) contiene gli errori descritti, e rappresenta il prompt inserito per effettuare l'iterazione di chiamata successiva (a meno che non sia stata soddisfatta una delle *condizioni di terminazioni* descritte al paragraffo 3.1 Metodologia).

3.2.1 Analisi dei Risultati

3.3.1.1 Analisi degli Score

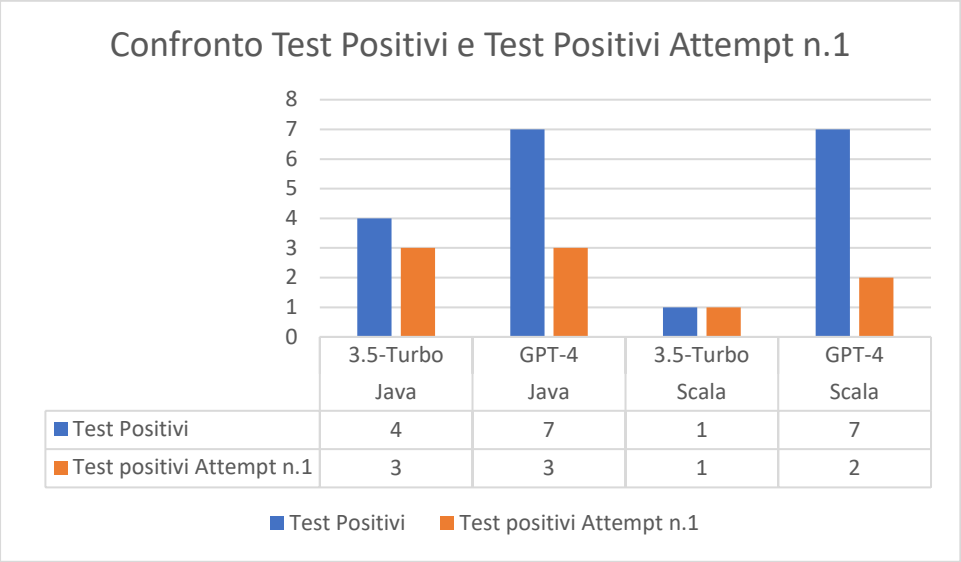
Si riporta di seguito un diagramma circolare rappresentante la distribuzione dei test passati con successo grazie al codice generato dagli LLM:



Possiamo da esso dedurre che, probabilmente a causa della popolarità del linguaggio, i risultati ottenuti dalla produzione di codice Java sono migliori in termini di attinenza ai test Junit presentati, riuscendo a passare con successo un totale di 11 istanze di test, rispetto alla controparte in linguaggio Scala 3, che ha avuto esito positivo in 8 casi.
È importante sottolineare che i test somministrati gli LLM facendo largo uso del collection framework di Java, non permettendo nella maggior parte dei casi l'utilizzo del collection framework nativo di Scala 3.

Dal grafico sopra riportato si puó anche evincere che il LLM Gpt-4 è piú competente nella produzione di codice di produzione a partire da codice di test Junit, poiché ad esso sono attribuibili 14 delle 19 istanze di test positivi.

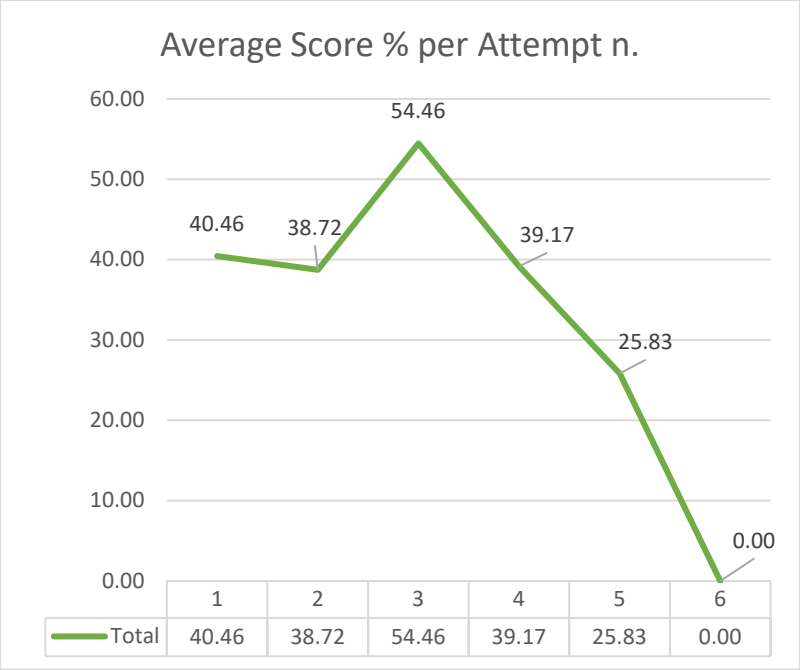
Si ritiene interessante effettuare un confronto tra le istanze di test soddisfatte alla prima iterazione :



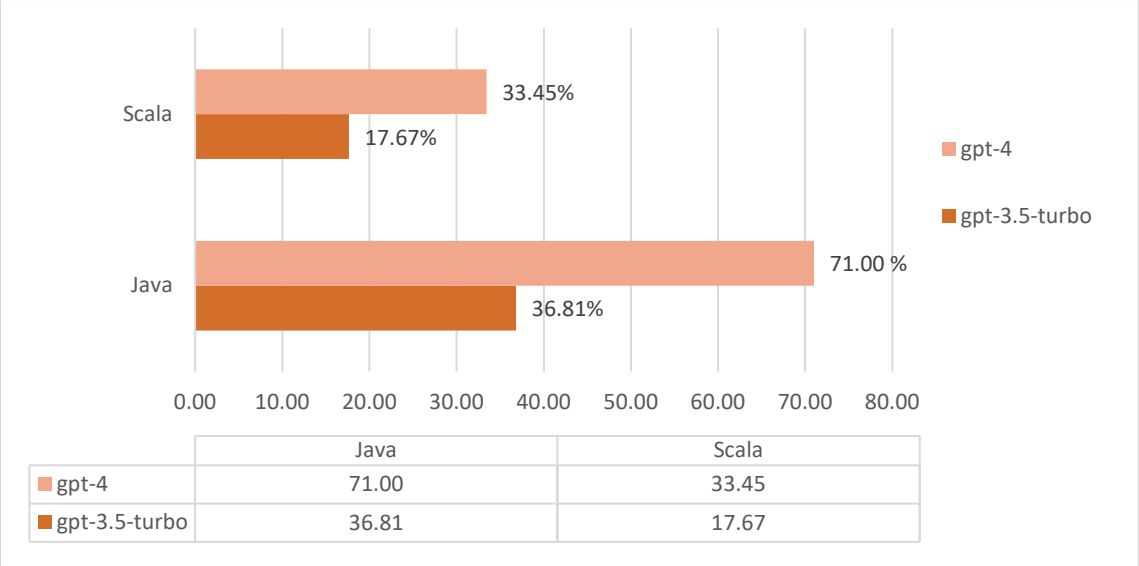
Dall'istogramma riportato si puó notare quanto GPT-4 sia migliore della versione precedente (GPT-3.5-Turbo) in termini di ricezione e comprensione delle modifiche da effettuare nelle iterazioni successive alla prime, migliorando i propri risultati fino al 350% (nel caso della coppia GPT-4 Scala, dove è riuscito a soddisfare 7 test nonostante alla prima iterazione le istanze corrette fossero solo 2);

Gpt-3.5-Turbo, al contrario, nonostante ottenga circa lo stesso numero di test positivi alla prima iterazione, non mostra un'elevata capacità di miglioramento a seguito di richieste di raffinamento successive (per quanto riguarda java c'è stato un miglioramento del 25%, mentre su Scala 3 non si rilevano variazioni nei risultati in termini di test positivi).

Sempre per quanto riguarda lo Score % in relazione al numero di iterazioni, si riporta di seguito un grafico ottenuto individuando le medie degli Score % per Attempt n., dal quale si puó notare come, mediamente, siano stati ottenuti i migliori risultati alla terza iterazione (escludendo la prima di somministrazione sorgenti):



Si riporta di seguito l'istogramma ottenuto calcolando la media dei migliori score per istanza di test, fissando le coppie LLM-Linguaggio:



Si puó quindi notare che mediamente le risposte fornite dagli LLM in esame siano piú pertinenti alle richieste nel caso in cui il linguaggio di riferimento sia Java, ottenendo una media di Score % pari al 71% per quanto riguarda Gpt-4 e 36.81% per Gpt-3.5-Turbo.

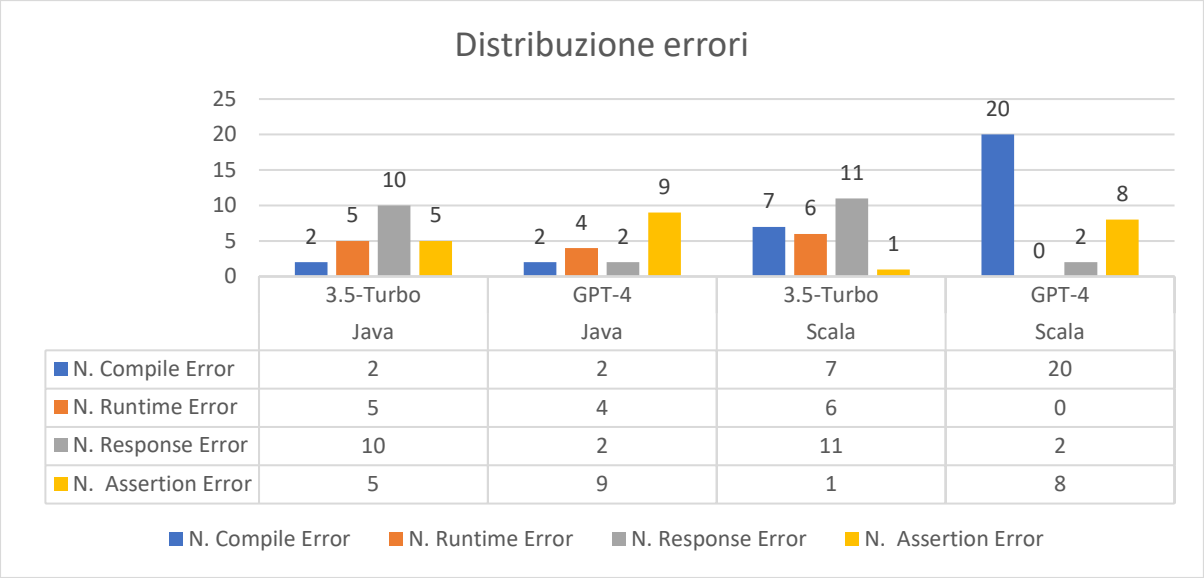
Per entrambi i linguaggi in esame, si puó dire che il modello Gpt-4 sia mediamente piú "affidabile" del suo predecessore, presentando una media di Score % quasi doppia rispetto a Gpt-3.5-turbo.

3.3.1.2 Analisi degli Errori

Analizzando gli errori riscontrati in sede ti testing del codice prodotto dagli LLM, sono state identificate 4 macro-tipologie:

- Errori di compilazione
- Errori a runtime
- Errori di asserzione
- Errori nella risposta fornita dall'LLM

Si lascia di seguito una raffigurazione della distribuzione degli errori rilevati, aggregati per LLM-Linguaggio:



	3.5-Turbo Java	GPT-4 Java	3.5-Turbo Scala	GPT-4 Scala
N. Compile Error	2	2	7	20
N. Runtime Error	5	4	6	0
N. Response Error	10	2	11	2
N. Assertion Error	5	9	1	8

Dal grafico riportato si puó notare che:

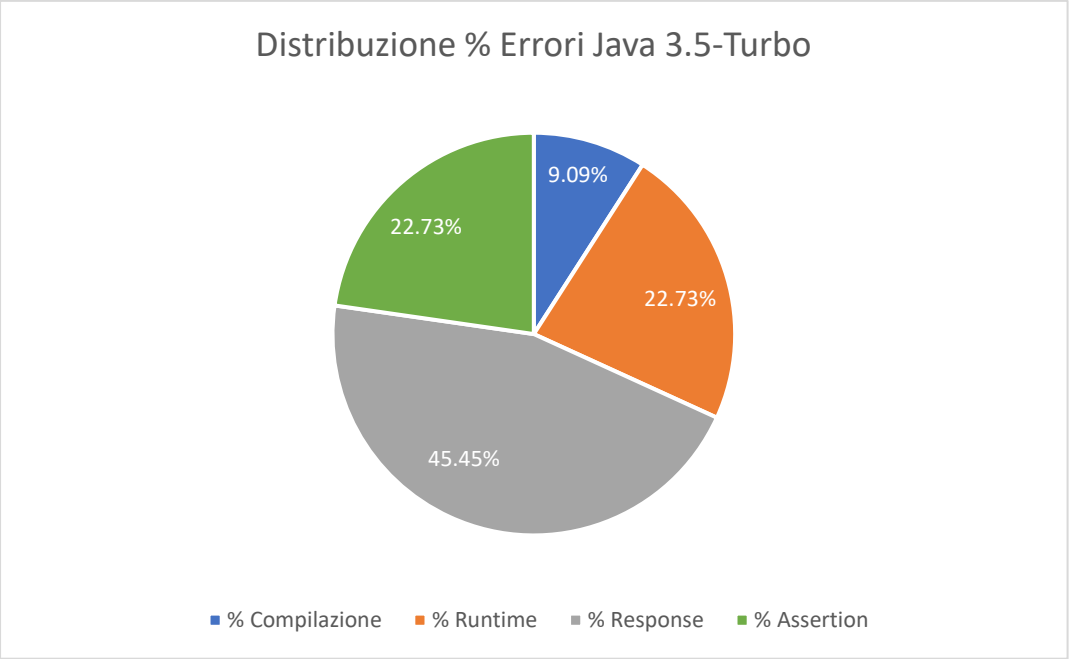
- Gpt-3.5-turbo è piú suscettibile ai Response Error (rappresentati dalle colonne di colore grigio, totalizzando 21 dei 25 errori di questa tipologia), probabilmente a causa delle minori dimensioni dell'LLM rispetto a Gpt-4;
- Gpt-4 presenta piú istanze di Assertion Error rispetto al predecessore (17 totali contro le 6 di Gpt-3.5-turbo) e, nonostante questo possa sembrare un dato poco incoraggiante, denota la capacità di Gpt-4 nel fornire spesso codice eseguibile e testabile mediante la libreria Junit;
- Confrontando il numero di errori dei due linguaggi in esame, si puó notare come entrambi i LLM abbiano piú incertezze nella produzione di codice Scala 3, presentando un totale di 55 errori rispetto ai 39 della controparte Java; la stessa

considerazione puó essere fatta analizzando le singole tipologie di errore, eccezion fatta per gli errori a runtime, che nel caso di Gpt-4-Scala ammontano a 0, ma “controbilanciati” dai 20 errori di compilazione.

È importante specificare che il codice prodotto spesso non comprendeva gli “import” necessari per la compilazione del codice, che sono quindi stati aggiunti dall’utente; è quindi verosimile aspettarsi un maggior numero di errori di compilazione nel caso questa procedura non fosse stata attuata.

Si presenta di seguito una rapida analisi della distribuzione degli errori per le singole coppie LLM-LINGUAGGIO.

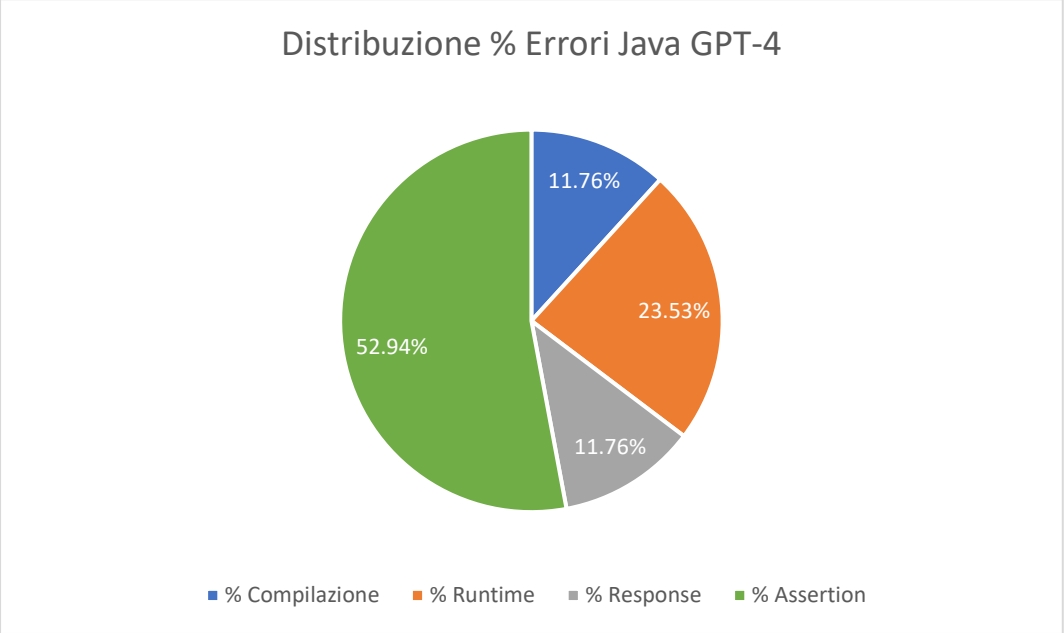
Distribuzione % errori Java Gpt-3.5-turbo



Nel caso della coppia Java-Gpt-3.5-turbo, ;a percentuale di errori di compilazione (9.09%) è relativamente bassa rispetto a quella di errori di runtime (22.73%). Questo puó suggerire che, una volta superati gli errori di compilazione, il codice generato ha una maggiore probabilità di essere eseguito con successo, ma potrebbero sorgere ulteriori problemi durante l'esecuzione effettiva, infatti la percentuale degli errori di asserzione è pari a quella di runtime (22.73%).

La percentuale di errori di risposta (45.45%) è significativamente più alta rispetto a quella delle altre tipologie, a riprova che il modello ha difficoltà a produrre risposte adeguate in modo consistente, anche quando l’input è sintatticamente corretto.

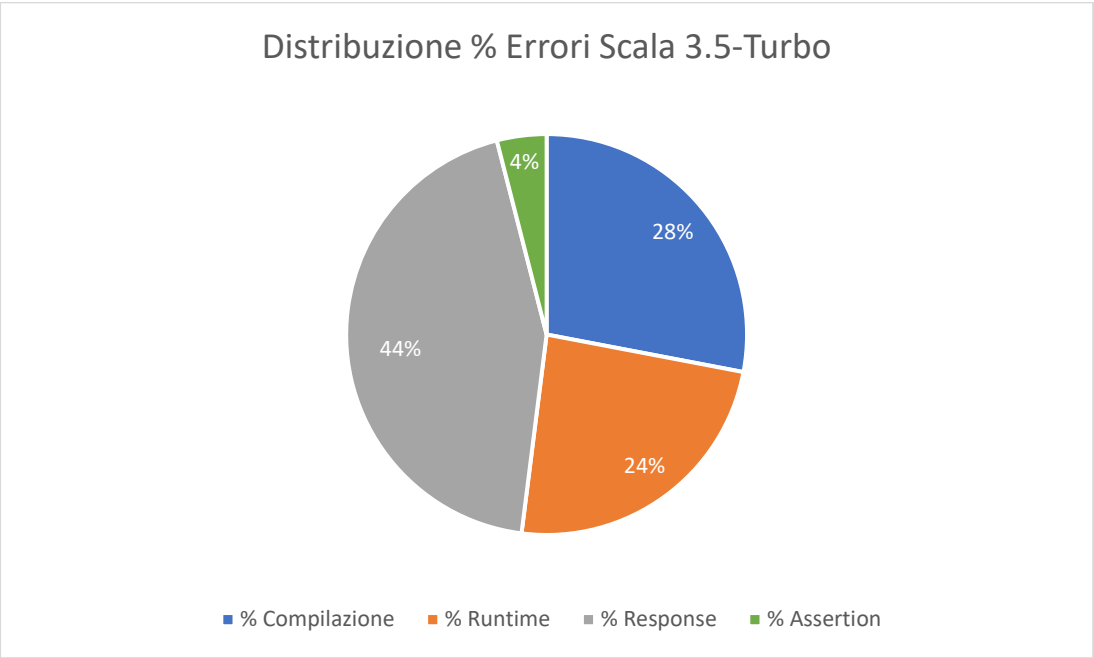
Distribuzione % errori Java Gpt-4



In riferimento alla coppia Java-Gpt-4 si puó notare come la percentuale di errori di compilazione (11.76%) e di runtime (23.53%) è rimasta sostanzialmente invariata rispetto a quella del modello precedente (9.09% e 22.73% rispettivamente).

La variazione maggiore rispetto alle medesime prove effettuate con il LLM Gpt-3.5-turbo è rappresentata dalla percentuale di errori di tipo Response (11.76%) e Assertion (52.94%), indicando un miglioramento nella comprensione e risoluzione delle specifiche fornite.

Distribuzione % errori Scala Gpt-3.5-turbo

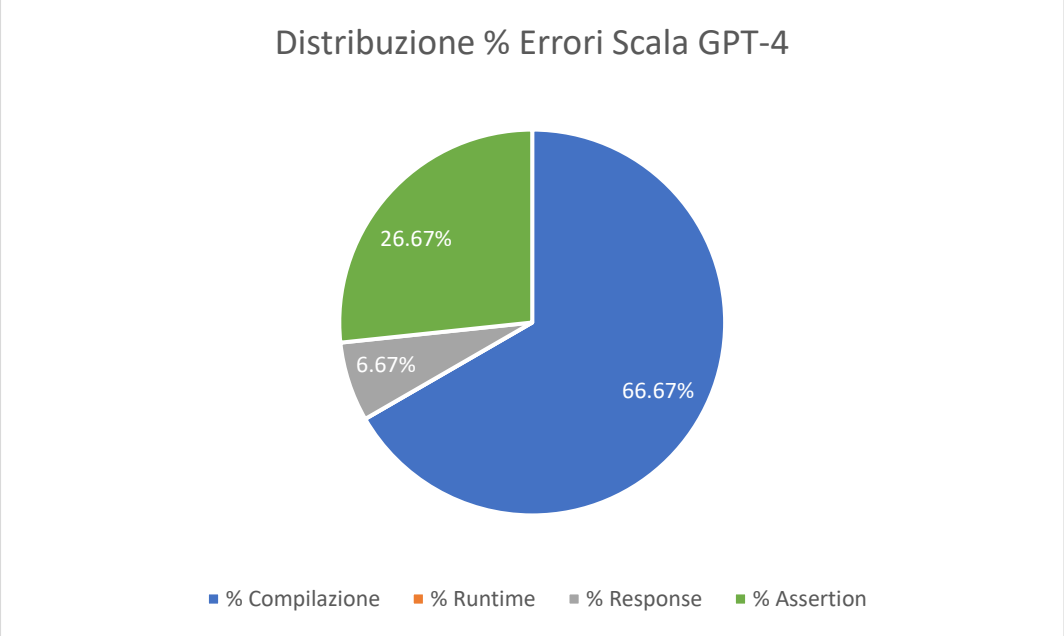


La percentuale di errori di compilazione (28.00%) è relativamente alta, indicando che il modello presenta delle difficoltà nella generazione di codice sintatticamente corretto in Scala 3. Anche la percentuale di errori di risposta (44.00%) è significativamente alta,

indicando come Gpt-3.5-turbo spesso non sia in grado di comprendere le specifiche e rispondere coerentemente ad esse.

La percentuale di errori di asserzione (4.00%) è relativamente bassa. Tuttavia, è importante considerare che un numero basso in questa tipologia indica un limitato numero di asserzioni nei test JUnit piuttosto che una maggiore precisione.

Distribuzione % errori Scala Gpt-4



La percentuale di errori di runtime (0.00%) è significativamente più bassa rispetto al modello 3.5-Turbo (24.00%), ad indicare una maggiore stabilità nella gestione degli errori durante l'esecuzione del codice generato. La percentuale di errori di risposta (6.67%) è notevolmente più bassa rispetto al modello precedente (44.00%), suggerendo che il modello GPT-4 potrebbe è più efficace nella produzione di risposte adeguate in Scala 3.

Nonostante il miglioramento rispetto al predecessore, il LLM Gpt-4 produce molto spesso codice che presenta errori di compilazione, sintomo di una ridotta capacità nella comprensione sintattica del linguaggio; è importante specificare, come fatto in precedenza, che i test Junit forniti fanno largo uso del collection framework di Java, e nella risoluzione di questi utilizzando Scala3 è fondamentale eseguire molteplici conversioni.

3.2.1.3 Considerazioni Generali

Volendo formulare delle condizioni generali sulle per formance dei LLM in esame nel contesto della produzione di codice in linguaggio Java e Scala 3 si puó notare come, in molti casi, si sono presentati errori di compilazione, gli errori comuni includono problemi di tipo, eccezioni inattese, e mancanza di importazioni necessarie, segno che i modelli non sono ancora in grado di simulare l'esecuzione del codice fornito.

In generale, questi risultati indicano che la generazione automatica di codice può essere soggetta a vari problemi e richiede attenzione nella correzione degli errori e nella gestione delle specifiche dei test. È importante rivedere attentamente il codice generato e correggerlo manualmente laddove necessario.

L'affidabilità a runtime (e per quanto riguarda le Assertion all'interno dei test) è migliore rispetto all'affidabilità di compilazione, indicando che, seppur impossibilitati ad eseguire il codice prodotto, i LLM sono in grado di strutturarne correttamente la logica interna.

Globalmente, i modelli mostrano risultati simili, suggerendo che il passaggio da GPT-3.5 Turbo a GPT-4 ha portato a miglioramenti modesti, ma significativi in termini di riduzione degli errori di tipo "Response".

3.3.1.4 Analisi Specifiche

Si presentano di seguito alcune considerazioni specifiche per ogni coppia LINGUAGGIO-MODELLO.

Java GPT-3.5-Turbo:

Il modello presenta una bassa percentuale di test positivi nonostante la popolarità del linguaggio, con particolare enfasi sugli errori di tipo "Response", fornendo all'utente risposte completamente errate e fuori contesto.

Java GPT-4:

Rispetto al suo predecessore, il GPT-4 mostra risultati simili in termini di percentuale di test positivi e affidabilità complessiva. Tuttavia, si nota una riduzione significativa degli errori a di tipo "Response", indicando un miglioramento nella robustezza.

Scala GPT-3.5 Turbo:

Questo modello presenta una bassa percentuale di test positivi, indicando difficoltà nella generazione di codice funzionale. La maggior parte degli errori emergono in fase di compilazione, suggerendo una limitata robustezza del codice generato, e nel contenuto delle risposte, indicando una non adeguata competenza per quanto riguarda lo sviluppo di codice Scala.

Da notare anche il numero elevato di "response error", che indica situazioni di "blocco" da parte del LLM, che non è in grado di fornire una, seppur errata, risposta

Scala GPT-4:

Il modello GPT-4 in ambito di sviluppo Scala è leggermente migliore rispetto al GPT-3.5 Turbo, con una maggiore percentuale di test positivi. Tuttavia, le prestazioni generali rimangono modeste, con sfide particolari in fase di compilazione.

3.3 Analisi del codice

Si riporta di seguito qualche esempio di codice

3.3 Analisi dei Costi

Anche in relazione dei risultati ottenuti, si ritiene importate fare un excursus per quanto riguarda i costi [POAI] delle chiamate ai due modelli utilizzati (ChatGPT3.5-Turbo e ChatGPT-4) :

Nello specifico, la spesa totale per eseguire i test presentati ammonta a (circa) **18,15€**, ripartiti come segue:

- ChatGPT-4 : **17.42€** -> 96% del totale
- ChatGPT-3.5-Turbo : **0.73€** -> 4% del totale

Considerando che le performance di GPT-4 riscontrate (nell'ambito di questo progetto) non si ritengono commisurate all'aumento della spesa, si consiglia l'utilizzo di ChatGPT-3.5-Turbo, effettuando molteplici cicli di correzione degli errori (soprattutto per quanto riguarda la produzione di codice Java) se non ci si trova all'interno di un contesto aziendale, dove la spesa sostenuta puó essere giustificabile da una riduzione dei tempi necessari per produrre codice adeguato (infatti, mediamente GPT-4 fornisce risposte e migliori con un minor numero di richieste rispetto al modello precedente.).

4. Conclusioni

In conclusione, questa relazione ha esplorato l'applicazione dei Large Language Models (LLM) nel processo di sviluppo del software, focalizzandosi sulla generazione automatica di codice a partire da test scritti in linguaggio Java utilizzando la libreria JUnit. I modelli considerati sono ChatGPT-3.5-Turbo e ChatGPT-4 forniti da OpenAI.

L'uso di LLM nella generazione di codice offre diversi vantaggi, tra cui la riduzione dei tempi di sviluppo, la minimizzazione degli errori umani, l'aumento della produttività e la facilità nella manutenzione del codice. Tuttavia, sono presenti sfide e considerazioni etiche, come la comprensione del contesto, la sicurezza, l'accettazione della comunità e la responsabilità nell'utilizzo di questa tecnologia.

L'analisi dei risultati ha evidenziato che la generazione automatica di codice presenta sfide significative, come errori di compilazione e difficoltà nella comprensione del contesto. È stata sottolineata l'importanza della revisione umana del codice generato per garantire la sua correttezza e coerenza; inoltre, sono state fornite considerazioni sugli aspetti economici legati alle chiamate ai modelli, consigliando l'utilizzo di ChatGPT-3.5-Turbo per ottimizzare i costi a discapito di un peggioramento delle prestazioni.

In generale, l'uso dei LLM nella generazione di codice rappresenta una prospettiva promettente, ma richiede un approccio attento e responsabile, considerando le sfide tecniche e le implicazioni etiche legate a questa tecnologia.

5. Riferimenti

[APLC] Analisi prompt per generare codice : **https://arxiv.org/pdf/2311.07599.pdf**

[BCML] Big Code Models Leaderboard : **https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard**

[DLLM] Definizione di LLM **:** **https://www.techtarget.com/whatis/definition/large-language-model-LLM**

[DLCO] Documentazione libreria Config typesafe : **https://github.com/lightbend/config**

[DLSS] Documentazione libreria Scala Swing : **https://index.scala-lang.org/scala/scala-swing**

[ESUL] Tesi LLM per la generazione di test : **https://arxiv.org/pdf/2305.00418.pdf**

[HTUL] Come utilizzare gli LLM : **https://medium.com/coderhack-com/here-is-section-i-of-the-outline-in-markdown-format-cf20c3dc229**

[ICLL] Introduzione di CodeLLAMA : **https://ai.meta.com/blog/code-llama-large-language-model-coding/**

[ISCO] Introduzione StarCoder : **https://huggingface.co/blog/starcoder**

[JUNIT] Documentazione libreria Junit Java : **https://javadoc.io/doc/junit/junit/latest/index.html**

[LOAA] Libreria d'interfacciamento API OpenAi : **https://platform.openai.com/docs/libraries/community-libraries**

[SOAI] Sito ufficiale OpenAi : **https://openai.com/**

[OASC] Repository libreria Openai-scala-client: **https://github.com/cequence-io/openai-scala-client**

[OAIM] Documentazione modelli OpenAi : **https://platform.openai.com/docs/models**

[OPAI] Sito ufficiale OpenAi : **https://openai.com/**

[PIMV] Pagina istituzionale (Unibo) Mirko Viroli : **https://www.unibo.it/sitoweb/mirko.viroli**

[POAI] Pricing API OpenAi: **https://openai.com/pricing**

[POOP] Pagina istituzionale (Unibo) corso di OOP : **https://www.unibo.it/it/didattica/insegnamenti/insegnamento/2023/378219**

[REPO] Repository di progetto : **https://github.com/MicheleTorroni/LlmTestApp**

[LLTE] Esempio LLM per la generazione dei test : **https://medium.com/@taiyuanz/can-llm-help-swe-write-test-cases-4d1cd3b51b3e**

[TEST] Foglio di calcolo contenente i risultati dei test : **https://github.com/MicheleTorroni/LlmTestApp/blob/main/doc/Tests.xlsx**