Università degli Studi di Milano Bicocca

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di laurea in Informatica

Estendere MultiMind: generazione di Unit Test attraverso IA

Relatore: Prof.ssa Daniela Micucci

Correlatore: Dott.ssa Benedetta Donato

Relazione della prova finale di:
Tommaso Rezzani
Matricola 874903

Anno Accademico 2024-2025

Arrivato alla fine di questo percorso è l'ora dei titoli di coda e come si è soliti fare è giusto ringraziare chi è stato al mio fianco:

Ringrazio la professoressa Daniela Micucci e la dottoressa Benedetta Donato per l'opportunità di partecipare a questo progetto e per gli aiuti dati nella stesura della tesi e nella conclusione del mio percorso.

Ringrazio i compagni informatici con cui ho condiviso tante emozioni ed esperienze tra lezioni, esami, studio di gruppo e molte, forse troppe, pause caffè.

Ringrazio gli amici, di piazza e non solo, con cui negli ultimi anni ho fatto moltissime avventure e che riescono sempre a farmi fare un sospiro di sollievo e farmi dimenticare tutte le preoccupazioni.
Un grazie speciale va ad alcuni di voi che nonostante siano passati molti anni continuate a supportarmi e sopportarmi sia nei momenti belli che in quelli pessimi.

Mamma e Papà, grazie di avermi sempre aiutato e supportato in tutto quello che faccio e grazie per rimettermi in riga quando faccio scelte sbagliate.
Nonostante le difficoltà e la lentezza alla fine ho finito anche questo percorso e spero siate orgogliosi di me, vi voglio bene.

Il grazie più grande però è per te, Camilla. Anche se non riusciamo a stare più di tre giorni senza urlarci addosso, i tuoi consigli e le tue parole d'aiuto, ogni tanto dette gentilmente, mi hanno sempre fatto superare molti momenti brutti e difficili.

Per ultimo grazie a te, Tommaso. Anche se gli ultimi mesi sono stati pessimi non ti sei mai arreso e hai dato tutto per arrivare al tuo obiettivo. Adesso con la corona in testa puoi finalmente alzare lo sguardo e dire "Nonno ce l'ho fatta, questa è per te"

Indice

List of Figures 6

List of Tables 7

Introduzione 8

Capitolo 1: L'impiego delle Intelligenze Artificiali come assistenti allo sviluppo 10

1.1 Gli assistenti ad oggi disponibili 10

1.2 MultiMind: Una nuova visione per lo sviluppo AI-Assisted 11

1.3 L'obiettivo: Generazione Automatica delle Test Unit 12

Capitolo 2: Tecnologie Utilizzate: Visual Studio Code, LLMs e API 14

2.1 Visual Studio Code 14

2.1.1 Panoramica di Visual Studio Code 14

2.1.2 Estensioni in Visual Studio Code 14

2.1.3 Sviluppo delle Estensioni nel contesto del progetto 16

2.2 Large Language Models e Google Gemini 16

2.2.1 Panoramica dei Large Language Models 16

2.2.2 Google Gemini: Implementazione e Vantaggi 17

2.2.3 Ruolo dei LLM nel Progetto 17

2.3 Architettura della Soluzione 18

2.3.1 Panoramica e Componenti principali dell'Architettura 18

2.3.2 Flusso di Esecuzione e Comunicazione 19

Capitolo 3: Test Unit Generation: Implementazione 22

3.1 Architettura del Sistema 22

3.2 Analisi di testUnitGeneration.action 22

3.2.1 Il Cuore del Processo: processTestGeneration() 23

3.2.2 Gestione dei Percorsi: calculateRelativeImportPath() 25

3.3 Analisi di testUnitGeneration.task 26

3.3.1 Il Metodo chiave: activate() 26

3.3.2 I Metodi di Supporto 28

3.4 Analisi di Extension 30
3.5 Gestione degli Errori e Sicurezza del codice 31
Capitolo 4: Generate Test Unit: Manuale Utente 32
4.1 Panoramica e Installazione 32
4.1.1 Quick Start e prerequisiti 32
4.2 Installazione ed Utilizzo 32
4.2.1 Configurazione del Progetto 32
4.2.2 Generazione Test Unit da Selezione 33
4.2.3 Generazione Test da File Completo 34
4.3 Supporto tecnico e Funzionalità automatiche 35
4.3.1 Problemi riscontrabili e risoluzioni 35
4.3.2 Gestione automatica delle directory 35
4.3.3 Note tecniche 36
4.4 Riproducibilità e disponibilità del codice 36
Capitolo 5: Conclusioni 37
Bibliografia e Sitografia 39

List of Figures

Figura 1: Architettura della soluzione e flusso d'esecuzione del lavoro 20
Figura 2: prima parte del codice del metodo processTestGeneration() in testUnitGeneration.action.ts 23
Figura 3: seconda parte del codice del metodo processTestGeneration() in testUnitGeneration.action.ts 24
Figura 4: terza parte del codice del metodo processTestGeneration() in testUnitGeneration.action.ts 24
Figura 5: codice del metodo calculateRelativeImportPath() in testUnitGeneration.action.ts 25
Figura 6: prima parte del codice del metodo activate() in testUnitGeneration.task.ts..... 27
Figura 7: seconda parte del codice del metodo activate() in testUnitGeneration.task.ts 27
Figura 8: codice del metodo extractImportPath() in testUnitGeneration.task.ts 28
Figura 9: codice del metodo buildTestPrompt() in testUnitGeneration.task.ts..... 29
Figura 10: codice del metodo processGeneratedTest() in testUnitGeneration.task.ts 29
Figura 11: il metodo activate() registra tutti i plugin disponibili in MultiMind 30
Figura 12: L'opzione Generate Test Unit che appare quando viene rilevato del testo evidenziato premendo il tasto destro..... 33
Figura 13: barra a schermo che indica quando manca al completamento della scrittura della test unit 34
Figura 14: codice del metodo che cerca se esiste già una cartella in base al nome 35
Figura 15: Struttura della repository dei file con generazione automatica 36

List of Tables

Tabella 1: triple Linguaggio-Framework-Estensione presenti nel metodo generateTestFilePath() 26
Tabella 2: Tabella che indica la stringa per l'installazione del framework specifico di ogni linguaggio 32
Tabella 3: Tabella che indica alcuni dei possibili messaggi e cause d'error, con soluzioni annesse 35

Introduzione

Lo sviluppo software degli ultimi anni si è evoluto in maniera notevole grazie all'introduzione delle intelligenze artificiali e alla diffusione dei Large Language Models. Per facilitare alcuni processi della programmazione, come ad esempio scrivere codice, suggerire soluzioni o spiegare blocchi logici, sono nati assistenti virtuali come GitHub Copilot, ChatGPT, Tabnine e Amazon CodeWhisperer che si sono affermati come ottimi strumenti in grado di aiutare gli utenti. Tuttavia, se analizziamo più in profondità questi strumenti, vengono alla luce alcune criticità significative che ne limitano il potenziale, quali ad esempio la dipendenza da singoli modelli AI che fornisce una sola proposta, e che spesso non espone alternative più efficienti, e la scarsa integrazione con il contesto reale di sviluppo costringe chi le usa a copiare e incollare manualmente porzioni di codice.

L'applicazione di questi assistenti AI esterni risulterebbe innovativa ed efficace in una delle aree trascurate dal loro impiego, quella della generazione automatica delle Test Unit. Questo utilizzo potrebbe considerarsi particolarmente vantaggioso se si considera che il testing rappresenta un passaggio indispensabile nel processo di sviluppo e la creazione manuale di test unitari costituisce un'attività ripetitiva e monotona che spesso viene eseguita in maniera superficiale e che assorbe tempo prezioso.

La soluzione proposta in questo progetto nasce dall'idea di colmare questa lacuna precisa andando ad ampliare MultiMind, un'estensione per Visual Studio Code che permette l'utilizzo coordinato di diversi modelli AI, per automatizzare la creazione di test unitari. Il sistema implementato non si limita a una semplice generazione dei test, ma utilizza diversi modelli AI integrati in sequenza per individuare e comprendere il codice da testare, e successivamente generare automaticamente i test appropriati, adattandosi a diversi linguaggi di programmazione e framework di testing usando meccanismi di rilevamento automatico che garantiscono una perfetta integrazione nel sistema del progetto esistente. Questa soluzione rappresenta anche come lo sviluppo AI-assisted può superare i limiti degli assistenti AI attuali usando un approccio più flessibile e cooperativo, in modo tale da liberare gli sviluppatori da compiti ripetitivi e noiosi e permettergli di concentrarsi sugli aspetti più creativi e strategici della programmazione

La relazione è strutturata nel seguente modo:

Capitolo 1: In questo capitolo si identificano le criticità degli assistenti AI attuali nello sviluppo e si evidenzia la necessità di automatizzare la generazione di test unitari, presentando poi la soluzione proposta interna a MultiMind.

Capitolo 2: In questo capitolo si descrivono le tecnologie coinvolte, da Visual Studio Code ai Large Language Models, per poi definire l'architettura modulare che sostiene l'intera soluzione.

Capitolo 3: Questo capitolo contiene la documentazione e l'implementazione tecnica del sistema di generazione automatica dei test, e analizza nel dettaglio le classi Action e Task e i loro meccanismi di funzionamento.

Capitolo 4: In questo capitolo è inserito il manuale per guidare l'utente nell'installazione e nell'utilizzo dell'estensione.

Capitolo 5: Il capitolo è dedicato all'analisi dei risultati raggiunti e all'individuazione delle direzioni possibili per sviluppi o miglioramenti futuri.

Capitolo 1: L'impiego delle Intelligenze Artificiali come assistenti allo sviluppo

1.1 Gli assistenti ad oggi disponibili

Negli ultimi anni, l'impiego dell'intelligenza artificiale nello sviluppo software ha subito una notevole evoluzione grazie alla diffusione dei Large Language Models (LLMs). Strumenti come GitHub Copilot, ChatGPT, Amazon CodeWhisperer e Tabnine si sono affermati come assistenti virtuali in grado di aiutare lo sviluppatore nella scrittura del codice suggerendo soluzioni, completando automaticamente funzioni, o spiegando blocchi logici di qualsiasi complessità.

L'introduzione di questi strumenti rappresenta un importante passo avanti verso una programmazione più versatile dal punto di vista di accessibilità, velocità e semplicità. Tuttavia, se analizziamo in maniera più approfondita le soluzioni ad oggi disponibili, possiamo notare la presenza di alcune criticità che ne limitano il potenziale, soprattutto in contesti produttivi di alto livello o su progetti di lunga durata.

Una delle limitazioni ai fattori di sviluppo attraverso l'utilizzo di questi strumenti, riguarda la dipendenza da un singolo modello AI e la necessità di appoggiarsi a servizi esterni a pagamento. Ad esempio, se si analizzano ChatGPT o Copilot si riscontra un modello basato su un numero di utilizzi fissi per molte funzionalità e per poterle usare continuamente è spesso necessario acquistare un abbonamento mensile con varie fasce di prezzo.

In secondo luogo, molti assistenti virtuali lavorano in modo unilaterale e chiuso fornendo una sola proposta alla volta, e spesso non espongono la presenza di altre possibili soluzioni, magari più efficienti e leggibili, con cui permettere un confronto. In caso vengano riscontrate ambiguità nel codice o vengano effettuate richieste complesse, questa "voce unica" espone allo sviluppatore il rischio di overreliance, portandolo ad accettare la soluzione proposta dall'assistenze senza ulteriori controlli o senza applicare il proprio pensiero critico.

Un altro limite delle AI è legato alla scarsa integrazione diretta con il contesto reale di sviluppo. Sebbene alcuni strumenti come Copilot riescano ad inferire con il file attivo, perché sono utilizzati direttamente sull'IDE, o a sfruttare una parte delle informazioni dell'ambiente di lavoro, è molto raro che l'assistente possa accedere in modo strutturato e automatico all'intero codice sorgente del progetto portandolo quindi ad avere una visione frammentata. È impossibile accedere per gli assistenti ai file di configurazione, ai codici sorgente o allo storico dei commit per poter dare risposte coerenti al contesto, quindi, in molte situazioni, l'utente deve copiare e incollare manualmente porzioni di codice o fornire descrizioni il più dettagliate possibile di ciò che vuole per poter ottenere una risposta il più precisa possibile. Questo approccio, oltre a rallentare il flusso di lavoro, può diventare insostenibile nei progetti complessi o di grandi dimensioni. Per gli LLM disponibili via API, inoltre, inviare grandi quantità di codice come contesto può risultare tecnicamente complicato ed economicamente proibitivo, poiché influisce direttamente sul costo di ogni richiesta.

Anche la personalizzazione è ancora una sfida aperta. Nella maggior parte dei casi, gli utenti possono solo modificare il comportamento degli assistenti o definirne le strategie operative tramite dei promp di testo e non in maniera modulare così da avere un adattamento reale. Non è possibile, ad esempio, specificare che un assistente privilegi la leggibilità del codice rispetto all'ottimizzazione delle prestazioni, o decidere che utilizzi una determinata libreria in maniera automatica, ma solo tramite un comando testuale che dovrà essere ripetuto ogni volta. La configurazione avanzata è spesso assente, o richiede competenze tecniche elevate e modifiche non supportate ufficialmente.

In sintesi, gli attuali assistenti AI, sebbene offrano un importante valore aggiunto, hanno ancora vari limiti in termini di apertura, interazione, confronto e autonomia. Questi limiti non invalidano il loro contributo e il materiale da loro proposto, ma suggeriscono la necessità di un'evoluzione, capace di abbracciare una visione più flessibile e cooperativa.

1.2 MultiMind: Una nuova visione per lo sviluppo AI-Assisted

La necessità di superare le limitazioni sopra descritte ha trovato una soluzione grazie allo sviluppo di MultiMind, un'estensione per Visual Studio Code, che rivoluziona il paradigma dell'integrazione dell'intelligenze artificiali nei vari processi di sviluppo. MultiMind nasce dalla consapevolezza che la complessità e la diversità dei compiti richiesti nello sviluppo moderno non possono essere gestite con efficienza da un singolo modello linguistico, indipendentemente da quanto questo sia avanzato.
Il framework MultiMind introduce un'architettura modulare che permette l'utilizzo coordinato di diversi Large Language Models all'interno dell'ambiente di sviluppo Visual Studio Code. Questa innovazione rappresenta una risposta diretta alla frammentazione e alla rigidità degli strumenti AI esistenti, offrendo un'unica piattaforma dove multiple intelligenze artificiali possono collaborare per affrontare compiti articolati che richiedono competenze specifiche differenti.

L'architettura di MultiMind si basa su un sistema di parti intercambiabili che comunicano attraverso interfacce standardizzate. Al centro del sistema troviamo i Driver, componenti specializzati che gestiscono la comunicazione con diversi servizi come GPT, Gemini e altri modelli emergenti. Questi driver non si limitano a porre domande e dare risposte, ma implementano delle logiche in modo che ciascun modello possa adattarsi al meglio e sfruttare le proprie caratteristiche e punti di forza. Il Driver Manager coordina queste interazioni, fungendo da gestore che può decidere dinamicamente quale modello utilizzare per ogni specifica operazione, basandosi su criteri di appropriatezza, disponibilità e performance.

La gestione delle attività più complesse è affidata ai Task Manager, componenti che rappresentano l'innovazione più significativa di MultiMind. Questi sistemi permettono di definire workflow articolati che possono coinvolgere multiple AI in sequenza o in parallelo, creando pipeline di elaborazione sofisticate. Un esempio è la generazione di codice seguita dalla sua validazione qualitativa: il Task Manager può coordinare un primo LLM specializzato nella generazione di codice con un secondo modello focalizzato sull'analisi della

qualità, creando un processo iterativo di raffinamento che continua fino al raggiungimento di standard qualitativi predefiniti.

L'integrazione nativa con Visual Studio Code rappresenta un altro aspetto fondamentale dell'innovazione di MultiMind. Il framework non si limita a funzionare come un plugin esterno, ma sfrutta appieno le API native dell'editor per dare all'utente un utilizzo lineare e guidato. Le Actions, componenti che gestiscono l'interfaccia tra l'utente e il sistema, permettono l'attivazione di workflow AI complessi attraverso interazioni naturali nell'ambiente di sviluppo, come la selezione di codice e l'utilizzo di menu contestuali.

La modularità del sistema facilita in modo importante l'estensibilità, e permette l'aggiunta di nuove funzionalità e l'integrazione di altre funzioni senza richiedere modifiche strutturali al core del framework. Questa caratteristica rende MultiMind non solo un tool attuale, ma una piattaforma evolutiva capace di adattarsi al rapido progresso nel campo dell'intelligenza artificiale.

1.3 L'obiettivo: Generazione Automatica delle Test Unit

La capacità di utilizzo di multiple intelligenze artificiali specializzate implementata in MultiMind permette di automatizzare svariati compiti durante il processo di sviluppo, ma allo stesso tempo ne mette in evidenza le limitazioni attualmente presenti nello strumento. L'implementazione di base di MultiMind si concentra principalmente su funzionalità generiche come generazione di documentazione automatica, valutazione e miglioria del codice e un'interfaccia chat per comunicare con i modelli AI. Se da un lato le funzionalità presenti dimostrano in maniera evidente le potenzialità dell'architettura e l'innovazione contenuta al suo interno, allo stesso tempo, le stesse non vengono utilizzate in altre aree critiche del processo di sviluppo alle quali l'applicazione di questo approccio innovativo porterebbe evidenti benefici.

Una delle lacune riscontrate riguarda l'assenza completa di supporto per la generazione automatica di test unitari. Questa carenza è importante dal momento che il testing di un codice rappresenta un passaggio indispensabile all'interno del processo di sviluppo e la creazione manuale di test unitari rappresenta un'attività ripetitiva che assorbe tempo e per tale ragione viene eseguita senza la necessaria attenzione.

Il lavoro che è stato svolto in questo progetto nasce proprio dall'idea di ottimizzare la creazione delle Test Unit, attraverso un plugin interno a MultiMind dedicato alla generazione automatica di test unitari, con l'obiettivo di garantire la correttezza e funzionalità del codice in modo rapido ed efficiente.

La soluzione che si propone mira a risolvere il problema della scrittura manuale dei test unitari, liberando chi sviluppa da compiti ripetitivi e monotoni per permettere di concentrarsi sugli aspetti più creativi e strategici del proprio lavoro. Il sistema implementato utilizza diversi modelli AI integrati e in sequenza. Il modo in cui agisce è diviso in due momenti: nel primo step individua e comprende il codice da testare, e sulla base di questo, nello step successivo vengono generati automaticamente i test appropriati.

Il sistema che è stato sviluppato si integra in maniera efficace nell'utilizzo di Visual Studio Code, permettendo a chi programma di generare test unitari per una classe o per uno o più metodi attraverso un semplice comando. Particolare attenzione è stata rivolta alla capacità del sistema di adattarsi dinamicamente a diversi ambienti di lavoro, implementando meccanismi di rilevamento automatico del linguaggio di programmazione e del framework di testing in uso, in modo da garantire che i test generati si integrino perfettamente nel sistema preesistente.

Capitolo 2: Tecnologie Utilizzate: Visual Studio Code, LLMs e API

2.1 Visual Studio Code

2.1.1 Panoramica di Visual Studio Code

Visual Studio Code (VS Code) è un editor di codice sorgente versatile e adottato su larga scala, sviluppato da Microsoft, e progettato in maniera meticolosa per fornire un ambiente di sviluppo potente ma che non penalizza le prestazioni della macchina, offrendo un grande supporto per moltissimi linguaggi di programmazione [1].

Visual Studio Code è basato su tecnologie web, e sfrutta principalmente il framework Electron che consente a VS Code di funzionare come un'applicazione multipiattaforma, in modo da poter operare senza problemi su sistemi operativi Windows, macOS e Linux. Lo sviluppo dell'editor stesso utilizza come linguaggi TypeScript e JavaScript, ed è incentrato sulla modularità e sull'ottimizzazione delle prestazioni [1].

Oltre al framework Electron, che fornisce la base per l'esecuzione come applicazione desktop, l'architettura di VS Code si basa su altri componenti, di cui i principali sono Monaco Editor e Language Server Protocol (LSP), che ne sostengono la funzionalità.

Il Monaco Editor rappresenta il core dell'editor e assume il ruolo di un potente e sofisticato motore che racchiude le capacità di editing di VS Code e fornisce molteplici opzioni per la manipolazione del codice. Il protocollo standardizzato, implementato attraverso LSP, permette di facilitare la comunicazione efficiente tra l'editor e gli strumenti specifici del linguaggio per sbloccare funzioni come il completamento automatico intelligente, il controllo degli errori in tempo reale e altre possibilità [1].

VS Code si distingue per il suo ampio supporto multi-linguaggio, includendo C++, C#, F#, Python e molti altri linguaggi di programmazione. L'editor offre strumenti di sviluppo completi come IntelliSense per il completamento intelligente del codice, un ambiente di debugging integrato e il supporto nativo per il controllo versione tramite Git [6]. Questa versatilità lo rende uno degli IDE più sofisticati e utilizzati nel panorama dello sviluppo software contemporaneo.

2.1.2 Estensioni in Visual Studio Code

L'adattabilità e l'utilizzo diffuso di Visual Studio Code è dovuto al suo ecosistema di estensioni considerate sicure e affidabili dagli utilizzatori. Le estensioni di VS Code sono componenti software modulari che aumentano le capacità principali dell'editor. Sono progettate per integrarsi alla perfezione dentro VS Code, in modo da poter fornire funzioni, strumenti e personalizzazioni aggiuntive che soddisfano una vasta gamma di esigenze e preferenze di sviluppo [1].

Il meccanismo di base con cui operano le estensioni si incentra su un'Application Programming Interface (API) ben definita. Questa API fornisce un set di interfacce e protocolli standardizzati che permettono alle estensioni di interagire con i componenti principali dell'editor, accedere ai dati, modificare il loro comportamento, manipolare il buffer di testo, visualizzare elementi dell'interfaccia utente, interagire con processi esterni e comunicare con server remoti.

Per garantire l'integrità del sistema e la sicurezza, Visual Studio Code implementa un modello di esecuzione sandbox per le estensioni, limitando le modifiche non autorizzate alle funzionalità chiave dell'IDE. Inoltre, Microsoft gestisce una piattaforma ufficiale di distribuzione delle estensioni, il Visual Studio Marketplace, che impone un rigoroso processo di validazione per assicurare qualità, compatibilità e conformità di sicurezza.

L'architettura delle estensioni è costruita attorno al Visual Studio SDK (Software Development Kit), che fornisce un insieme completo di librerie, interfacce e servizi. Gli sviluppatori possono estendere Visual Studio Code attraverso l'uso di due meccanismi principali: il Managed Extensibility Framework (MEF) e il sistema Visual Studio Package (VSIX) [6]. Il MEF consente la creazione di miglioramenti leggeri e modulari permettendo agli sviluppatori di definire e comporre componenti in modo dinamico, senza necessità di deployment di pacchetti completi. Il formato VSIX rappresenta il meccanismo standard di packaging per le estensioni, racchiudendo i metadati necessari, i binari e i file di configurazione richiesti per il deployment.

Le estensioni non sono semplici parti aggiuntive, ma diventano parte integrante di VS Code permettendogli di adattarsi alle diverse esigenze degli sviluppatori [1]. Il supporto linguistico offre possibilità come sottolineatura della sintassi, completamento automatico intelligente del codice e capacità di debugging per vari linguaggi di programmazione. Altri strumenti inclusi consentono ad esempio di rilevare problemi nello stile del codice, stabilire una formattazione coerente del codice in tutto il progetto e integrare tecnologie e sistemi di controllo come Git. L'automazione del flusso di lavoro permette l'esecuzione di attività ripetitive come la creazione di script di build personalizzati. In ultimo, alcune estensioni come Remote-SSH e WSL consentono agli sviluppatori di lavorare su codice che risiede su server remoti o a cui non avrebbero accesso diretto.

Le estensioni di VS Code possono sfruttare diverse categorie di capacità con l'obiettivo di estendere l'editor. Tra queste categorie sono incluse le capacità di workbench per personalizzare l'interfaccia utente, le abilità di editing per migliorare l'esperienza di modifica del codice, e le funzionalità di integrazione con servizi esterni per connettere l'editor a piattaforme e strumenti di terze parti [5].

Le estensioni di Visual Studio Code sono principalmente sviluppate utilizzando TypeScript, un linguaggio di programmazione sviluppato da Microsoft che estende JavaScript aggiungendo definizioni di tipo statico [2]. TypeScript offre vantaggi nello sviluppo di estensioni attraverso caratteristiche come la type safety, che aiuta a rilevare errori durante lo sviluppo e non solo a runtime; l'IntelliSense avanzato che migliora l'auto completamento e la navigazione del codice; il refactoring sicuro che permette di applicare modifiche al codice con maggiore affidabilità; la documentazione integrata dove i tipi dei dati servono come documentazione

vivente del codice e infine l'interoperabilità che garantisce piena compatibilità con codice JavaScript già esistente.

2.1.3 Sviluppo delle Estensioni nel contesto del progetto

Nel contesto di questo progetto, Visual Studio Code rappresenta la piattaforma di riferimento per l'implementazione dell'estensione MultiMind. L'API di estensione di VS Code si occupa di fornire gli strumenti necessari per creare interfacce utente personalizzate, gestire comandi e integrare funzionalità AI direttamente all'interno dell'ambiente di sviluppo [1].

L'architettura delle estensioni VS Code si basa su un sistema di attivazione event-driven, dove le estensioni vengono caricate solo quando necessario così da garantire prestazioni ottimali in ogni momento. Per il progetto, questo approccio permette di gestire in maniera efficiente l'integrazione di multiple API di intelligenza artificiale senza compromettere le prestazioni dell'editor.

Le estensioni possono essere indirizzate a diversi elementi: quelle che si occupano di estendere l'editor possono migliorare l'esperienza di sviluppo modificando l'evidenziazione della sintassi e fornendo auto completamento, mentre quelle legate ai comandi e menu possono integrare nuovi comandi, opzioni di menu e azioni della barra degli strumenti [6].

L'API delle estensioni VS Code offre l'accesso a funzionalità cruciali per il progetto. Alcune di queste opzioni sono la gestione dei workspace, l'interazione con il sistema di file, la creazione di webview personalizzate per interfacce utente avanzate e l'integrazione con il sistema di comandi dell'editor. Le capacità che vengono offerte attraverso le API sono fondamentali per riuscire ad implementare un sistema che permette agli sviluppatori di interagire con diversi modelli LLM all'interno dell'ambiente di sviluppo [1].

2.2 Large Language Models e Google Gemini

2.2.1 Panoramica dei Large Language Models

I Large Language Models (LLM) rappresentano un'evoluzione importante nei campi dell'intelligenza artificiale e del Natural Language Processing (NLP). Questi modelli, che sono spesso basati su architetture di deep learning, vengono progettati in maniera scrupolosa per formulare e generare testo il più simile possibile a quello umano.

Il meccanismo con cui operano i Large Language Models può essere pensato come un processo di addestramento in varie fasi [4]: si inizia con il pretraining, in cui il modello viene esposto senza supervisione a grandissime quantità di dati testuali per apprendere le rappresentazioni statistiche del linguaggio; in seguito, avviene il fine-tuning, una fase in cui il modello supervisionato viene coinvolto in compiti o domini specifici per utilizzare dataset curati e tecniche di apprendimento. Infine, nella fase di inference, il modello addestrato viene utilizzato in contesti comuni per generare testo o fare previsioni basate su query di input.

L'integrazione degli assistenti AI nei flussi di lavoro di sviluppo si sta rapidamente evolvendo, passando da attività assistite dall'automazione a interazioni per far collaborare sviluppatori e AI [6].

2.2.2 Google Gemini: Implementazione e Vantaggi

Google Gemini rappresenta l'ultima generazione di modelli linguistici sviluppati da Google. Si distingue dagli altri modelli di intelligenza artificiale grazie alla sua architettura multimodale che permette di elaborare testo, immagini, audio e video, alla sua capacità di lavorare a vari livelli di prestazione in base a quale delle versioni disponibili viene utilizzata (Nano, Pro, Ultra) o in base al contesto d'utilizzo, all'efficienza computazionale ottimizzata che consente di avere prestazioni elevate con il minor consumo possibile di risorse, e ad un'integrazione nativa progettata per inserirsi in maniera fluida nel contesto di Google [3].

Una caratteristica che rende Google Gemini particolarmente attrattivo è la disponibilità di accedere ai servizi API utilizzando chiavi di autenticazione gratuite (APIKey), aspetto che è stato sfruttato nell'implementazione di questo progetto. Google fornisce accesso a Gemini tramite Google AI Studio e le Gemini API, che offrono molteplici strumenti, come ad esempio, REST API con interfacce HTTP standardizzate per l'integrazione; SDK per piattaforme multiple con diverse librerie per i linguaggi di programmazione; supporto streaming per risposte in tempo reale e configurazione flessibile con parametri personalizzabili per temperatura, top-p, e altri iper-parametri.

Nel contesto dello sviluppo e delle estensioni per VS Code, Gemini ha ottime capacità di comprensione per l'analisi di codice in vari linguaggi, inoltre può supportare la generazione di codice per implementazioni, test e documentazione, e fornisce assistenza nel processo di debugging attraverso la rilevazione di errori e il suggerimento di correzioni, e supporta il refactoring intelligente fornendo pareri per migliorare la qualità e la struttura del codice [3].

L'integrazione di Gemini avviene attraverso una classe astratta "AIDriver" che gestisce le proprietà comuni ai diversi driver AI tra cui l'autenticazione tramite APIKey, la gestione delle versioni API e la configurazione dei modelli [4]. Questa architettura modulare, visibile nella struttura delle classi sviluppate, permette di mantenere separata la logica di comunicazione con le API dalla logica applicativa, facilitando future estensioni e modifiche.

2.2.3 Ruolo dei LLM nel Progetto

I Large Language Models assumono un ruolo centrale all'interno del progetto e vengono usati come motori di intelligenza artificiale specializzati per diverse tipologie di compiti di sviluppo. Usare una strategia multi-modello permette di sfruttare i punti di forza specifici di ciascun LLM, superando le limitazioni che possiedono comunemente i sistemi basati su un modello singolo [4].

L'architettura modulare sviluppata per il progetto dà la possibilità di integrare diversi provider di LLM, come ad esempio Google Gemini. Questo approccio multi-agente facilita la specializzazione dei compiti: alcuni

modelli possono essere ottimizzati per la generazione di codice, altri per l'analisi e il debugging, mentre altri ancora per la documentazione e i commenti.

La selezione dinamica dei modelli avviene attraverso un sistema di supervisione intelligente che valuta la natura della richiesta fatta dall'utente e assegna il compito di risolverla al modello più appropriato [4]. Questo meccanismo migliora in modo significativo la qualità delle risposte fornite agli sviluppatori, e garantisce che ogni tipo di query venga gestita dal modello con le competenze più adatte per quel tipo di problema.

2.3 Architettura della Soluzione

2.3.1 Panoramica e Componenti principali dell'Architettura

L'architettura della soluzione che si propone è progettata seguendo i principi di modularità e scalabilità per permettere un'integrazione efficace di diverse tecnologie di AI dentro l'ambiente Visual Studio Code. Il sistema si basa su una struttura formata da componenti interconnessi che rendono più semplice la comunicazione tra l'interfaccia utente, la logica di business e i servizi esterni di AI.

La progettazione e la sperimentazione di nuovi compiti assistiti dalle AI richiedono tuttavia non pochi investimenti. La realizzazione di un processo di sviluppo AI-assisted all'interno di un IDE richiede l'implementazione di meccanismi per attivare gli assistenti al momento giusto, gestire l'interazione con uno o più assistenti, processare i risultati raccolti e visualizzare il feedback da inviare agli sviluppatori [6].

L'architettura è costruita intorno a quattro componenti fondamentali, con l'obiettivo di lavorare in sinergia per fornire un'esperienza di sviluppo AI-assisted migliore possibile.

Il primo componente sono i Driver, che hanno il ruolo di interfacce standardizzate posizionate tra l'estensione e i servizi AI esterni [4]. Ogni Driver ha il compito di implementare un protocollo di comunicazione specifico per un determinato provider di intelligenza artificiale, e gestisce gli aspetti critici come l'autenticazione tramite APIKey, la serializzazione delle richieste, la gestione degli errori di rete e la normalizzazione delle risposte. I Driver servono come collegamento fondamentale tra il plugin e i vari assistenti AI o le varie API esterne, facilitando la comunicazione tra questi senza soluzione di continuità così da garantire l'interoperabilità tra diversi modelli. Questo tipo di astrazione è ciò che permette al sistema di supportare più di un provider AI allo stesso tempo e di mantenere un'interfaccia unificata verso il resto dell'applicazione. La gerarchia dei driver è estensibile grazie alla possibilità di configurare e integrare facilmente nuovi assistenti AI e modelli [6].

I Driver Manager sono il secondo livello architetturale e si occupano di organizzare le interazioni con le AI [4]. Progettato come singleton, il Driver Manager garantisce l'attivazione di un'istanza alla volta, in modo da prevenire ridondanze e ottimizzare l'allocazione delle risorse. Questo genere di componente mantiene un registro dei Driver disponibili, gestisce le tre fasi del loro ciclo di vita (inizializzazione, attivazione, disattivazione), e fornisce un livello di astrazione che separa la logica applicativa dai dettagli delle implementazioni dei singoli provider. I Driver Manager implementano anche strategie di load balancing e

failover per garantire la resistenza del sistema. Dal punto di vista tecnico, il Driver Manager fornisce due metodi principali per l'interazione con le AI: CallBack, che implementa un meccanismo asincrono per recuperare la prima risposta AI disponibile (modalità "Continue After First Response"), e FetchAll, che raccoglie tutte le risposte fornite prima di procedere (modalità "Continue After Last Response") [6].

I Task rappresentano il terzo componente e definiscono le unità di lavoro specifiche che possono essere eseguite dai modelli AI. Ogni Task incorpora la logica necessaria per processare un tipo di richiesta, come la generazione di codice, l'analisi di bug, o la valutazione di un metodo. I Task sono categorizzati in base al loro livello di specificità: Task Definiti, che seguono requisiti ben strutturati con contesti dettagliati, che sono ideali quando le risposte AI devono combaciare con vincoli rigorosi; Task Aperti, che offrono maggiore flessibilità permettendo agli utenti di fornire istruzioni ampie che guidano il processo decisionale dell'AI. I Task offrono flessibilità nell'elaborazione delle risposte e permettono agli utenti di scegliere se avere risultati immediati attraverso la modalità di esecuzione sincrona o di eseguire elaborazioni più complesse in maniera asincrona [4]. I Task Manager hanno il compito di coordinare l'esecuzione di Task multipli abilitando workflow complessi che coinvolgono la collaborazione tra diversi modelli AI. Servono come intermediari tra Task e Actions, coordinando l'esecuzione di compiti multipli e consentendo quindi l'analisi e l'integrazione dei risultati prodotti da diversi assistenti AI [6].

Le Action costituiscono l'ultimo componente principale e hanno la funzione di ponte tra l'interfaccia utente di VS Code e la logica interna dell'estensione. Le Action prendono le interazioni fatte dall'utente (comandi da palette, click su pulsanti, input da chat) e le traducono in operazioni all'interno dei Task e dei Driver sottostanti. Rappresentano gli eventi specifici che gli utenti possono attivare dall'interfaccia VS Code, e la risposta a questi tipicamente richiede l'esecuzione di un task che coinvolge uno o più AI [6]. Questo livello di astrazione garantisce che l'utilizzo dell'estensione sia sempre ottimizzata a prescindere dalla complessità delle operazioni AI eseguite.

2.3.2 Flusso di Esecuzione e Comunicazione

Il flusso di esecuzione (figura 1) segue un pattern ben definito che garantisce efficienza e una facile manutenzione. Quando un utente attiva una funzionalità AI attraverso l'interfaccia di VS Code, l'Action a cui corrisponde riceve la richiesta e la analizza per determinare il tipo di elaborazione necessaria da svolgere. Il processo inizia con l'utente che esegue un evento GUI che avvia un'Action [6].

A seconda della complessità dell'operazione che deve essere eseguita, l'Action può eseguire direttamente un Task o un Task Manager per dirigere l'esecuzione di task multipli. L'Action crea quindi uno o più Task appropriati, che vengono passati al Task Manager per la direzione del lavoro. Terminato il processo l'Action crea quindi uno o più Task appropriati, che vengono passati al Task Manager per la direzione del lavoro.

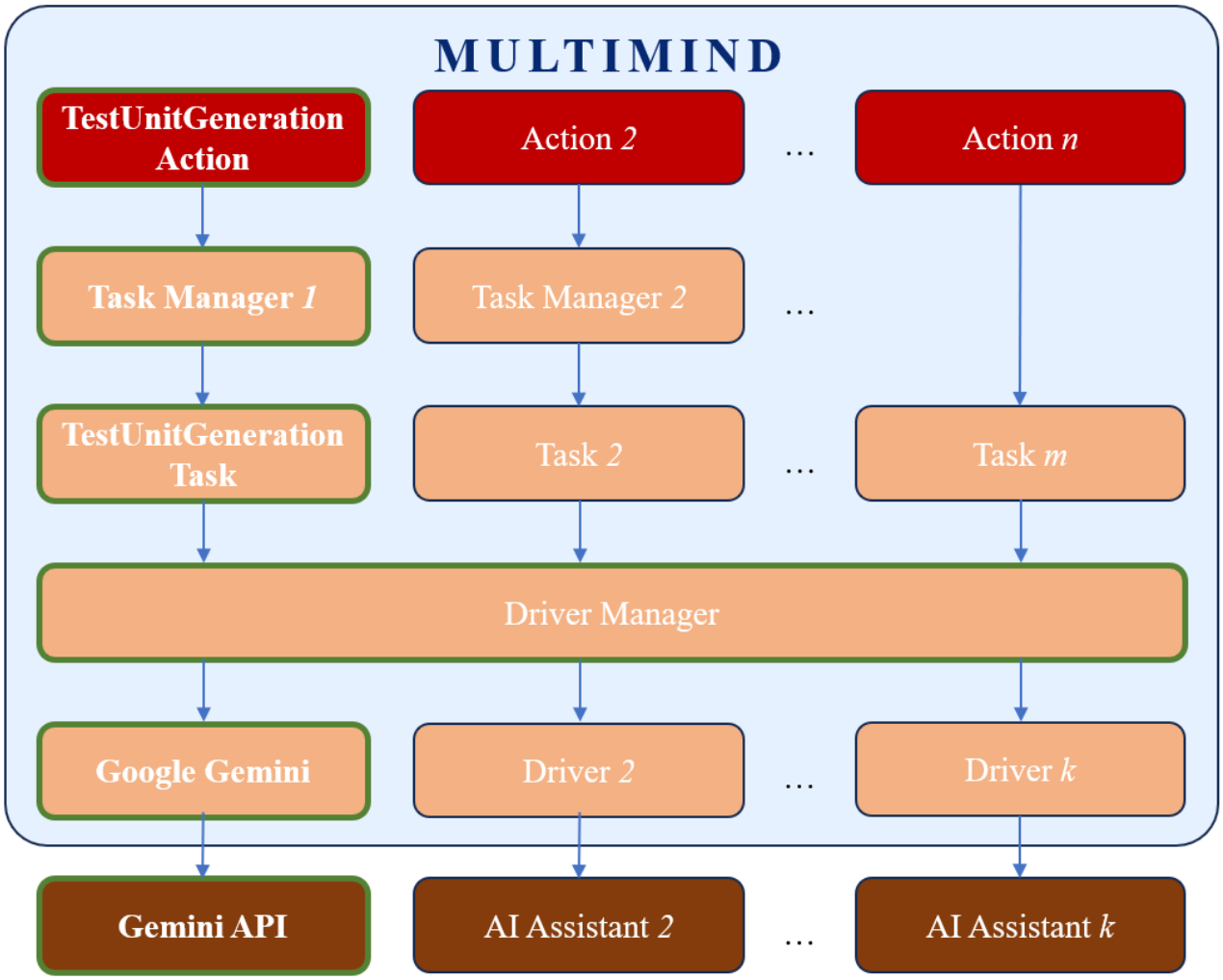

Figura 1: Architettura della soluzione e flusso d'esecuzione del lavoro

Il Task Manager seleziona i Driver più adatti per l'esecuzione basandosi su determinati criteri come il tipo di modello richiesto, la disponibilità del servizio e le preferenze dell'utente. Un Task Manager esegue i task sia in parallelo che in modo sequenziale, e ogni Task invoca assistenti AI esterni e servizi in base alle necessità [6]. I Driver, una volta selezionati, eseguono le chiamate API verso i rispettivi servizi AI e si occupano di gestire tutti gli aspetti tecnici della comunicazione.

Le risposte ricevute dai servizi AI vengono processate dai Driver per normalizzare il formato e gestire eventuali errori. L'assistente AI processa la richiesta e restituisce una risposta, che viene propagata tramite il driver al driver manager e poi al task. Una volta corrette, se necessario, vengono passate ai task per l'elaborazione specifica del dominio. I Task possono applicare trasformazioni aggiuntive, filtraggio, o combinazioni di risultati provenienti da multiple fonti AI.

Infine, i risultati elaborati vengono restituiti alle Action, che si occupano di presentarli all'utente utilizzando l'interfaccia di Visual Studio Code attraverso notifiche, pannelli dedicati, modifiche dirette al codice nell'editor o nuovi file.

Questa architettura modulare garantisce che ogni componente abbia delle responsabilità ben definite e permette di aggiungere ulteriori estensioni del sistema senza avere impatti significativi sui componenti già esistenti. La separazione delle responsabilità facilita anche il testing e il debugging, contribuendo alla robustezza complessiva della soluzione.

Capitolo 3: Test Unit Generation: Implementazione

3.1 Architettura del Sistema

Il plugin per la generazione tramite intelligenze artificiali dei test unitari è strutturato seguendo il pattern Action-Task che divide in maniera chiara e semplice la logica di comunicazione con l'API da quella utente. L'architettura così strutturata garantisce una separazione netta delle responsabilità: la classe Action gestisce l'interfaccia utente e l'interazione con VS Code, mentre la classe Task implementa la logica applicativa e l'integrazione con i servizi AI. Il file Extension ha il ruolo di coordinatore per l'attivazione e la registrazione di tutti i componenti del sistema.

3.2 Analisi di testUnitGeneration.action

La classe TestUnitGenerationAction rappresenta il cuore dell'interfaccia utente per la funzionalità di Test Unit AI-generated. Questa classe si occupa di gestire tutti i passaggi in cui l'utente interagisce con l'ambiente VS Code, partendo dalla registrazione dei comandi fino alla presentazione dei risultati finali.

Il costruttore della classe prende il contesto dell'estensione VS Code e procede in modo immediato alla registrazione dei comandi attraverso il metodo registerCommands(). Questo approccio permette che tutti i comandi presenti siano subito disponibili per essere utilizzati dall'utente, sin dal momento in cui la classe viene istanziata.

Il metodo registerCommands() si occupa di registrare due comandi distinti di VS Code. Il primo comando, multimind.generateTestUnit, è dedicato alla generazione di test a partire da una selezione di codice, mentre il secondo comando, multimind.generateTestFromFile, permette di generare test per un intero file. Entrambi i comandi vengono registrati utilizzando vscode.commands.registerCommand per poi essere aggiunti nel contesto e garantire una corretta gestione del loro ciclo.

I metodi generateTestFromSelection() e generateTestFromFile() hanno la funzione di entry point per le due modalità operative. Seguono entrambi un pattern identico: verificano la presenza di un editor attivo, ne validano il contenuto (selezione o file completo), e lasciano l'elaborazione di tale contenuto al metodo processTestGeneration(). Le differenze tra i due pattern sono nel tipo di contenuto processato e nel flag che viene passato per indicare se si tratta di una selezione o di un file completo.

3.2.1 Il Cuore del Processo: processTestGeneration()

Il metodo processTestGeneration() (figure 2, 3 e 4) rappresenta il fulcro dell'intera funzionalità. Questo metodo comanda l'intero flusso di generazione dei test unitari attraverso una serie di passaggi ben definiti.

```
private async processTestGeneration(sourceCode: string, editor: vscode.TextEditor, isSelection: boolean): Promise<void> {
    // Chiedi all'utente il nome del file di test
    const testFileName = await vscode.window.showInputBox({
        prompt: 'Inserisci il nome per il file di test (senza estensione)',
        placeHolder: isSelection ? 'es: loginMethod' : 'es: userService',
        validateInput: (value: string) => {
            if (!value || !value.trim()) {
                return 'Il nome del file non può essere vuoto';
            }
            // Verifica caratteri validi per nome file
            if (!/^[a-zA-Z0-9_-]+$/.test(value.trim())) {
                return 'Il nome può contenere solo lettere, numeri, underscore e trattini';
            }
            return null;
        }
    });

    // Se l'utente cancella o non inserisce nulla, interrompi
    if (!testFileName) {
        vscode.window.showWarningMessage('Generazione test annullata.');
        return;
    }
```

Figura 2: prima parte del codice del metodo processTestGeneration() in testUnitGeneration.action.ts

Il processo inizia con la richiesta all'utente di dare un nome al file che conterrà il test unitario generato. Questa interazione avviene tramite vscode.window.showInputBox(), che ha un dialog con validazione integrata. La verifica controlla che il campo del nome non sia vuoto e che contenga solo caratteri in regola per nominare un file (lettere, numeri, underscore e trattini). Il placeholder del dialog cambia in maniera dinamica in base al fatto che si stia processando un testo selezionato (es: loginMethod) o un file completo (es: userService), fornendo un'indicazione contestuale all'utente.

Una volta nominato il file, il metodo calcola il percorso di import utilizzando calculateRelativeImportPath(). Questo step è fondamentale perché determina in che modo il file andrà ad importare il codice sorgente che bisogna testare.

```
await vscode.window.withProgress({
    location: vscode.ProgressLocation.Notification,
    title: "Generazione unit test in corso...",
    cancellable: false
}, async (progress) => {
    try {
        progress.report({ increment: 0, message: "Preparazione richiesta..." });

        // Calcola il percorso relativo che sarà usato nel test
        const relativePath = this.calculateRelativeImportPath(editor.document.fileName, testFileName.trim(), editor.document.languageId);

        // Prepara la richiesta AI con informazioni sul percorso
        const aiRequest: AIRequest = {
            prompt: `${sourceCode}

        IMPORT_PATH: ${relativePath}`,
            language: editor.document.languageId,
            temperature: 0.3,
            top_p: 0.9
        };

        progress.report({ increment: 25, message: "Invio richiesta all'IA..." });
```

Figura 3: seconda parte del codice del metodo processTestGeneration() in testUnitGeneration.action.ts

Il metodo costruisce quindi un oggetto AIRequest che contiene il codice sorgente e include il percorso di import calcolato come parte del prompt in modo non convenzionale. Questa informazione aggiuntiva viene formattata come IMPORT_PATH: ${relativePath} e viene accodata al codice sorgente, così da fornire al sistema AI tutte le informazioni necessarie per generare import corretti.

L'intero processo è avvolto in un vscode.window.withProgress() che fornisce feedback visivo all'utente attraverso una barra di progresso. Questo elemento è importante perché la generazione AI può richiedere svariati secondi, e l'utente deve sapere se l'operazione è in corso.

```
        // Genera il test
        const response = await TestUnitGeneration.getFirstResponse(aiRequest);

        if (!response.response) {
            throw new Error('Nessuna risposta ricevuta dall\'IA');
        }

        progress.report({ increment: 75, message: "Creazione file di test..." });

        // Salva il file di test con il nome scelto dall'utente
        await this.saveTestFile(editor.document.fileName, response.response, editor.document.languageId, testFileName.trim());

        progress.report({ increment: 100, message: "Completato!" });

        vscode.window.showInformationMessage('Unit test generato con successo!');

    } catch (error) {
        console.error('Error in processTestGeneration:', error);
        throw error;
    }
```

Figura 4: terza parte del codice del metodo processTestGeneration() in testUnitGeneration.action.ts

3.2.2 Gestione dei Percorsi: calculateRelativeImportPath()

Il metodo calculateRelativeImportPath(), (figura 5) implementa una logica per gestire le convenzioni di import specifiche per ogni linguaggio di programmazione.

Il metodo inizia calcolando la struttura delle directory: determina in che punto si trova il file sorgente, verifica se esista già una directory adatta (definita da un nominato trytest) e nel caso non ci fosse la crea e determina il percorso relativo tra questi due punti. A seguito di queste prime azioni, utilizza le API Node.js path.dirname(), path.join(), e path.relative() per gestire in modo corretto i percorsi su diversi sistemi operativi.

```
//Calcola il percorso relativo di import dal file di test al file sorgente
private calculateRelativeImportPath(sourceFilePath: string, testFileName: string, language: string): string {
    const sourceDir = path.dirname(sourceFilePath);
    const testDir = path.join(sourceDir, 'try_tests');
    const sourceBaseName = path.basename(sourceFilePath, path.extname(sourceFilePath));

    // Calcola il percorso relativo dal file di test al file sorgente
    const relativePath = path.relative(testDir, sourceFilePath);

    // Rimuovi l'estensione e normalizza per l'import
    const importPath = relativePath.replace(path.extname(relativePath), '');

    // Gestisci diversi formati di import per linguaggio
    switch (language) {
        case 'javascript':
        case 'typescript':
            // Per JS/TS, usa path Unix-style e aggiungi ./ se necessario
            const unixPath = importPath.replace(/\\/g, '/');
            return unixPath.startsWith('../') ? unixPath : `./${unixPath}`;

        case 'python':
            // Per Python, converti path in moduli (sostituisci / con .)
            return importPath.replace(/[/\\]/g, '.');

        case 'java':
            // Per Java, usa il nome della classe
            return sourceBaseName;

        default:
            // Default: ritorna il path relativo normalizzato
            return importPath.replace(/\\/g, '/');
    }
}
```

Figura 5: codice del metodo calculateRelativeImportPath() in testUnitGeneration.action.ts

La parte più interessante del metodo è lo switch statement che gestisce i diversi import in base al linguaggio. Ad esempio, per JavaScript e TypeScript, il metodo converte i percorsi in stile Unix e aggiunge il prefisso ./ quando necessario. Per Python, trasforma i separatori di percorso in punti così da rispettare la convenzione dei moduli Python. Per Java, utilizza solo il nome base del file, mentre per altri linguaggi applica una normalizzazione generica. I metodi saveTestFile() e generateTestFilePath() completano la gestione dei file.

Il primo si occupa della creazione fisica del file, della creazione della directory, se necessaria, e dell'apertura automatica del file generato nell'editor. Il secondo invece implementa le regole standard di naming specifiche in base al linguaggio rilevato (.test.js per JavaScript, test_.py per Python, Test.java per Java, etc.) (tabella 1).

Nella tabella sottostante (tabella 1) sono contenute le triple che rappresentano i linguaggi e i relativi framework ed estensioni di testo attualmente gestiti dal metodo generateTestFilePath().

Linguaggio	**Framework**	**Estensione File Test**
JavaScript	Jest	.test.js
TypeScript	Jest	.test.ts
Python	pytest	test_.py
Java	JUnit 5	Test.java
C#	NUnit	Tests.cs
Go	testing package	_test.go
PHP	PHPUnit	Test.php

Tabella 1: triple Linguaggio-Framework-Estensione presenti nel metodo generateTestFilePath()

3.3 Analisi di testUnitGeneration.task

La classe TestUnitGeneration va ad estendere la classe base Task e implementa tutta la logica necessaria che permette la generazione effettiva dei test unitari. Questa classe è progettata per essere stateless e utilizza solo metodi statici in modo da seguire un pattern funzionale che rende più semplice il testing e la manutenzione del codice stesso.

La classe mantiene una mappatura costante TESTING_FRAMEWORKS che associa ogni linguaggio di programmazione al suo framework di testing più comune. Questa mappatura è utilizzata per personalizzare i prompt inviati al sistema AI e garantisce che i test generati utilizzino le convenzioni e le librerie appropriate di ogni linguaggio.

3.3.1 Il Metodo chiave: activate()

Il metodo activate() (figure 6 e 7), rappresenta il punto di partenza per la generazione dei test unitari.

Il metodo inizia con la configurazione del prompt di sistema per ottimizzare la risposta dell'AI. Il messaggio "*You are an expert coder. Generate comprehensive unit tests following best practices*" è stato scelto dopo diversi test per garantire la miglior qualità possibile dei test generati.

```
public static async activate(data: AIRequest): Promise<AIResponse[]> {
    // Configura il sistema per generazione test
    data.system = 'You are an expert coder. Generate comprehensive unit tests following best practices.';

    // Estrai percorso di import e pulisci il codice sorgente
    const { importPath, sourceCode } = this.extractImportPath(data.prompt);

    // Costruisce il prompt specifico per test unit
    data.prompt = this.buildTestPrompt(sourceCode, importPath, data.language);

    // Parametri ottimizzati per generazione codice
    data.temperature ??= 0.3;
    data.top_p ??= 0.9;
```

Figura 6: prima parte del codice del metodo activate() in testUnitGeneration.task.ts

Il processo continua con l'estrazione del percorso di import dal prompt ricevuto. Questo avviene tramite il metodo extractImportPath(), che separa il codice sorgente dalle informazioni di contesto (come IMPORT_PATH: ./src/utils/helper) che erano state aggiunte dalla classe Action. Questa separazione è necessaria perché il codice sorgente, per essere processato, deve essere isolato, mentre le informazioni di contesto vengono utilizzate nel loro complesso per la costruzione del prompt finale.

Il metodo buildTestPrompt() si occupa di creare un prompt ottimizzato che includa le istruzioni specifiche per la generazione di test comprensibili, il percorso di import esatto che bisogna utilizzare e le informazioni sul framework di testing appropriato per il linguaggio utilizzato. Per TypeScript, ad esempio, il prompt include specifiche per Jest con @types/jest e configurazioni per describe, it, e expect.

```
try {
    const aiProviders = ['google-gemini'];
    const responses = await Task.getAiAllResponses(data, aiProviders, 'request');

    console.log('Test unit generation responses:', responses);

    // Processa le risposte per estrarre solo il codice
    return responses.map(response => ({
        ...response,
        response: response.response ?
            this.processGeneratedTest(
                this.extractCodeFromMarkdown(response.response, data.language ? [data.language] : undefined) || response.response,
                data.language
            ) : response.response
    }));

} catch (error) {
    console.error('Error in test unit generation:', error);
    throw new Error(`Failed to generate unit tests: ${error}`);
}
```

Figura 7: seconda parte del codice del metodo activate() in testUnitGeneration.task.ts

L'invocazione dell'AI viene eseguita tramite Task.getAiAllResponses() e si utilizza il provider Google Gemini, di cui si è trattato nel secondo capitolo. I valori parametri temperature e top_p sono stati selezionati, tra quelli possibili, in modo da ottimizzare la generazione dei test unitari dopo vari tentativi e controlli (sono stati scelti i valori più bassi per avere un maggior determinismo).

Alla fine del ciclo, le risposte vengono processate in modo da estrarre solo il codice rilevante. Il metodo extractCodeFromMarkdown() si occupa di rimuovere i blocchi markdown che spesso circondano il codice generato dall'AI, mentre processGeneratedTest() effettua delle validazioni aggiuntive utilizzando specifiche per il linguaggio usato.

3.3.2 I Metodi di Supporto

Il metodo activate() è complementare ad altri tre metodi definiti di supporto.

Il metodo extractImportPath() (figura 8) utilizza espressioni per identificare e separare le informazioni di import dal codice sorgente. Questo approccio è sicuro e gestisce in modo corretto i casi limite dove il prompt potrebbe contenere molteplici righe di metadati.

```
//Estrae il percorso di import dal prompt e ritorna il codice pulito
private static extractImportPath(prompt: string): { importPath: string; sourceCode: string } {
    const lines = prompt.split('\n');
    const importPathLine = lines.find(line => line.startsWith('IMPORT_PATH: '));

    return {
        importPath: importPathLine?.replace('IMPORT_PATH: ', '') || '',
        sourceCode: importPathLine ?
            lines.filter(line => !line.startsWith('IMPORT_PATH: ')).join('\n') :
            prompt
    };
}
```

Figura 8: codice del metodo extractImportPath() in testUnitGeneration.task.ts

Il metodo buildTestPrompt() (figura 9) implementa una logica complessa per la costruzione del prompt finale. Oltre alle istruzioni base, si occupa di aggiungere informazioni specifiche sul framework di testing usando il supporto della mappatura TESTING_FRAMEWORKS.

```
    //Costruisce il prompt per la generazione dei test
    private static buildTestPrompt(sourceCode: string, importPath: string, language?: string): string {
        let prompt = `Generate comprehensive unit tests for the following code:

${sourceCode}

${importPath ? `Use this exact import path: ${importPath}` : ''}

Generate ONLY the test code with proper imports and setup.
Include all necessary imports for the testing framework and use comprehensive test cases with proper TypeScript/JavaScript syntax.`;

        // Aggiungi framework specifico se il linguaggio è supportato
        if (language && language in this.TESTING_FRAMEWORKS) {
            const framework = this.TESTING_FRAMEWORKS[language as keyof typeof this.TESTING_FRAMEWORKS];
            prompt += `\n\nTarget language: ${language}. Use ${framework} testing framework with proper imports and configuration.
            For TypeScript, assume @types/jest is available and ensure proper Jest setup with describe, it, and expect functions.`;
        }

        return prompt;
    }
```

Figura 9: codice del metodo buildTestPrompt() in testUnitGeneration.task.ts

Il metodo processGeneratedTest() (figura 10) effettua il post-processing specifico per linguaggio del codice. Per TypeScript e JavaScript; ad esempio, verifica che il codice generato includa correttamente le funzioni Jest (describe, it, expect) senza aggiungere altri import non necessari, dato che il prompt è stato costruito per garantire che l'AI abbia già generato quelli necessari.

```
//Processa il test generato per assicurarsi che abbia gli import necessari
private static processGeneratedTest(testCode: string, language?: string): string {
    if (!testCode || !language) {return testCode;}

    // Per TypeScript/JavaScript, verifica che ci siano gli import Jest necessari
    if ((language === 'typescript' || language === 'javascript') && testCode.includes('describe(')) {
        // Se il codice usa Jest ma non ha gli import necessari, mantieni il codice originale
        // L'AI dovrebbe già aver generato gli import corretti grazie al prompt migliorato
        return testCode;
    }

    return testCode;
}
```

Figura 10: codice del metodo processGeneratedTest() in testUnitGeneration.task.ts

3.4 Analisi di Extension

Il file extension.ts si occupa di attivare e coordinare tutti i componenti del sistema. La funzione activate() viene chiamata in maniera automatica da Visual Studio Code quando l'estensione viene caricata per la prima volta al suo interno.

Dentro questa funzione, vengono registrati tutti i diversi componenti dell'estensione, inclusi quelli già esistenti (TestDisposable, CommentDisposable, CodeReviewAction, etc.) insieme al nuovo metodo progettato TestUnitGenerationAction (figura 11).

TestUnitGenerationAction viene istanziato con il passaggio del contesto dell'estensione, che contiene tutte le informazioni necessarie per poterlo integrare con VS Code. I componenti vengono aggiunti a context.subscriptions, così da garantire che Visual Studio Code possa gestire in modo corretto tutto il ciclo di vita dell'estensione, inclusa la pulizia delle risorse nel caso in cui l'estensione venga disattivata o eliminata.

```
export function activate(context: vscode.ExtensionContext) {

    // Use the console to output diagnostic information (console.log) and errors (console.error)
    // This line of code will only be executed once when your extension is activated
    console.log('Congratulations, your extension "multimind" is now active!');

    // The command has been defined in the package.json file
    // Now provide the implementation of the command with registerCommand
    // The commandId parameter must match the command field in package.json
    const disposable = vscode.commands.registerCommand('multimind.helloWorld', () => {
        // The code you place here will be executed every time your command is executed
        // Display a message box to the user
        vscode.window.showInformationMessage('Hello World from multimind!');
    });

    const codeReviewAction = new CodeReviewAction();

    context.subscriptions.push(
        disposable,
        TestDisposable,
        CommentDisposable,
        codeReviewAction,
        vscode.window.registerWebviewViewProvider(
            ChatViewProvider.viewType,
            new ChatViewProvider(context)
        ),
        vscode.commands.registerCommand('multimind.openChat', () => {
            ChatPanel.createOrShow();
        })
    );
    const testUnitGenerationAction = new TestUnitGenerationAction(context);
    context.subscriptions.push(testUnitGenerationAction);
}
```

Figura 11: il metodo activate() registra tutti i plugin disponibili in MultiMind

3.5 Gestione degli Errori e Sicurezza del codice

Tutto il sistema implementa una strategia di error handling basata su più livelli così da garantire la massima sicurezza e affidabilità dell'estensione. A livello di validazione degli input, vengono controllati la presenza di editor attivi, la validità delle selezioni, e la correttezza dei nomi dei file. A livello di operazioni I/O, ogni chiamata a file system o API esterne è rinforzata con try-catch e con messaggi di errore user-friendly. A livello di integrazione AI si ha una gestione dei timeout, degli errori di rete, e delle risposte non fornite nel modo corretto.

Grazie al mantenimento stabile del sistema, questa architettura multilivello garantisce all'utente l'invio di feedback continui e appropriati, anche nel caso in cui si rilevasse la presenza di eccezioni o di input inaspettati.

Capitolo 4: Generate Test Unit: Manuale Utente

4.1 Panoramica e Installazione

Generate Test Unit è un'estensione di MultiMind sviluppata per effettuare la generazione automatica di test unitari utilizzando dei modelli AI. La creazione dei test avviene usando il codice scritto e vengono seguite le best practices del framework di testing del progetto.

4.1.1 Quick Start e prerequisiti

I prerequisiti per poter utilizzare l'estensione sono:

- Visual Studio Code (versione 1.74.0 o superiore): L'IDE principale dove l'estensione verrà installata e utilizzata.
- Node.js (versione 16.x o superiore): Necessario per progetti JavaScript/TypeScript.
- Progetto attivo: Serve avere un file di codice aperto nell'editor di VS Code

bash

 # 1. installa il framework di testing (esempio TypeScript)

npm install --save-dev jest @types/jest

 # 2. scarica e installa l'estensione .vsix in VS Code

 # 3. Selezione del codice → Tasto destro → "Generate Test Unit"

 # 4. Dai un nome al file di test → Attendi la generazione

4.2 Installazione ed Utilizzo

4.2.1 Configurazione del Progetto

Se il framework di testing non dovesse essere già presente nella directory del progetto è possibile installarlo usando il comando d'installazione corretto in base al linguaggio (tabella 2) nel prompt dei comandi:

Linguaggio	**Framework**	**Comando d'installazione**
JavaScript/TypeScript	Jest	npm install --save-dev jest @types/jest
Python	Pytest	pip install pytest
Java	JUnit/TestNG	Aggiungere dipendenza Maven/Gradle
C#	NUnit/xUnit	dotnet add package NUnit
Go	Testing	Incluso in Go standard Library
PHP	PHPUnit	composer require --dev phpunit/phpunit

Tabella 2: Tabella che indica la stringa per l'installazione del framework specifico di ogni linguaggio

Attualmente rileva solo alcuni linguaggi, ma solo tre i sono quelli che hanno supporto tecnico completo: JavaScript/TypeScript, Python e Java.
Gli altri linguaggi invece hanno solo un supporto di base per la generazione dei test unitari.

Per scaricare l'estensione invece ci sono due possibilità:

Da file VSIX:

1. Scaricare il file .vsix dalla repository
2. In VS Code: Ctrl+Shift+P → "Extensions: Install from VSIX
3. Seleziona il file .vsix scaricato
4. Riavviare VS Code se necessario

Tramite VS Code Marketplace: Cerca "MultiMind" nel marketplace delle estensioni.

4.2.2 Generazione Test Unit da Selezione

La generazione delle Test Unit tramite selezione di una specifica classe, funzione o metodo avviene seguendo questi passaggi in ordine:

1. Aprire il file sorgente nel progetto VS Code.
2. Selezionare la porzione di codice di cui si vuole generare il test (funzione, classe, metodo)
3. Utilizzare il comando Generate Test Unit in uno dei due modi possibili:
    a. Aprire la Command Palette (Ctrl+Shift+P su Windows/Linux, Cmd+Shift+P su Mac), digitare e selezionare multimind.generateTestUnit
    b. Premere tasto destro e selezionare l'opzione Generate Test Unit (figura 12)

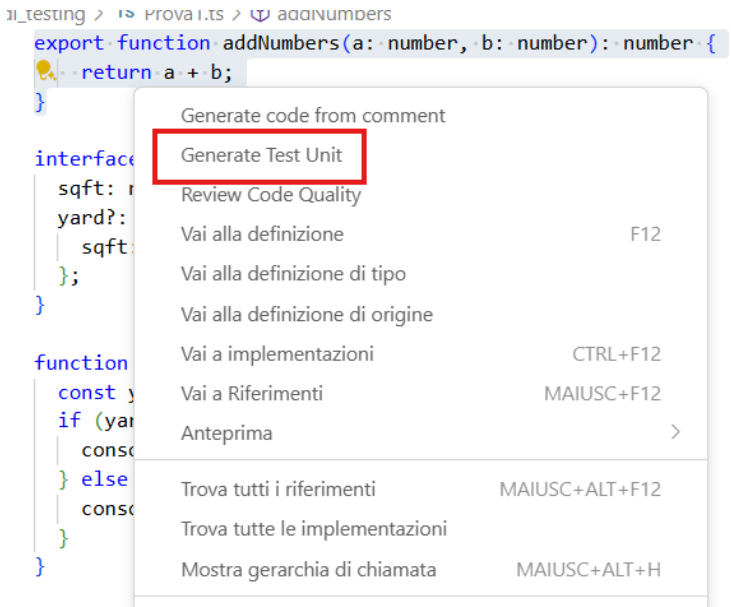

Figura 12: L'opzione Generate Test Unit che appare quando viene rilevato del testo evidenziato premendo il tasto destro

4. Inserire il nome del file di test quando richiesto (es: loginMethod) usando solo i caratteri validi (lettere, numeri, underscore e trattini) senza includere l'estensione del file.
5. Attendere che l'AI generi la test unit (la durata del processo sarà visibile tramite una barra a schermo) (figura 13).

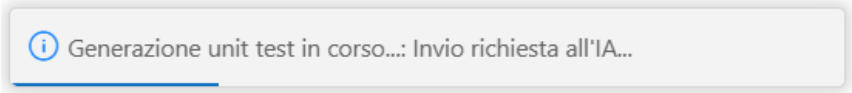

Figura 13: barra a schermo che indica quando manca al completamento della scrittura della test unit

6. Il file di test verrà automaticamente posizionato nella cartella di riferimento e sarà aperto in una nuova finestra accanto al codice originale.

4.2.3 Generazione Test da File Completo

La generazione delle Test Unit di un file completo avviene seguendo questi passaggi in ordine:

1. Aprire il file sorgente completo nel proprio progetto VS Code
2. Aprire la Command Palette (Ctrl+Shift+P su Windows/Linux, Cmd+Shift+P su Mac)
3. Digitare e selezionare: multimind.generateTestFromFile
4. Inserire il nome del file di test quando richiesto (es: userService) usando solo caratteri validi (lettere, numeri, underscore e trattini) senza includere l'estensione del file
5. Attendere che l'AI generi la test unit per il file intero (la durata del processo sarà visibile tramite una barra a schermo) (figura 13)
6. Il file di test verrà automaticamente posizionato nella cartella di riferimento e sarà aperto in una nuova finestra accanto al codice originale

4.3 Supporto tecnico e Funzionalità automatiche

4.3.1 Problemi riscontrabili e risoluzioni

In questo paragrafo si espongono alcuni dei problemi che si potrebbero riscontrare durante l'utilizzo dell'estensione, e cosa bisogna fare per risolverli (tabella 3):

Messaggio d'errore	**Causa**	**Soluzione**
"Nessun editor attivo"	Nessun file aperto in VS Code	Assicurarsi di avere un file aperto nell'editor VS Code.
"Seleziona del codice per generare la test unit"	Nessun codice selezionato	Evidenziare il codice da testare prima di eseguire il comando
"Il nome può contenere solo lettere, numeri, underscore e trattini"	Caratteri non validi nel nome del file	usare solo caratteri validi (evita spazi e caratteri speciali).
"Nessuna risposta ricevuta dall'AI"	Problemi di connessione	Verificare la connessione alla rete e riprovare

Tabella 3: Tabella che indica alcuni dei possibili messaggi e cause d'error, con soluzioni annesse

Inoltre, per usufruire in maniera ottimale dell'estensione è consigliato strutturare il proprio codice in maniera organizzata e commentata per avere una generazione dei test il più perfetta possibile. È anche consigliato di rivedere manualmente i test generati per poterli personalizzare in base alle proprie esigenze

4.3.2 Gestione automatica delle directory

L'estensione include funzionalità intelligenti per la gestione delle directory di test (figura 14):

```
    // Strategia directory: crea try_tests se non esiste già una directory di test
    const testDirNames = ['try_tests', 'trytests', 'trytest'];
    let testDir = sourceDir;

    // Cerca se esiste già una directory di test
    for (const dirName of testDirNames) {
        const possibleTestDir = path.join(sourceDir, dirName);
        try {
            // Se la directory esiste, usala
            testDir = possibleTestDir;
            break;
        } catch {
            // Directory non esiste, continua
        }
    }

    // Se non esiste nessuna directory di test, crea try_tests
    if (testDir === sourceDir) {
        testDir = path.join(sourceDir, 'try_tests');
    }

    return path.join(testDir, testFileName);
}
```

Figura 14: codice del metodo che cerca se esiste già una cartella in base al nome

- Rilevamento automatico: L'estensione cerca in maniera automatica le cartelle di test esistenti nel progetto
- Creazione automatica: Se non viene trovata una cartella appropriata, l'estensione creerà automaticamente una directory try_tests/ nella radice del progetto (figura 15)
- Gestione degli import: Gli import necessari vengono automaticamente configurati basandosi sul percorso relativo del file sorgente

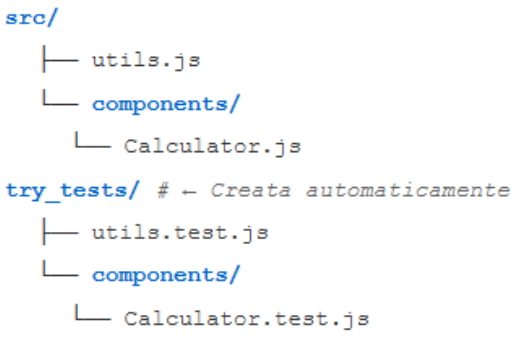

Figura 15: Struttura della repository dei file con generazione automatica

4.3.3 Note tecniche

L'estensione si basa su intelligenze artificiali avanzate per analizzare il codice sorgente e generare test unitari appropriati in base al contesto. Il sistema è progettato per comprendere la struttura e la logica del codice, così da produrre test che seguono automaticamente le best practices del framework di testing utilizzato nel progetto.

La compatibilità multipiattaforma viene garantita attraverso il supporto nativo di Visual Studio Code, rendendo l'estensione utilizzabile su tutti i principali sistemi operativi, come Windows, macOS o Linux, senza necessità di configurazioni aggiuntive specifiche per il sistema.

4.4 Riproducibilità e disponibilità del codice

Il codice sviluppato per questo studio è disponibile su GitLab per garantire la riproducibilità dei risultati e favorire ulteriori sviluppi a questo link: https://gitlab.com/llm-se/llm-code-plugin-multimind/-/tree/Tommaso_Rezzani-dev?ref_type=heads

Capitolo 5: Conclusioni

Il panorama attuale degli assistenti AI per lo sviluppo software, anche se rappresenta un passo avanti significativo nell'evoluzione della programmazione, presenta ancora delle limitazioni che non permettono di sfruttare il pieno potenziale che hanno. L'analisi condotta ha portato alla luce come strumenti consolidati, tra cui GitHub Copilot, ChatGPT, Tabnine e Amazon CodeWhisperer, abbiano delle problematiche strutturali che limitano la loro efficacia nonostante offrano un valore aggiunto indiscutibile.

La dipendenza da singoli modelli AI, l'approccio unilaterale nella proposta delle soluzioni e la scarsa integrazione con il contesto reale di sviluppo sono alcuni degli ostacoli più significativi che rallentano il flusso di lavoro e riducono l'autonomia degli sviluppatori. È proprio in questo scenario che è stata sviluppata la necessità di un approccio più flessibile e cooperativo, in grado di superare gli strumenti esistenti.

MultiMind viene creato come risposta concreta a queste esigenze, introducendo un paradigma rivoluzionario basato sull'utilizzo coordinato di diverse intelligenze artificiali. Tuttavia, nell'analisi delle funzionalità attualmente implementate all'interno del framework, è stata rilevata una funzionalità che guadagnerebbe molto in caso venisse assistita: l'assenza totale di un supporto per la generazione automatica delle Test Unit. Questa mancanza ha una enorme rilevanza se si considera che il testing rappresenta un passaggio indispensabile nel processo di sviluppo, e la creazione manuale di test unitari costituisce un'attività ripetitiva che assorbe tempo prezioso, spesso eseguita senza la necessaria attenzione a causa della sua monotonia.

La soluzione proposta in questo progetto si inserisce nell'architettura di MultiMind, sfruttando appieno le potenzialità del framework per affrontare il processo della generazione automatica di test unitari. Il sistema sviluppato non si limita a una semplice automazione del processo, ma implementa un approccio intelligente che permette agli assistenti AI di comprendere il contesto del codice da testare e generare automaticamente test appropriati, così da liberare l'utente da compiti ripetitivi e permettergli di concentrarsi sugli aspetti più creativi e strategici del proprio lavoro.

Il sistema implementato rappresenta soluzione tecnica efficace e valida; inoltre, fornisce anche un esempio concreto della facilità con cui è possibile estendere l'architettura modulare di MultiMind in modo da poter sfruttare al meglio le intelligenze artificiali in altri compiti durante le fasi della programmazione. L'integrazione nativa con Visual Studio Code rende facile e intuitiva l'utilizzo della funzione grazie all'interfaccia utente, mentre la capacità di potersi adattare in modo dinamico a diversi linguaggi di programmazione e ai rispettivi framework di testing assicura che i test generati si inseriscano perfettamente all'intero del progetto.

Nonostante i risultati positivi ottenuti durante lo sviluppo, l'implementazione attuale lascia comunque lo spazio per ideare dei possibili sviluppi futuri o miglioramenti dei sistemi presenti.

Una delle possibili aree di sviluppo riguarda la gestione delle repository in cui vengono posizionati i test. Attualmente, il sistema implementa un meccanismo di rilevamento automatico delle cartelle di test esistenti, che tuttavia potrebbe non essere sempre ottimale per tutti i progetti. Dare la possibilità all'utilizzatore di scegliere manualmente la repository di destinazione, così da evitare il rilevamento automatico in caso fosse necessario, rappresenterebbe un significativo incremento della flessibilità del sistema. Questa miglioria risulterebbe utile in progetti con strutture di directory complesse o non convenzionali, dove la logica automatica potrebbe far fatica a identificare correttamente la destinazione più appropriata per i file di test generati.

Un altro perfezionamento possibile è l'ampliamento del supporto linguistico. Anche se l'implementazione attuale fornisce un supporto tecnico completo per JavaScript, TypeScript, Python e Java, e un supporto basico per altri linguaggi, l'espansione della copertura linguistica costituirebbe un valore aggiunto da non poco.

Un'aggiunta interessante invece può essere quella di implementare una validity chain con multiple AI. L'approccio attuale per creare le Test Unit utilizza un singolo modello AI per la generazione, e di conseguenza la risposta ottenuta si assume come valida anche se non è quella ottimale. Le potenzialità dell'architettura MultiMind permetterebbero di implementare un sistema più sofisticato dove verrebbero sfruttate diverse intelligenze artificiali in parallelo per generare più test unit del medesimo caso. Successivamente, un meccanismo di validazione a posteriori analizzerebbe le proposte ottenute per identificare la soluzione più appropriata o combinare gli elementi migliori di ciascuna risposta. Questo approccio multi-agente non solo migliorerebbe la qualità dei test generati, ma rappresenterebbe anche un'evoluzione naturale del paradigma di cooperazione introdotto da MultiMind, dimostrando in maniera concreta come la collaborazione tra diverse AI possa produrre risultati superiori rispetto all'utilizzo di singoli modelli.

Questi potenziali sviluppi futuri non rappresentano attuali debolezze strutturali del sistema o mancanze, ma piuttosto indicano delle opportunità di crescita formando un percorso delineato per il futuro dell'estensione. L'architettura modulare sviluppata fornisce delle basi solide necessarie per implementare questi o altri miglioramenti senza preoccuparsi di modificare e rompere il core del sistema.

Bibliografia e Sitografia

[1] https://code.visualstudio.com - Documentazione ufficiale di Visual Studio Code

[2] https://www.w3schools.com/typescript/typescript_intro.php - Introduzione a TypeScript

[3] https://aistudio.google.com/welcome - Google AI Studio per l'accesso alle API Gemini

[4] https://code.visualstudio.com/api - API di estensione di Visual Studio Code

[5] https://code.visualstudio.com/api/extension-capabilities/overview - Panoramica delle capacità delle estensioni VS Code

[6] https://www.researchgate.net/publication/392716369_MultiMind_A_Plug-in_for_the_Implementation_of_Development_Tasks_Aided_by_AI_Assistants - Donato, B., Mariani, L., Micucci, D., Riganelli, O., & Somaschini, M. (2025). MultiMind: A Plug-in for the Implementation of Development Tasks Aided by AI Assistants. In Companion Proceedings of the 33rd ACM Symposium on the Foundations of Software Engineering (FSE '25), June 23–27, 2025, Trondheim, Norway.

[7] https://www.researchgate.net/publication/370213526_Evaluating_the_Code_Quality_of_AI-Assisted_Code_Generation_Tools_An_Empirical_Study_on_GitHub_Copilot_Amazon_CodeWhisperer_and_ChatGPT - Burak Yetiştiren ·Işık Özsoy ·Miray · Ayerdem ·Eray Tüzün: Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT, 21 April 2023.

[8] https://code.visualstudio.com/docs/copilot/overview - Documentazione ufficiale di Copilot

[9] https://platform.openai.com/docs/guides/text?api-mode=responses – Documentazione ufficiale ChatGPT

[10] https://docs.tabnine.com/main - Documentazione ufficiale TabNine

[11] https://docs.aws.amazon.com/codewhisperer/ - Documentazione ufficiale AmazonCodeWhisperer

[12] https://doi.org/10.1080/0144929X.2024.2431068 - Torkamaan, H., Steinert, S., Pera, M. S., Kudina, O., Freire, S. K., Verma, H., … Oviedo-Trespalacios, O. (2024). Challenges and future directions for integration of large language models into socio-technical systems. *Behaviour & Information Technology*, 1–20.