ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Sviluppo di un assistente AI prototipale per Visual Studio Code

Tesi di laurea in:
Programmazione ad Oggetti

Relatore
Prof. Mirko Viroli

Candidato
Gioele Bucci

Correlatori
Dott. Gianluca Aguzzi
Dott. Nicolas Farabegoli

III Sessione di Laurea
Anno Accademico 2024-2025

Sommario

Gli assistenti alla programmazione basati sull'intelligenza artificiale sono ormai una presenza comune negli ambienti di sviluppo moderni e la loro rapida evoluzione li sta rendendo sempre più capaci ed accessibili.
Data la crescente rilevanza di questi strumenti nell'ambito dello sviluppo software, la presente tesi si propone di approfondirne la conoscenza attraverso un duplice approccio, che combini l'analisi di una soluzione già esistente e consolidata allo sviluppo pratico di un assistente AI prototipale.

Per farlo si è partiti da un'analisi delle funzionalità principali di GitHub Copilot, da cui sono stati ricavati i requisiti di un assistente AI generico, in grado di offrire completamenti del codice inline, azioni di interazione contestuale per la modifica del codice e una chat conversazionale. Successivamente, sulla base dei requisiti emersi, è stato progettato e implementato un prototipo funzionante, integrandolo all'interno dell'ambiente di sviluppo Visual Studio Code sotto forma di estensione.

Parole chiave: assistenti AI, intelligenza artificiale, GitHub Copilot, Visual Studio Code.

A Nicolò.

Indice

Introduzione

Gli assistenti alla programmazione basati sull'intelligenza artificiale stanno diventando sempre più capaci ed accessibili, acquisendo una crescente rilevanza nel mondo dello sviluppo software. Questi strumenti, inizialmente limitati al suggerimento o auto-completamento del codice, stanno attraversando una fase di rapida evoluzione che continua ad ampliarne le capacità e sono ormai integrati nella maggior parte degli ambienti di sviluppo moderni.
Le motivazioni dietro a questa evoluzione vanno ricondotte principalmente agli enormi progressi recentemente compiuti nell'ambito dei Large Language Models, sui quali gli assistenti AI si basano: poiché modelli migliori si traducono in assistenti più capaci, questi sono diventati in grado di eseguire compiti di ben più alto livello, arrivando ad ottenere capacità "agentiche" che consentono loro di progettare, sviluppare e testare software in maniera pressoché autonoma.

Di fronte ad un tale scenario risulta fondamentale comprendere i meccanismi alla base del funzionamento degli assistenti AI, analizzando come le capacità generative dei Large Language Models possano essere sfruttate per la loro implementazione ed esplorando le metodologie che ne consentono l'integrazione all'interno di un ambiente di sviluppo. Per soddisfare tali esigenze, la presente tesi si propone di approfondire lo studio di questi strumenti mediante la realizzazione di un assistente AI prototipale, prendendo come modello di riferimento una soluzione analoga già affermata.

In una prima fase di analisi verranno identificate le feature principali di GitHub Copilot, individuato come caso di studio per via della sua ampia diffusione e capacità avanzate, al fine di delineare i requisiti fondamentali di un generico assistente AI. I requisiti emersi guideranno le successive fasi di progettazione e implementazione del prototipo, che verrà integrato all'interno di Visual Studio Code sotto forma di estensione, adottando le soluzioni architetturali necessarie per far sì che l'assistente realizzato risulti performante, stabile e integrato nativamente all'interno dell'IDE.

Al termine del processo di sviluppo, l'assistente verrà sottoposto ad un processo di validazione basato su una serie di test applicativi mirati, volti a verificare il corretto

funzionamento delle sue funzionalità principali: un sistema di auto-completamento del codice inline, un'interfaccia di chat conversazionale e delle azioni di interazione contestuale per la modifica del codice.

Struttura della Tesi

La trattazione verrà articolata come segue:

- **Capitolo 1: Background** Breve introduzione sui fondamenti tecnologici alla base degli assistenti AI e analisi dei trend che ne hanno guidato l'evoluzione.
- **Capitolo 2: Analisi** Indagine su GitHub Copilot, identificandone le feature principali necessarie a delineare i requisiti per l'assistente da realizzare.
- **Capitolo 3: Design** Progettazione dell'assistente sulla base dei requisiti emersi e descrizione dell'architettura software adottata.
- **Capitolo 4: Implementazione** Sviluppo dell'assistente come estensione per Visual Studio Code, con l'obiettivo di ricreare le funzionalità emerse nella fase di analisi nel rispetto dei vincoli e le best practice imposte dall'IDE.
- **Capitolo 5: Dimostrazione** Dimostrazione delle capacità dell'assistente realizzato.
- **Capitolo 6: Conclusioni** Sintesi dei risultati ottenuti e prospettive future.

Capitolo 1

Background

1.1 AI-assisted programming e LLMs

Con l'espressione "AI-assisted programming" si intende l'utilizzo di tecnologie che impiegano l'intelligenza artificiale per supportare il programmatore nel processo di sviluppo. In questo ambito si collocano gli assistenti alla programmazione basati sull'intelligenza artificiale, anche conosciuti come *AI pair programmers*, d'ora in poi indicati più brevemente come "assistenti AI".
Questi rappresentano un'evoluzione significativa rispetto ad altri strumenti tradizionalmente integrati all'interno degli ambienti di sviluppo, come linter o syntax highlighter: infatti, a differenza di un supporto "sintattico", gli assistenti AI offrono funzionalità proattive ben più avanzate, che includono il suggerimento di correzioni, la spiegazione di porzioni di codice e la generazione di interi file sorgente a partire da descrizioni fornite dall'utente in linguaggio naturale.

Dietro al funzionamento di questi strumenti risiedono i Large Language Models (LLMs): si tratta di reti neurali di grandi dimensioni, con un numero di parametri che varia dai miliardi fino ai trilioni (10^{12}), specificamente progettate per eseguire compiti di *natural language processing*[1] legati alla comprensione e generazione di testo in linguaggio naturale.
Questi modelli vengono addestrati utilizzando enormi quantità di dati testuali

[1] ibm.com/think/topics/natural-language-processing

eterogenei, come libri, documenti o contenuti web e sono basati su architetture di tipo *Transformer*[2] che eccellono nel riconoscimento dei pattern presenti all'interno dei dati, consentendo al modello di apprendere le varie relazioni presenti tra parole e concetti.

Le capacità di comprensione e generazione del testo dei LLMs derivano dal processo di addestramento primario al quale vengono sottoposti, detto *pre-training*, durante il quale questi imparano a prevedere la parola successiva all'interno di una data sequenza di testo, operando di fatto come enormi motori di predizione statistica.
Sebbene questi modelli nascano come strumenti per la predizione del testo, esistono due tecniche principali (spesso complementari) che consentono di adattare un LLM pre-addestrato ad un contesto più specifico.

La tecnica più strutturata è il *fine-tuning*, un processo di "specializzazione" che consiste nel ri-addestrare il modello su un dataset più piccolo e mirato, con l'obiettivo di adattarne i parametri per un particolare dominio applicativo, mantenendo al contempo le capacità generali apprese durante la fase di pre-training.
Prendendo come dataset di riferimento svariati repository contenenti codice open-source, questo approccio ha reso possibile la realizzazione di modelli in grado di generare codice corretto e funzionante a partire da istruzioni in linguaggio naturale, ponendo le basi per la nascita degli assistenti AI odierni.
Con l'aumentare delle capacità dei modelli, e in particolare grazie all'espansione della loro *context window* (ovvero la quantità di testo ricevuto in input che un modello può "ricordare"), è emersa una tecnica alternativa più flessibile: il *prompt engineering*. Questo approccio si basa sulla formulazione di istruzioni (prompt) dettagliate e ricche di contesto, che consentono di guidare un modello generalista nell'esecuzione di compiti specializzati in tempo reale, senza la necessità di doverlo sottoporre ad un ulteriore processo di addestramento.

1.1.1 Knowledge cutoff e grounding

Un'importante limitazione dei LLMs deriva dal fatto che i dati utilizzati nel loro addestramento, per quanto estesi, rappresentano comunque un'"istantanea" del

[2] ibm.com/think/topics/transformer-model

mondo in un determinato momento.
Ciò significa che esiste una data, nota come *knowledge cutoff*, oltre la quale un modello non dispone più di informazioni aggiornate, poiché queste non erano incluse nei suoi dati di training: di conseguenza, se interrogato su eventi successivi a tale data, un LLM fornirà risposte obsolete o inaccurate.
Questo risulta particolarmente problematico per modelli addestrati a svolgere compiti di programmazione poiché, ad esempio, potrebbero non essere a conoscenza di vulnerabilità recentemente scoperte o di aggiornamenti nelle librerie e nei framework più utilizzati, fornendo codice obsoleto, non funzionante o addirittura vulnerabile.

Per superare questo limite, sono stati sviluppati dei meccanismi di *grounding*, che consentono ad un modello, qualora questo lo ritenga necessario, di effettuare ricerche sul web. Questa tecnica aiuta a mitigare fortemente le problematiche legate al knowledge cutoff e consente al modello di fornire risposte aggiornate e pertinenti anche su contenuti non presenti nei propri dati di addestramento.

1.2 Funzionamento di un assistente AI

Un assistente AI opera attraverso un costante scambio di informazioni effettuato tra l'ambiente di sviluppo (IDE) nel quale è integrato, l'utente e il LLM.
Il suo funzionamento può essere suddiviso in tre fasi principali:

1. **Raccolta del contesto**: In seguito ad una richiesta effettuata dall'utente, l'IDE procede con la raccolta del contesto: con questo termine si intende l'insieme di tutte le informazioni che verranno fornite al LLM per consentirgli di comprendere la richiesta dell'utente e generare una risposta pertinente.
   Il contesto viene costruito automaticamente dall'IDE, combinando informazioni come il contenuto del file corrente, la posizione del cursore ed eventuale codice selezionato, ma può anche essere arricchito dall'utente tramite l'aggiunta di ulteriori file reputati rilevanti per il compito da svolgere.

2. **Elaborazione tramite LLM**: Il contesto raccolto viene unito alla richiesta effettuata dall'utente per formare un *prompt*, ovvero l'input testuale da

fornire al LLM. Il modello analizza il prompt ricevuto e, in base al tipo di interazione richiesta, genera un'opportuna risposta, come una porzione di codice o una spiegazione.

3. **Presentazione dei risultati**: Al termine della generazione, l'output del modello viene ricevuto dall'IDE e presentato all'utente, che può accettare, rifiutare o reiterare sulla risposta, rincominciando il ciclo.

Dal punto di vista architetturale, la maggior parte degli assistenti AI disponibili sul mercato adottano un approccio client-server, in cui l'IDE funge da client e invia le varie richieste ad un server remoto che ospita il modello, ma esistono anche delle alternative open-source che consentono di interfacciarsi con LLMs che eseguono localmente, riducendo la latenza e preservando la privacy del codice, solitamente a scapito di prestazioni inferiori.

1.2.1 Funzionalità agentiche

A partire dal 2025, assistenti di nuova generazione come Claude Code[3] e GitHub Copilot[4] sono stati dotati di funzionalità "agentiche", acquisendo la capacità di agire autonomamente all'interno dell'IDE, come mostrato in Figura 1.1.
Al fine di svolgere un compito assegnato dall'utente, questi assistenti possono autonomamente decidere di eseguire comandi, modificare file, effettuare ricerche nel progetto ed interagire con strumenti esterni, divenendo in grado di scrivere, testare e debuggare interi applicativi in maniera quasi interamente autonoma.

1.3 Diffusione degli assistenti AI

Al fine di comprendere appieno la natura della trasformazione tecnologica innescata dall'avvento degli assistenti AI, è necessario analizzare dati e trend particolarmente significativi che evidenziano la portata del fenomeno e i fattori che ne stanno guidando la diffusione.

[3] anthropic.com/claude-code
[4] github.com/features/copilot

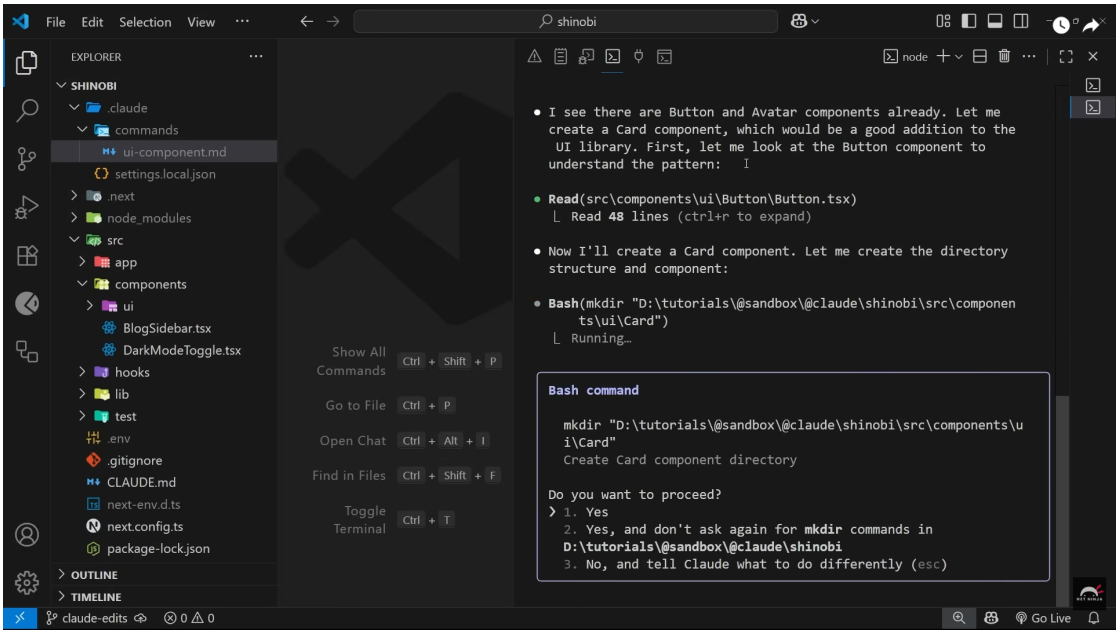

Figura 1.1: Esempio di utilizzo delle funzionalità agentiche di Claude Code: in seguito ad una richiesta di alto livello da parte dell'utente (implementare un nuovo componente grafico all'interno di una pagina web), l'assistente procede autonomamente ad analizzare la codebase ed eseguire i comandi per la creazione dei file e le directory necessarie.

Una delle fonti più autorevoli per misurare le tendenze nell'ambito dello sviluppo software è l'annuale "Stack Overflow Developer Survey", i cui dati del 2024 offrono un'immagine concreta dell'impatto che hanno avuto gli assistenti AI nel settore.
Dei più di 60.000 sviluppatori che hanno preso parte al sondaggio, il 76% dichiarava di utilizzare (o voler iniziare ad utilizzare) tali strumenti, percentuale che è salita ad 84% nell'indagine del 2025[5].
Inoltre, sebbene gli sviluppatori avessero pareri discordanti riguardo l'accuratezza degli output generati e la capacità degli assistenti AI nell'eseguire compiti complessi, la maggioranza (81%) ha riconosciuto un aumento della produttività come il principale beneficio derivante dall'utilizzo di tali strumenti.
Questa percezione è stata confermata anche da varie evidenze empiriche, come uno studio del 2024 che ha coinvolto circa 5000 sviluppatori di aziende come Microsoft e Accenture, i cui risultati mostrano come l'accesso ad un AI assistant fosse in

[5]survey.stackoverflow.co/2024/ai e survey.stackoverflow.co/2025/ai

grado di aumentare il numero di task completati del 26,08%, con benefici maggiori registrati tra gli sviluppatori meno esperti [CDJ+24].

Uno dei fattori chiave dietro all'adozione degli assistenti AI su larga scala va ricondotto ai progressi compiuti nel campo dei Large Language Models, ed è la drastica riduzione dei costi di inferenza dei modelli, ovvero i costi associati all'utilizzo di un LLM pre-addestrato. Per illustrare la portata del fenomeno, si consideri che il costo per raggiungere performance paragonabili a quelle di GPT-3.5 (prendendo come riferimento il benchmark MMLU in Figura 1.2) è sceso da 20 dollari per milione di token a novembre 2022 a soli 0,07 dollari ad ottobre 2024, una riduzione nei costi di inferenza di oltre 280 volte in meno di due anni [MFP+25, p. 64].

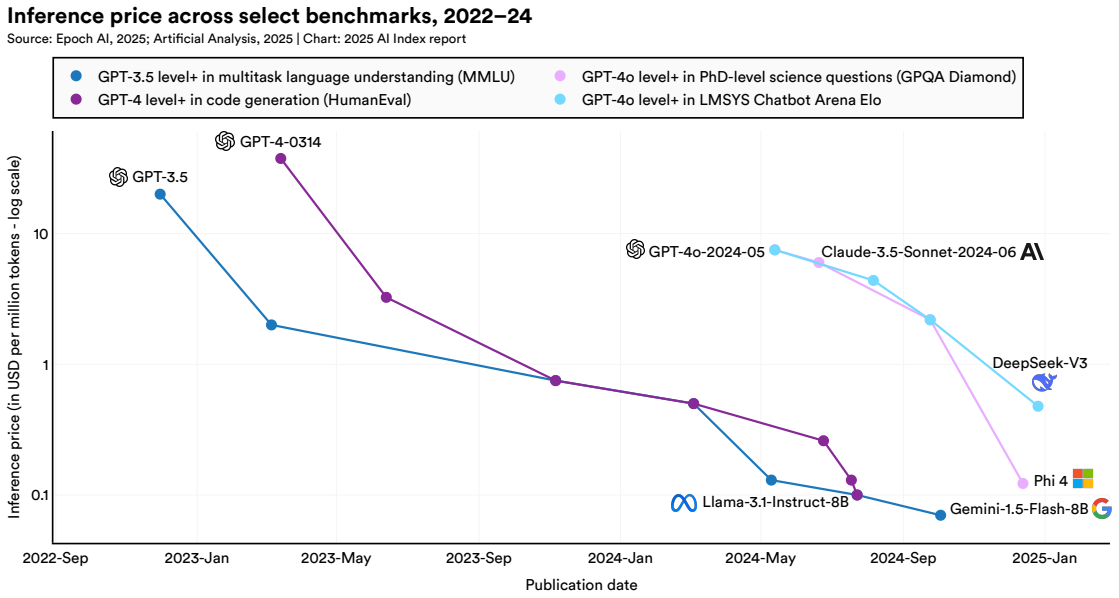

Figura 1.2: Riduzione del costo di inferenza a parità di performance su vari benchmark.

Un tale abbattimento dei costi d'utilizzo, in combinazione al costante aumento delle performance dei modelli, che in molti ambiti superano già le capacità umane [MFP+25, p. 64], ha consentito agli assistenti AI di diventare in breve tempo degli strumenti economicamente sostenibili, favorendone la diffusione.

Un ulteriore fattore da considerare è la progressiva riduzione del divario prestazionale tra i LLM proprietari (*closed-weight*) e quelli *open-weight*.

La distinzione fondamentale tra le due tipologie risiede nell'accessibilità ai "pesi" del modello, ovvero ai suoi parametri interni. I modelli closed-weight (come la serie GPT di OpenAI) sono accessibili unicamente tramite API e mantengono privati i loro pesi e la loro architettura. D'altro canto, i modelli open-weight (come Llama 3.1 o DeepSeek V3) rendono i propri pesi pubblicamente disponibili, permettendo a chiunque di ispezionarli ed adattarli a scopi specifici.
Come si può osservare in Figura 1.3, le capacità dei principali modelli open-weight si stanno rapidamente avvicinando a quelle delle controparti closed-weight, giustificandone la crescente adozione come modelli alla base di svariati assistenti AI non-proprietari [MFP+25, pp. 95–96].

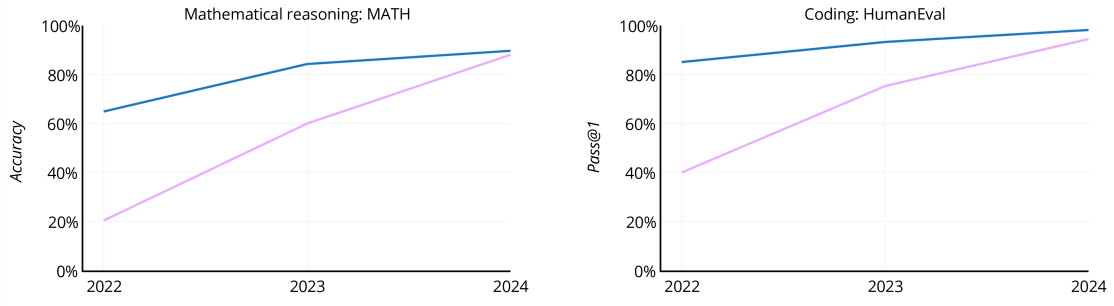

Figura 1.3: Confronto tra le performance dei migliori modelli closed-weight (in blu) e open-weight (in rosa) tra il 2022 e il 2024 su benchmark di ragionamento matematico (MATH) e di programmazione (HumanEval).

Capitolo 2

Analisi

L'obiettivo della presente tesi è quello di realizzare un assistente AI prototipale in grado di supportare il programmatore durante il processo di sviluppo software.
Tale assistente dovrà essere integrato all'interno di un IDE e presentare funzionalità analoghe a quelle degli assistenti alla programmazione moderni, come l'autocompletamento del codice e la possibilità di eseguire modifiche sui file sorgente in base alle istruzioni fornite dall'utente in linguaggio naturale.

Per poter delineare gli esatti requisiti dell'assistente da realizzare risulta utile analizzare le funzionalità offerte da un simile prodotto già esistente: la scelta è ricaduta su GitHub Copilot, un assistente sviluppato da GitHub in collaborazione con Microsoft e OpenAI, preso come caso di studio per via della sua vasta adozione e funzionalità avanzate.

Sebbene Copilot sia disponibile per diversi IDE, l'analisi che segue si concentrerà in modo specifico sulla sua implementazione per Visual Studio Code (d'ora in poi indicato come VS Code).
Tale decisione è motivata sia dalla popolarità dell'IDE, che negli ultimi anni si è affermato come il più utilizzato dagli sviluppatori[1], sia per la sua natura modulare basata su estensioni, rendendolo la piattaforma ideale per lo sviluppo e l'integrazione del prototipo oggetto di questa tesi.

[1] survey.stackoverflow.co/2025/technology

2.1 Feature principali di GitHub Copilot

2.1.1 Inline completion

Una delle feature principali di GitHub Copilot è l'*inline completion* (anche nota come *inline suggestion*), che fornisce suggerimenti di completamento del codice che appaiono direttamente all'interno dell'editor, come mostrato in Figura 2.1. Analizzando il contenuto del file corrente (includendo il codice già scritto ed eventuali commenti), l'assistente è in grado di suggerire in tempo reale righe o interi blocchi di codice pertinenti: questa capacità va ben oltre il semplice autocompletamento sintattico presente in molti IDE (funzionalità che su VS Code prende il nome di *Intellisense*[2]), fornendo suggerimenti che spesso consistono in interi blocchi di codice coerenti con l'intento del programmatore.

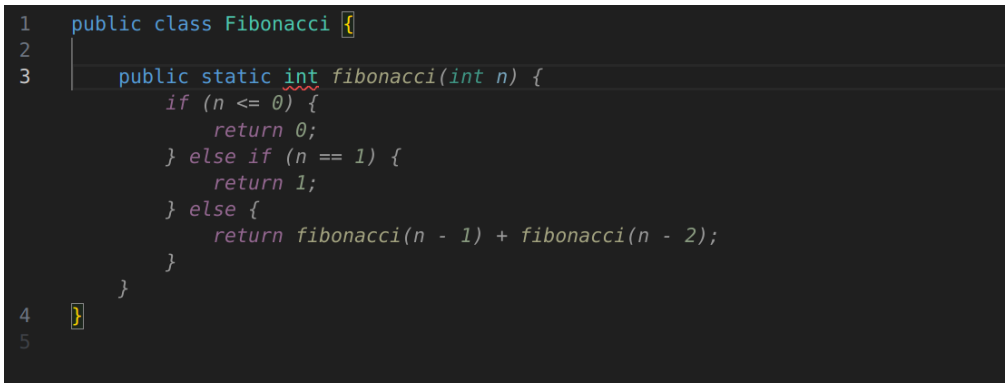

Figura 2.1: Esempio di code completion. Dopo aver iniziato la scrittura della definizione del metodo `fibonacci`, Copilot ne suggerisce l'intera implementazione (il suggerimento è il codice in trasparenza).

2.1.2 Interazione contestuale

Oltre al completamento del codice "passivo" fornito dalle inline completions, Copilot offre anche vari strumenti di interazione diretta che consentono all'utente di interagire con l'assistente formulando le proprie richieste in linguaggio naturale. Tra queste vi sono le azioni di interazione contestuale, che consentono all'utente

[2] code.visualstudio.com/docs/editing/intellisense

di modificare il contenuto del file corrente o la sola porzione di codice selezionata, come mostrato in Figura 2.2.

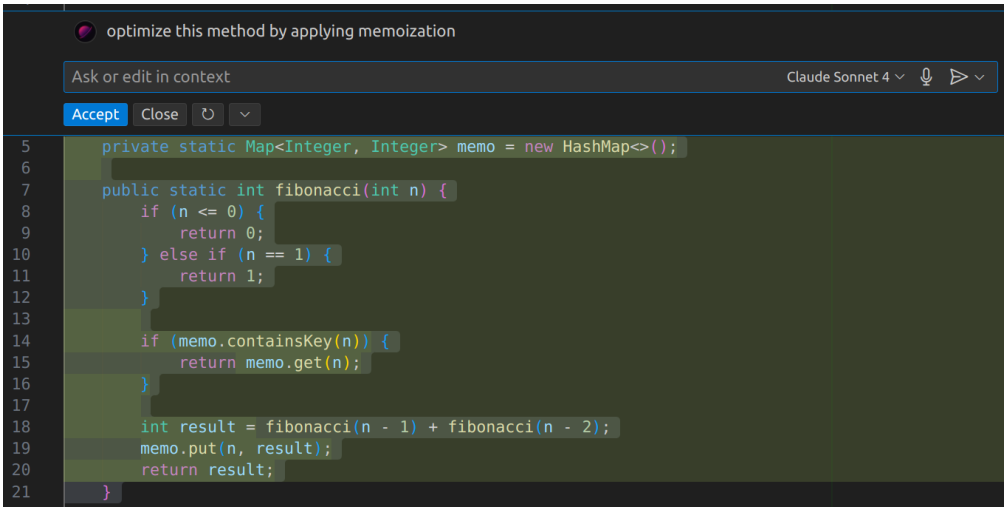

Figura 2.2: Esempio di utilizzo dell'interazione contestuale per effettuare una modifica al metodo `fibonacci` mostrato in Figura 2.1, in seguito ad una richiesta di ottimizzazione da parte dell'utente.

2.1.3 Chat

Oltre agli strumenti integrati direttamente nell'editor, Copilot mette a disposizione anche un pannello di chat che permette di dialogare liberamente con l'assistente. Questa modalità d'uso offre un ambiente separato che l'utente può utilizzare per vari scopi, come ottenere spiegazioni dettagliate, suggerimenti architetturali o supporto nella risoluzione di errori (come mostrato in Figura 2.3).
La chat mantiene ovviamente l'intero contesto della conversazione, per garantire che le risposte generate siano sempre pertinenti. Inoltre, tale contesto può essere ulteriormente arricchito dall'utente, che può aggiungervi ulteriori file presenti nell'IDE, permettendo all'assistente di leggerne il contenuto e fornire così risposte ancor più mirate.

Funzionalità agentiche

È opportuno sottolineare che a partire dal 2025 le funzionalità del pannello di chat sono state notevolmente ampliate con l'introduzione di una "modalità agen-

te" (*agent mode*) che permette a Copilot di svolgere compiti di più alto livello. Grazie a questa feature l'utente può assegnare un certo obiettivo all'assistente, lasciando che questo lo svolga in autonomia, individuando i file da modificare e utilizzando eventuali comandi o strumenti reputati necessari (ad esempio installando dipendenze tramite terminale), procedendo iterativamente fino al completamento del compito.

Queste funzionalità agentiche, oltre ad essere feature sperimentali soggette a continui cambiamenti, trasformano l'assistente da elemento di supporto al programmatore ad esecutore autonomo e una tale transizione ne giustifica l'esclusione da questo studio. L'obiettivo della presente tesi rimane infatti la progettazione e l'implementazione di un assistente nel senso più stretto del termine e per tale ragione le funzionalità agentiche non verranno ulteriormente analizzate, né verranno implementate nel prototipo.

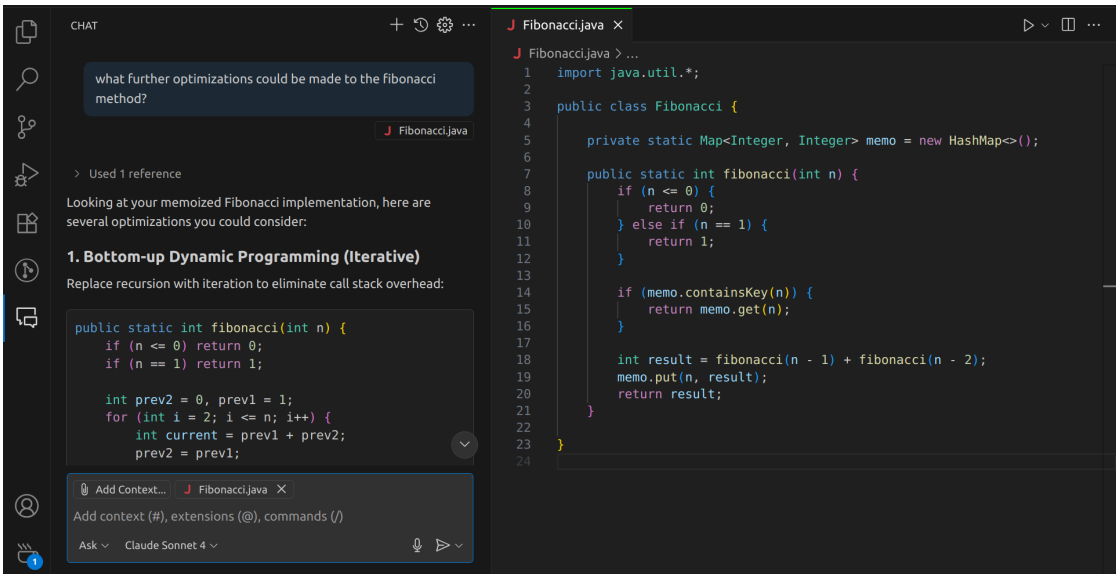

Figura 2.3: Conversazione tra utente e assistente riguardo ulteriori ottimizzazioni al metodo `fibonacci` in cui l'assistente suggerisce un approccio implementativo alternativo, motivandone i vantaggi rispetto alla soluzione corrente.

2.1.4 Istruzioni personalizzate

Al fine di allineare gli output generati dall'assistente alle specificità di un determinato progetto, Copilot mette a disposizione un meccanismo di personalizzazione noto come *custom instructions* ("istruzioni personalizzate").
Modificando un apposito file (`.github/copilot-instructions.md`), l'utente può definire un insieme di direttive in linguaggio naturale al fine di modificare il comportamento dell'assistente. Tali linee guida influenzano sia il codice generato dall'assistente che le risposte fornite in chat, permettendo di rendere gli output generati conformi ad eventuali standard, pratiche di sviluppo o requisiti architetturali.

```
1 # Coding Guidelines for TypeScript Projects
2
3 When writing TypeScript code, adhere to the following guidelines:
4
5 - Use descriptive names for methods and keep them short and single-purpose.
6 - Add JSDoc comments to public methods only.
7 - Make sure to sanitize any user input.
8 - Never hardcode secrets (i.e. API keys, credentials...)
9 - Use `try/catch` for operations that can fail.
```

Listato 2.1: Esempio di file di istruzioni personalizzate nel quale vengono fornite istruzioni aggiuntive all'assistente da utilizzare per la scrittura di codice TypeScript.

2.2 Requisiti funzionali

RF1 L'assistente dovrà esporre delle azioni di interazione contestuale, che consentano di modificare il contenuto del file corrente.

RF2 Tutte le funzionalità di interazione contestuale dovranno consentire di limitare l'intervento dell'assistente sulla sola porzione di codice selezionata dall'utente.

RF3 L'assistente dovrà poter supportare un dialogo conversazionale con l'utente, esponendo tale funzionalità attraverso un apposito pannello di chat.

RF4 L'utente dovrà poter includere file e risorse aggiuntive nel contesto della chat.

RF5 L'utente dovrà poter modificare i diversi aspetti operativi dell'assistente, come il provider AI, il modello utilizzato e i parametri di configurazione ad esso associati.

RF6 L'assistente deve consentire all'utente di poter abilitare o disabilitare l'utilizzo del grounding, qualora il provider scelto supporti tale funzionalità.

RF7 L'assistente dovrà fornire automaticamente suggerimenti inline in tempo reale, pertinenti al contenuto del file corrente.

RF8 Il comportamento dell'assistente dovrà adattarsi ad eventuali direttive aggiuntive definite dall'utente all'interno di un apposito file di configurazione.

2.3 Requisiti non funzionali

RNF1 L'architettura del sistema dovrà essere modulare per garantire la separazione delle responsabilità e la futura estendibilità.

RNF2 L'architettura del sistema dovrà ottimizzare l'utilizzo delle risorse per far sì che l'integrazione dell'assistente nell'IDE risulti performante e stabile.

RNF3 Le operazioni a lunga esecuzione, come le interazioni con i LLMs, dovranno essere gestite in modo asincrono per non compromettere la reattività dell'IDE.

2.4 Analisi del dominio

L'assistente AI dovrà gestire il flusso di informazioni e le interazioni tra tre entità principali: l'utente, l'IDE e il LLM.

Come mostrato nel diagramma di flusso riportato in Figura 2.4, tali interazioni formano un ciclo operativo che si attiverà in seguito ad una richiesta dell'utente, che potrà essere esplicita, come una domanda posta nella chat, o implicita, come un autocompletamento innescato durante la digitazione del codice.

In seguito l'assistente dovrà interagire con l'IDE per raccogliere il contesto necessario, unendolo alla richiesta dell'utente ed eventuali istruzioni aggiuntive per formare un prompt, che verrà elaborato da un LLM esterno.
Ricevuta la risposta del modello l'assistente si dovrà occupare di processarla e presentarla all'utente, concludendo il ciclo e rimanendo in attesa di una nuova interazione.

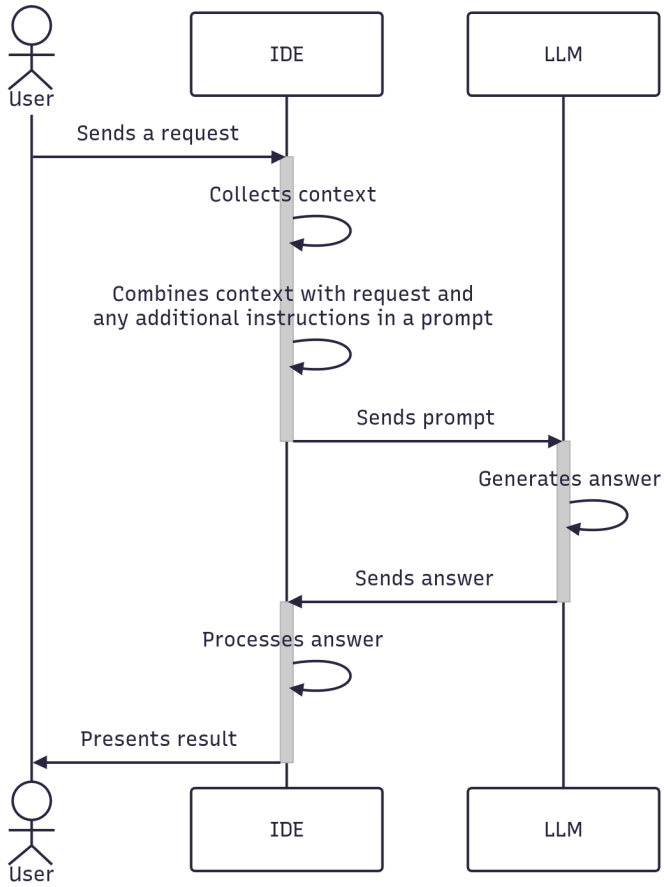

Figura 2.4: Flusso operativo dell'assistente AI, che mostra la principale sequenza di interazioni tra utente, IDE e LLM.

Capitolo 3

Design

Nel presente capitolo verrà definito il modello architetturale dell'assistente AI da realizzare, sulla base dai requisiti emersi nella precedente fase di analisi. Ciò verra fatto attraverso la definizione dei componenti logici fondamentali di un generico AI-assisted IDE, astraendo dai vincoli che verrebbero inevitabilmente imposti dalla scelta di una piattaforma specifica.

I dettagli implementativi verranno rimandati al prossimo capitolo, nel quale verrà descritto il processo di mappatura dell'architettura logica emersa nel contesto di VS Code, mantenendo in considerazione i vincoli e le best practice imposti dell'ambiente.

3.1 Architettura service-oriented

Problema Come stabilito dal RNF1, è necessario definire un'architettura modulare facilmente estendibile, che separi chiaramente le responsabilità dei vari componenti del sistema e al contempo ottimizzi l'utilizzo delle risorse (RNF2).

Soluzione È stata adottata un'architettura service-oriented, mostrata in Figura 3.1, che utilizza il pattern Factory Method per consentire l'inizializzazione "on-demand" dei vari servizi a runtime (*lazy instantiation*).

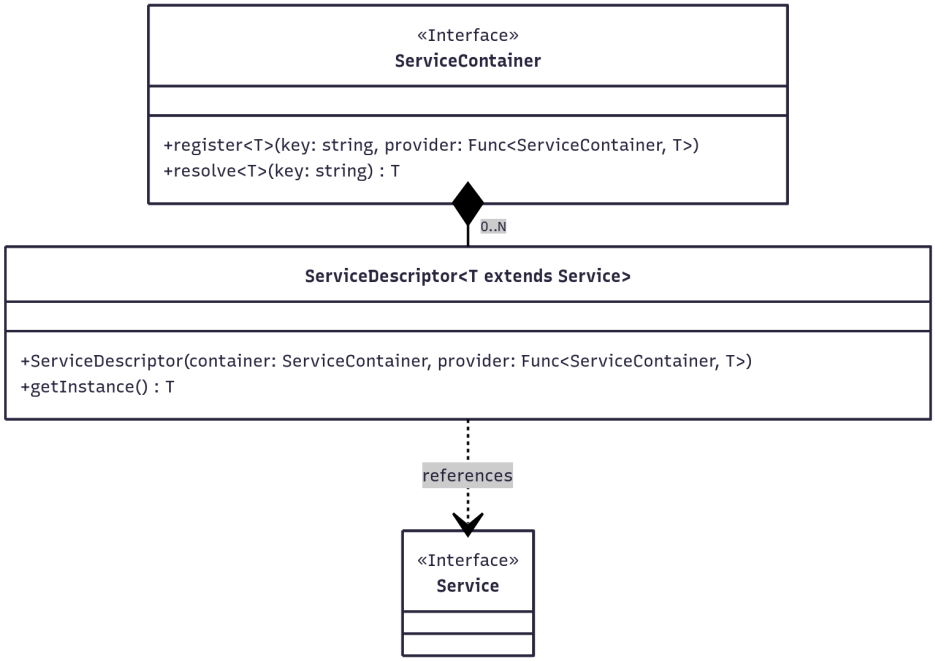

Figura 3.1: Architettura service-oriented per la modularizzazione del sistema e la gestione ottimizzata delle risorse tramite lazy loading.

Di seguito viene data una descrizione più dettagliata dei componenti dell'architettura presentata:

- **ServiceContainer**: un DI container (Dependency Injection container) progettato ad-hoc per la registrazione, creazione e risoluzione dei vari servizi. Espone un metodo `register` che consente di registrare un servizio senza però instanziarlo immediatamente (associandone il factory method ad una certa chiave) andando a crearne l'istanza solamente quando tale servizio viene richiesto per la prima volta tramite il metodo `resolve`. In seguito alla creazione di un servizio, la relativa istanza viene conservata e riutilizzata per tutte le richieste successive, comportandosi di fatto come un singleton.
- **ServiceDescriptor**: un semplice proxy che incapsula il factory method di un dato servizio (identificato da un'interfaccia tag) e ne gestisce l'istanza singleton. Poiché il factory method riceve il `ServiceContainer` come parametro, ciascun servizio è in grado di risolvere eventuali dipendenze interne al momento della creazione, realizzando il pattern di Dependency Injection.

Problema Occorre evitare errori a runtime causati, ad esempio, dalla richiesta di un servizio non esistente o una chiave digitata in modo errato. Per farlo è necessario garantire la coerenza dei tipi tra la chiave utilizzata per la registrazione di un servizio nel DI container e l'oggetto restituito dalla sua risoluzione.

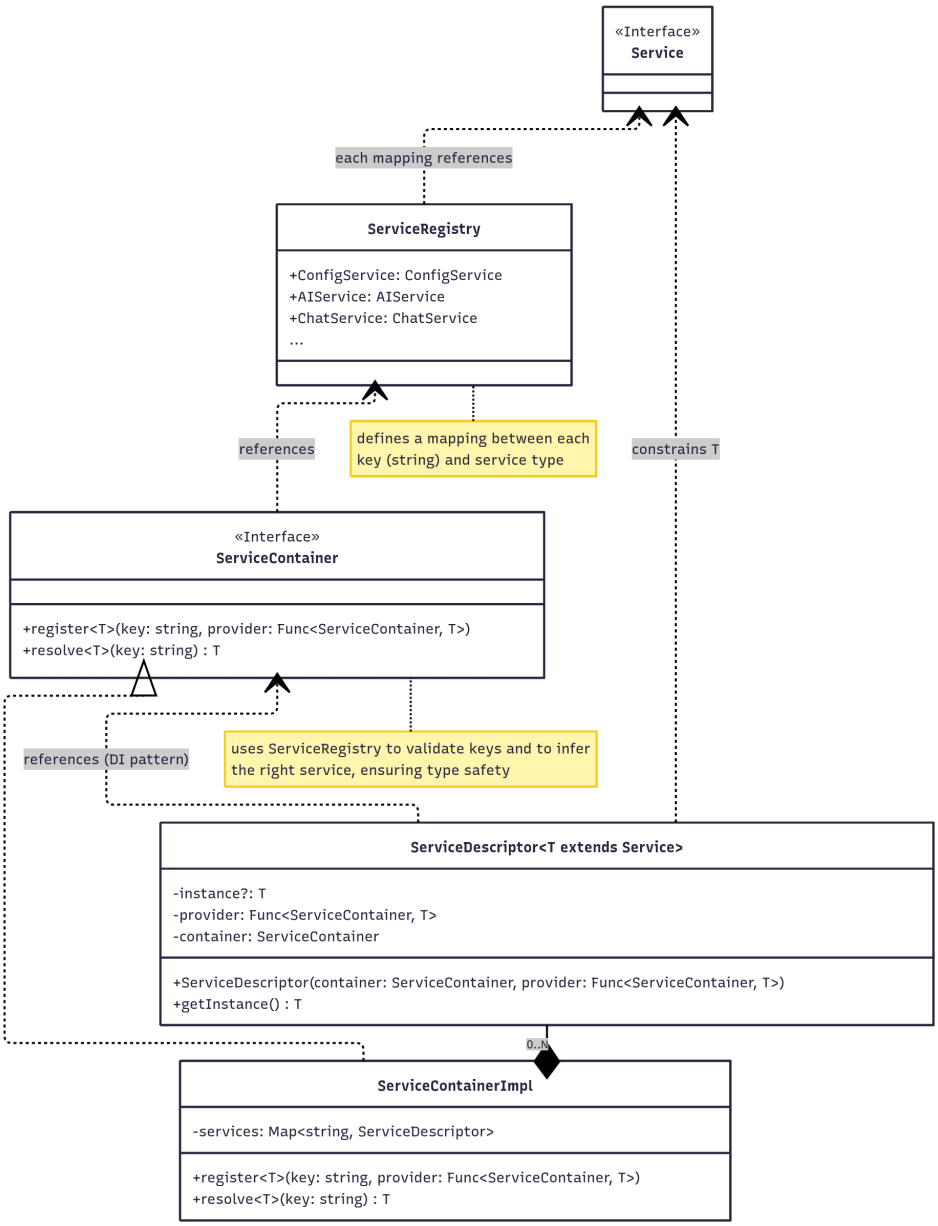

Figura 3.2: Architettura service-oriented completa e dettagliata. L'aggiunta di `ServiceRegistry` garantisce type-safety nella registrazione e risoluzione dei servizi.

Soluzione Come mostrato in Figura 3.2, l'architettura service-oriented precedentemente presentata è stata estesa introducendo `ServiceRegistry`, che stabilisce una relazione tra la stringa letterale (la chiave) di un certo servizio e il relativo tipo dell'istanza ad esso associata.

3.2 Servizi principali

Definita l'architettura service-oriented su cui si baserà il progetto, si può procedere con la definizione dei componenti logici fondamentali di un generico assistente AI.

3.2.1 ConfigService

Indipendentemente dall'IDE in cui viene integrato, l'assistente AI esiste come un'estensione dell'ambiente, che l'utente può attivare o disattivare a seconda delle necessità. In fase di design questo ciclo di vita verrà modellato dal componente generico `ExtensionManager`, che espone dei rispettivi metodi `activate` e `deactivate` (la cui implementazione effettiva dipenderà ovviamente dall'IDE specifico).

Problema Come stabilito dal RF5, l'utente deve poter modificare il comportamento dell'assistente configurandone diversi parametri, inclusi alcuni la cui definizione deve essere resa obbligatoria (come la chiave API e il provider AI da utilizzare). È perciò necessario un componente in grado di leggere e validare tali configurazioni all'avvio dell'assistente, mettendole a disposizione in maniera strutturata al resto del sistema.

Soluzione È stato creato un servizio dedicato `ConfigService`, riportato in Figura 3.3, che espone due metodi:

- `isConfigValid`, che valida la configurazione utente e verifica la presenza dei parametri obbligatori. Questo metodo verrà chiamato da `ExtensionManager` all'avvio dell'assistente, per assicurarsi che l'ambiente sia correttamente configurato prima di procedere con l'inizializzazione degli altri componenti.

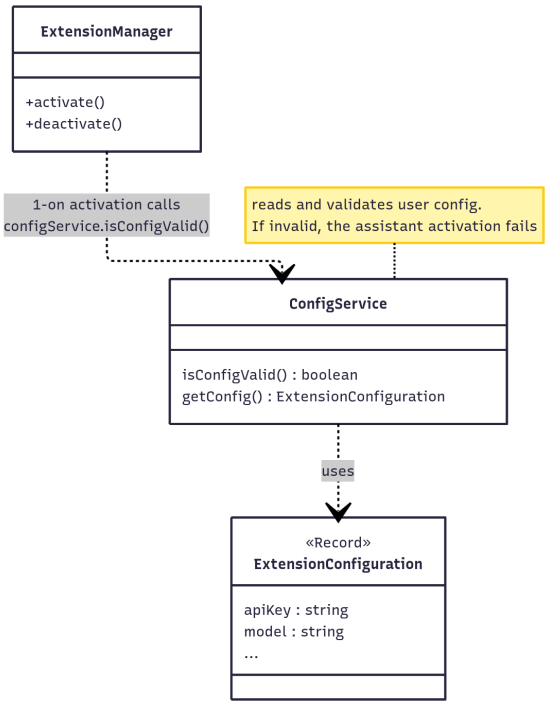

Figura 3.3: Architettura di `ConfigService`, responsabile del controllo della configurazione utente all'avvio dell'assistente.

- `getConfig`, che restituisce la configurazione utente in maniera strutturata come un oggetto di tipo `ExtensionConfiguration`.

3.2.2 AIService

Un elemento fondamentale di qualunque AI-assisted IDE è il componente responsabile della comunicazione con i LLMs esterni. Questo si occupa di raccogliere ogni richiesta del sistema verso i modelli AI, gestendo internamente aspetti come l'assemblaggio del prompt da fornire al modello e l'effettiva interazione con l'API remota dello specifico provider.

Problema Creare un punto di accesso centralizzato che consenta la comunicazione trasparente tra i vari componenti del sistema e i modelli AI esterni, indipendentemente dalle specifiche modalità di interazione effettivamente richieste dal provider sottostante.

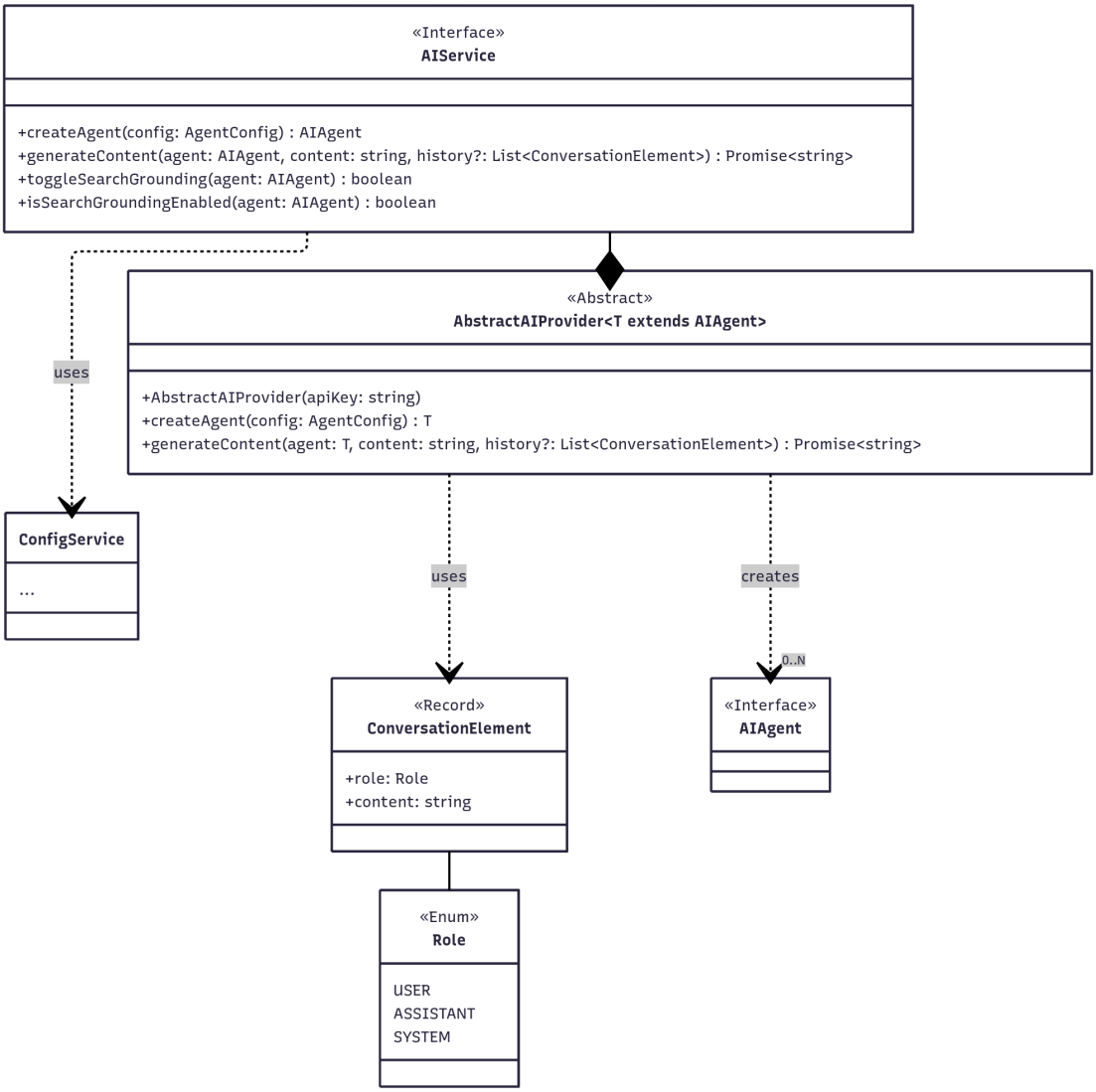

Figura 3.4: Applicazione del pattern Strategy per la gestione dei provider AI.

Soluzione È stato creato un servizio dedicato `AIService`, riportato in Figura 3.3, applicando il pattern Strategy per garantire che l'interazione con i vari modelli sia agnostica rispetto all'effettivo provider AI scelto dall'utente (come ad esempio OpenAI o Google).
L'architettura adottata supporta provider diversi, permettendo di poterne facilmente aggiungere di nuovi in futuro, ed è composta da tre elementi principali:

- **AIService**: servizio principale che delega l'esecuzione delle operazioni a pro-

vider intercambiabili in base alla configurazione scelta dall'utente (ottenuta tramite `ConfigService`), rispettando il principio di sostituzione di Liskov.
Il metodo che consente l'interazione con i LLMs è `generateContent`, che accetta come parametro opzionale una lista di `ConversationElement`, utilizzata per modellare l'eventuale cronologia dei messaggi, associando il contenuto testuale di ciascun messaggio (`content`) al ruolo (`role`) di chi lo ha inviato (utente, assistente o sistema).

- **AbstractAIProvider**: classe astratta che funge da base comune per l'implementazione di provider specifici: implementa una relativa interfaccia `AI Provider`, omessa dallo schema UML per brevità, e presenta un costruttore il cui unico argomento è l'"api key", cioè una stringa che varia in base al provider scelto e che funge da identificatore univoco necessario ad autorizzare l'assistente ad effettuare richieste verso la sua API.

- **AIAgent**: interfaccia tag che rappresenta un'istanza pre-configurata di un modello AI. La possibilità di creare più agenti è necessaria poiché le varie funzionalità dell'assistente potrebbero necessitare di agenti con istruzioni e potenzialmente anche configurazioni differenti.
  Ad esempio, un agente responsabile della funzionalità di modifica del codice dovrà essere istruito tramite system prompt a restituire esclusivamente codice sorgente, omettendo qualsiasi spiegazione in linguaggio naturale che ne impedirebbe l'applicazione diretta nell'editor.

3.2.3 ChatService

Problema Come richiesto dal RF3, occorre un pannello di chat in grado di gestire conversazioni "multi-turno" tra utente e assistente che mantenga un proprio stato interno, composto dalla cronologia dei messaggi e da eventuali file aggiunti al contesto da parte dell'utente (RF4).

Soluzione È stata adottata l'architettura riportata in Figura 3.5, basata sullo scambio di informazioni tra il pannello di chat e il resto del sistema, mediata dal

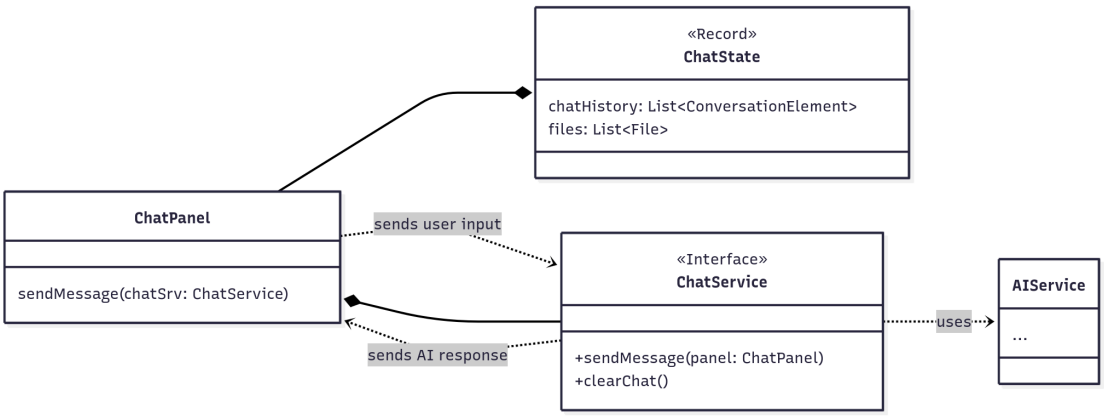

Figura 3.5: Architettura logica della funzionalità di chat.

servizio `ChatService`. Di seguito viene data una descrizione più dettagliata dei componenti dell'architettura presentata:

- **ChatPanel**: componente di presentazione della chat, sotto forma di un pannello dedicato all interno dell'IDE. È responsabile della visualizzazione dello stato della conversazione e della cattura degli input dell'utente, come l'invio di un messaggio nella chat, che vengono inviati a `ChatService` per essere elaborati.
- **ChatService**: agisce da mediatore tra il pannello di chat e il resto del sistema, gestendo i messaggi ricevuti da `ChatPanel`. Delega la generazione delle risposte dell'assistente ad `AIService` e si integra con l'API dell'IDE per fornire il contenuto dei file aggiunti al contesto. Al termine dell'elaborazione di un messaggio invia la risposta a `ChatPanel`, che si occuperà di presentarla all'utente aggiornando la propria interfaccia.
- **ChatState**: mantiene lo stato della chat, composto dalla cronologia dei messaggi scambiati tra utente e assistente, e da eventuali file ulteriori aggiunti al contesto.

È importante sottolineare che quella presentata in Figura 3.5 è un'architettura puramente logica, che si limita a definire le responsabilità e le interazioni fondamentali tra i componenti. Poiché la creazione di un pannello "custom" come quello richiesto dalla funzionalità di chat all'interno di un IDE richiede una forte inte-

grazione con l'API specifica dell'ambiente scelto, in fase di implementazione sarà inevitabile riadattare questa architettura per sottostare ai vincoli e le modalità di integrazione imposte dall'IDE nel quale l'assistente verrà integrato.

3.3 Comandi

Problema È necessario un meccanismo centralizzato per la gestione delle azioni invocabili dall'utente, in grado di verificare a priori la presenza dei prerequisiti richiesti per la loro esecuzione.

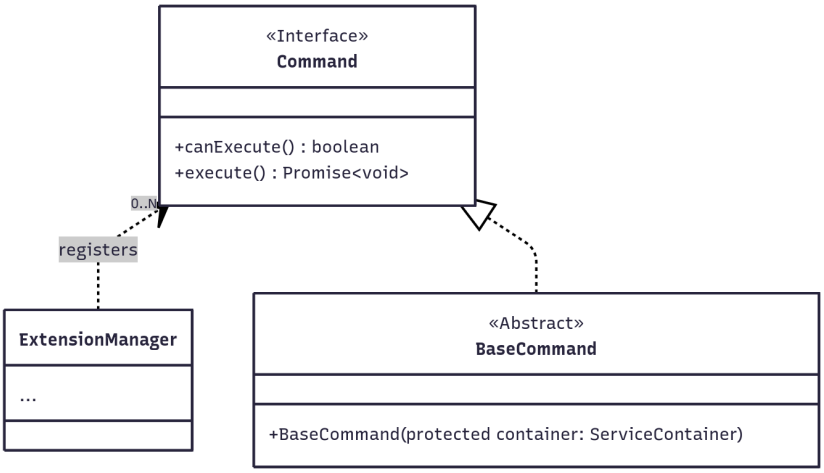

Figura 3.6: Architettura basata sul pattern Command adottata per le azioni invocabili dall'utente. La classe astratta `BaseCommand` si occupa di acquisire il service container per la risoluzione di eventuali dipendenze a runtime.

Soluzione Come mostrato in Figura 3.6, è stato applicato il pattern Command per incapsulare ogni azione in un comando specifico.
L'architettura adottata, sebbene utilizzata primariamente per le funzionalità di interazione contestuale, è sufficientemente generica da poter essere impiegata per qualsiasi tipo di comando aggiunto in futuro.

L'interfaccia `Command` espone due metodi principali:

- `canExecute`, che verifica se il comando è eseguibile nel contesto corrente dell'IDE. Ad esempio, un comando di refactoring del codice è eseguibile solamente se l'utente ha il focus su un editor di testo attivo.
- `execute`, che espone la logica operativa del comando. La restituzione di una `Promise` deriva dal RNF3 ed è fondamentale per garantire che un'operazione computazionalmente bloccante, come la comunicazione con un modello AI, sia gestita in modo asincrono, preservando la reattività dell'IDE.

3.3.1 Comandi di interazione contestuale

Per implementare le funzionalità di interazione contestuale richieste dal RF1, è necessario definire una serie di comandi in grado di modificare il contenuto del file corrente. Alcuni di questi necessitano di un input da parte dell'utente, come un comando di modifica generica (in cui l'utente fornisce le istruzioni per la modifica in linguaggio naturale), mentre altri operano in maniera autonoma, come un comando di refactoring che ristruttura automaticamente il codice senza bisogno di indicazioni.

Indipendentemente dalla loro natura, tutti i comandi di interazione contestuale dovranno essere in grado di operare sia sull'intero file corrente che, in alternativa, sulla sola porzione di codice selezionata dall'utente, come stabilito dal RF2.
È chiaro che questi condivideranno una serie di caratteristiche strutturali e comportamentali: infatti tutti richiedono un editor di testo attivo per poter essere eseguiti, estraggono il codice su cui operare in maniera analoga, invocano `AIService` per la generazione e infine sostituiscono il codice originale con il risultato generato.

Problema Fornire una base comune per l'implementazione dei comandi di interazione contestuale, in modo da evitare duplicazioni di codice e consentire l'aggiunta di ulteriori comandi con il minimo sforzo.

Soluzione L'architettura precedentemente definita è stata estesa applicando pattern Template Method. Come mostrato in Figura 3.7, il template method è `applyTransformation` e viene utilizzato dalla classe astratta `BaseEditorTransformer`,

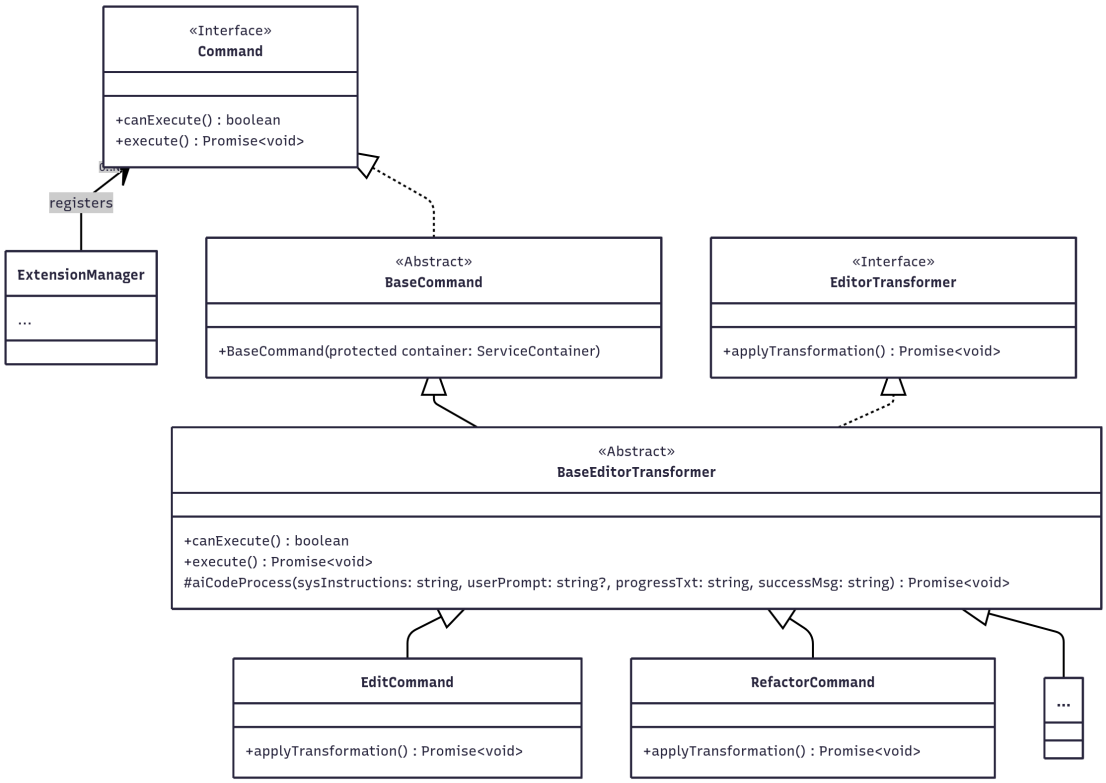

Figura 3.7: Architettura dei comandi completa.

introdotta per fungere da superclasse ai vari comandi di interazione contestuale, incapsulandone le funzionalità condivise nel rispetto del principio DRY.

La logica comune è contenuta all'interno del metodo protetto `aiCodeProcess` di `BaseEditorTransformer`, che accetta come parametri le istruzioni per l'agente (`systemInstructions`), le eventuali istruzioni fornite dall'utente (`userPrompt`), e due stringhe per personalizzare il contenuto del messaggio mostrato durante l'esecuzione del comando e quello mostrato al suo termine (`progressText` e `succ essMessage`).

Capitolo 4

Implementazione

4.1 Anatomia di un'estensione VS Code

Ciascuna estensione per VS Code necessita di un file manifest (`package.json`), che contiene metadati come il nome, la versione e soprattutto le modalità con cui l'estensione si integra all'interno dell'IDE.

Tale integrazione avviene dichiarando dei *Contribution Points*[1] ("punti di contribuzione"), ovvero una serie di dichiarazioni JSON inserite all'interno del manifest tramite le quali un'estensione può estendere le funzionalità dell'IDE.
Nel contesto del progetto realizzato sono stati dichiarati contribution points per:

- **Comandi** aggiuntivi, al fine di implementare le funzionalità di interazione contestuale, che l'utente può invocare tramite la palette dei comandi (definendo anche relativi keybindings, per associare scorciatoie da tastiera ai comandi implementati).

- **Configurazioni**, per consentire all'utente di modificare le proprietà dell'estensione tramite il file `settings.json`, permettendo di personalizzare aspetti come il modello AI da utilizzare e i vari parametri dell'agente.

[1] code.visualstudio.com/api/references/contribution-points

- **Interfacce utente personalizzate** per creare il pannello di chat all'interno dell'IDE.

4.2 Strumenti e risorse utilizzate nello sviluppo

- **Visual Studio Code** come ambiente di sviluppo.
- **TypeScript** come linguaggio di programmazione per l'intera architettura dell'estensione.
- **JavaScript**, **HTML** e **CSS** per l'implementazione dell'interfaccia utente del pannello di chat tramite Webview API (si veda Sezione 4.9).
- **Node.js** come runtime environment per TypeScript e JavaScript.
- **Git** per il controllo versione.

Per la creazione della struttura iniziale del progetto è stato utilizzato lo scaffolding tool Yeoman[2], che consente di generare uno scheletro per un'estensione VS Code già pre-configurata con TypeScript e i vari strumenti di build necessari.

Infine, oltre alla documentazione ufficiale fornita da VS Code, una risorsa fondamentale impiegata per la comprensione di pattern e l'implementazione di soluzioni comuni è stato il repository "vscode-extension-samples"[3] messo a disposizione da Microsoft su GitHub e contenente svariati esempi di codice.

4.3 ServiceContainer: DI container

Come si può vedere dal codice riportato nel Listato 4.1, `ServiceContainerImpl` fornisce le implementazioni per i metodi `register` e `resolve` definiti dall'interfaccia `ServiceContainer`, utilizzati rispettivamente per la registrazione e la risoluzione di un determinato servizio.
Il metodo `register` associa l'identificatore univoco di un servizio al relativo factory method necessario per la sua creazione, salvando tale informazione all'in-

[2] yeoman.io
[3] github.com/microsoft/vscode-extension-samples

terno di una mappa interna (riga 2) e delegandone la gestione ad un'istanza di `ServiceDescriptor` (riga 8).
Quando, in seguito alla registrazione di un servizio, viene chiamato il metodo `resolve`, il container utilizza la key fornita per individuare il `ServiceDescriptor` corrispondente (righe 14-16) e richiederne l'istanza tramite il metodo `getInstance` (riga 21).

```
1  export class ServiceContainerImpl implements ServiceContainer {
2      private services = new Map<string, ServiceDescriptor>();
3
4      public register<K extends keyof ServiceRegistry>(
5          key: K,
6          provider: (container: ServiceContainer) => ServiceRegistry[K]
7      ): void {
8          this.services.set(key as string, new ServiceDescriptor(provider, this));
9      }
10
11     public resolve<K extends keyof ServiceRegistry>(
12         key: K
13     ): ServiceRegistry[K] {
14         const descriptor = this.services.get(
15             key as string
16         ) as ServiceDescriptor<ServiceRegistry[K]>;
17         if (!descriptor) {
18             throw new Error(`Service ${String(key)} not registered`);
19         }
20
21         return descriptor.getInstance();
22     }
23 }
```

Listato 4.1: Implementazione di `ServiceContainer`.

La classe `ServiceDescriptor`, riportata nel Listato 4.2, incapsula la logica di creazione e la gestione dell'istanza di un singolo servizio.
In seguito alla registrazione di un servizio, `ServiceDescriptor` riceve tramite costruttore il factory method ad esso associato e l'istanza del container stesso. Quest'ultima è fondamentale per consentire la risoluzione di eventuali dipendenze interne al momento dell'effettiva istanziazione del servizio, realizzando così il

pattern di Dependency Injection.

Il metodo pubblico `getInstance` applica il principio di lazy instantiation: nel momento in cui il servizio viene richiesto da `ServiceContainer` per la prima volta, viene eseguito il factory method ad esso associato (riga 11), memorizzandone l'output cosicché tutte le chiamate successive per il medesimo servizio restituiscano l'istanza precedentemente creata (riga 13), implementando di fatto il pattern Singleton.

```
1  class ServiceDescriptor<T extends Service = Service> {
2      private instance?: T;
3
4      constructor(
5          private readonly provider: (container: ServiceContainer) => T,
6          private readonly container: ServiceContainer
7      ) {}
8
9      public getInstance(): T {
10         if (!this.instance) {
11             this.instance = this.provider(this.container);
12         }
13         return this.instance;
14     }
15 }
```

Listato 4.2: Implementazione di `ServiceDescriptor`.

4.3.1 Garantire type safety nella risoluzione dei servizi

Nell'implementazione del DI container è fondamentale garantire la coerenza dei tipi tra la chiave utilizzata per la registrazione di un servizio e l'oggetto restituito dalla sua risoluzione. Per farlo sono state utilizzate diverse funzionalità avanzate di TypeScript che hanno permesso di ottenere una soluzione *type-safe* in grado di individuare tali errori a compile-time.

Innanzitutto è stata creata una semplice interfaccia TypeScript `ServiceRegistry` (Listato 4.3) che stabilisce una relazione tra la stringa letterale associata ad un servizio, cioè la sua chiave (es. la *stringa* "AIService"), e il relativo tipo dell'istanza ad esso associata (es. l'*interfaccia* `AIService`).

```
export type ServiceRegistry = {
    "ConfigService": ConfigService;
    "AIService": AIService;
    // other services
}
```

Listato 4.3: `ServiceRegistry`.

Così facendo, il metodo `register<K extends keyof ServiceRegistry>` può essere comodamente definito utilizzando l'operatore TypeScript `keyof`[4], vincolando `K` ad assumere il valore di una delle chiavi dichiarate in `ServiceRegistry`.
Come si può vedere nell'esempio riportato nel Listato 4.4, questo approccio impedisce l'utilizzo di chiavi arbitrarie o errate, sollevando in tali casi un errore a compile-time.

```
container.resolve("ConfigService"); // correct
container.resolve("somethingElse"); // compile error: Argument of type
↪  '"somethingElse"' is not assignable to parameter of type 'keyof
↪  ServiceRegistry'.ts(2345)
```

Listato 4.4: Cercare di registrare o risolvere un servizio tramite una chiave non presente in `ServiceDescriptor` risulta in un errore a compile-time.

Anche nel caso del metodo `resolve<K extends keyof ServiceRegistry>` viene applicato il medesimo principio. In aggiunta, poiché il tipo di ritorno del metodo è `ServiceRegistry[K]`, TypeScript collega dinamicamente la chiave `K` al relativo tipo ad essa associato all'interno di `ServiceRegistry`.
Ciò significa che se, ad esempio, `resolve` riceve come parametro `key` la stringa "ChatService", allora `ServiceRegistry[K]` verrà automaticamente risolto ad un'istanza di tipo `ChatService`.

Definita l'architettura generale è possibile procedere con l'analisi dei singoli servizi che compongono l'estensione.
Ciascun servizio è stato pensato per svolgere un compito specifico e può comunicare

[4] typescriptlang.org/docs/handbook/2/keyof-types

con altri servizi ricevendoli in maniera esplicita dal costruttore o, qualora ciò non fosse possibile, risolvendoli tramite il `ServiceContainer`.

4.4 ConfigService: lettura della configurazione utente

Tutti i parametri configurabili dell'assistente devono essere dichiarati all'interno del file manifest dell'estensione. Alcuni di questi parametri (la cui assenza impedirebbe il funzionamento dell'estensione) sono obbligatori, come la chiave API e il modello da utilizzare. Dopodiché ci sono una serie di parametri facoltativi che consentono all'utente di modificare la configurazione del modello, cambiandone ad esempio la temperatura o il numero massimo di output tokens (che influenzano rispettivamente la "creatività" e la lunghezza delle risposte fornite dall'assistente) o abilitare/disabilitare funzionalità specifiche, come l'inline completion.

Nel contesto dell'ecosistema VS Code, tali informazioni vengono memorizzate all'interno di un file di configurazione specifico, `settings.json`[5], al quale l'estensione deve accedere al momento della propria attivazione. Per farlo l'API dell'IDE mette a disposizione un apposito oggetto `WorkspaceConfiguration`[6], che presenta un metodo `get` per ottenere il valore associato ad uno specifico parametro.

Tale metodo si rivela però problematico per la gestione dei parametri opzionali legati alla configurazione dei modelli AI: se un parametro non viene definito dall'utente, `get` restituirà il valore di default del suo tipo di dato (ad esempio 0 per un numero). Ciò chiaramente non è ideale, dato che in tali casi sarebbe preferibile lasciare che il provider AI possa utilizzare il proprio valore di default per tale parametro.

Per risolvere questa problematica è stato implementato un metodo privato `getConfigValue` che consente di ottenere il valore associato ad un dato parametro solamente se questo è stato esplicitamente definito dall'utente.
Come mostrato nel Listato 4.5, dopo aver ottenuto l'oggetto di configurazione

[5] code.visualstudio.com/docs/configure/settings
[6] code.visualstudio.com/api/references/vscode-api#WorkspaceConfiguration

`WorkspaceConfiguration` (riga 4) viene chiamato il suo metodo `inspect`, che consente di determinare se un dato parametro sia stato esplicitamente definito dall'utente (righe 7-10): in caso affermativo viene chiamato il metodo `get` per ottenerne il valore, altrimenti viene restituito `undefined` (riga 11).

```
1  export class ConfigServiceImpl implements ConfigService {
2      ...
3      private getConfigValue<T>(key: string): T | undefined {
4          const config = vscode.workspace.getConfiguration(Constants.EXTENSION_ID);
5          const inspection = config.inspect(key);
6
7          const isSet =
8              inspection?.globalValue !== undefined ||
9              inspection?.workspaceValue !== undefined ||
10             inspection?.workspaceFolderValue !== undefined;
11         return isSet ? config.get<T>(key) : undefined;
12     }
13 }
```

Listato 4.5: Implementazione del metodo privato `getConfigValue`, responsabile dell'adeguata risoluzione dei parametri opzionali.

Il metodo `isConfigValid`, la cui implementazione è omessa per brevità, viene chiamato da `ExtensionManager` all'avvio dell'estensione per controllare che l'utente abbia definito tutti i parametri obbligatori, mostrando opportuni messaggi di errore in caso contrario. Validata la configurazione viene chiamato `getConfig` (riportato nel Listato 4.6), che restituisce l'intera configurazione utente in maniera strutturata come un oggetto di tipo `ExtensionConfiguration` (righe 1-6).
Per i parametri obbligatori, la cui presenza è garantita in seguito alla validazione iniziale, viene invocato direttamente il metodo `WorkspaceConfiguration.get` (riga 17), utilizzando l'operatore di asserzione non-null (`!`) per indicare a TypeScript che la presenza del valore associato a tali parametri è garantita.
Per i parametri opzionali viene invece utilizzato il metodo `getConfigValue` (riga 20), cosicché il provider AI possa utilizzare i propri valori di default per i parametri non esplicitamente definiti dall'utente.

```
1  export type ExtensionConfiguration = {
2      apiKey: string; // mandatory parameters
```

```
3       ...
4       temperature?: number; // optional parameters
5       ...
6   }
7
8   export class ConfigServiceImpl implements ConfigService {
9
10      public isConfigValid(): boolean {...}
11
12      public getConfig(): ExtensionConfiguration {
13          const config = vscode.workspace.getConfiguration(EXTENSION_ID);
14
15          return {
16              // required settings: if isConfigValid() is true, these will always be
                ↪  defined.
17              apiKey: config.get<string>("apiKey")!,
18              ...
19              // optional settings, undefined if not explicitly set by user
20              temperature: this.getConfigValue<number>("temperature"),
21              ...
22          };
23      }
24      ...
25  }
```

Listato 4.6: Implementazione del metodo `getConfig`, che restituisce la configurazione come un oggetto di tipo `ExtensionConfiguration`.

4.5 AIService: interazione con i modelli AI

In fase di implementazione è stato deciso di dare maggior flessibilità ai modelli AI utilizzabili dall'assistente, permettendo alle diverse funzionalità di utilizzare modelli differenti in base alle loro necessità. Ad esempio, mentre la funzionalità di chat beneficerebbe maggiormente dall'utilizzo di un agente più performante ma lento, per altre feature come l'inline completion, al fine di fornire i suggerimenti il più velocemente possibile, sarebbe preferibile utilizzare un agente meno performante ma più rapido.

Per farlo è stata apportata una lieve modifica all'architettura presentata in Sezio-

ne 3.2.2, facendo in modo che il modello dell'agente non venga letto dalla configurazione globale tramite `ConfigService`, ma bensì passato come argomento al metodo `createAgent` di `AIService`.

4.5.1 Implementazione concreta di un provider

Nonostante l'architettura sviluppata sia stata concepita per consentire l'integrazione di molteplici provider specifici, nel corso dello sviluppo ci si è limitati all'implementazione e utilizzo di un solo provider per i modelli offerti da Google.

Tale scelta è stata dettata da motivazioni prettamente pratiche: al momento dello sviluppo dell'assistente oggetto della presente tesi, Google si è rivelato l'unico provider a offrire un accesso gratuito alla propria API. Inoltre, sebbene tale accesso presenti chiaramente delle limitazioni, queste si sono dimostrate sufficientemente generose da non ostacolare il processo di sviluppo e testing.

Nel Listato 4.7 viene presentata l'implementazione di `GoogleAIProvider`, un provider AI concreto che si occupa dell'interazione con la Gemini API di Google[7]. Dal codice si può osservare che al momento della creazione di `GoogleAIProvider` (che chiama il costruttore definito da `AbstractAIProvider`) viene istanziato l'SDK di Google Generative AI (riga 6), necessario per l'interfacciamento diretto con l'API specifica per i modelli di Google.

```
1  export class GoogleAIProvider extends AbstractAIProvider<GoogleAIAgent> {
2      private ai?: GoogleGenAI;
3
4      constructor(apiKey: string) {
5          super(apiKey);
6          this.ai = new GoogleGenAI({ apiKey: this.apiKey });
7      }
8
9      public async generateContent(
10         agent: GoogleAIAgent,
11         content: string,
12         history?: ConversationElement[],
13         instructions?: string
14     ): Promise<string> {
```

[7] ai.google.dev/gemini-api

```
        if (!this.ai) {
            throw new Error("Google AI provider not initialized");
        }
        // convert provider-agnostic history to Google's format
        const contents: Content[] = this.convertHistory(history);
        // add the current user message
        contents.push({
            role: "user",
            parts: [{ text: content }],
        });

        const googleAgent = agent;
        const response = await this.ai.models.generateContent({
            model: agent.model,
            contents: contents,
            config: googleAgent.getConfigWithInstructions(instructions),
        });

        let responseText = response.text || "";
        return this.isSearchGroundingEnabled(agent)
            ? this.appendGroundingMetadata(responseText, response)
            : responseText;
    }
    ...
```

Listato 4.7: Implementazione del provider AI di Google.

Conversione della cronologia di conversazione

Poiché `AIService` fornisce l'eventuale cronologia della conversazione (utilizzata nella feature di chat) in un formato "agnostico" rispetto ai provider (si veda Sezione 3.2.2), è stato necessario effettuare una conversione interna nella specifica struttura dati richiesta dall'API di Google, cioè `Content[]`[8]. Il metodo privato responsabile di tale conversione è `convertHistory`, la cui implementazione è riportata nel Listato 4.8.

```
private convertHistory(history?: ConversationElement[]): Content[] {
    const contents: Content[] = [];

    if (history && history.length > 0) {
```

[8] ai.google.dev/api/caching/#Content

```
        for (const msg of history) {
            contents.push({
                role: msg.role === "assistant" ? "model" : msg.role,
                parts: [{ text: msg.content }],
            });
        }
    }

    return contents;
}
```

Listato 4.8: Conversione dal formato di cronologia messaggi generico `ConversationElement[]`, utilizzato da `AIService`, a `Content[]`, utilizzato dalla Gemini API.

Dopo la conversione dell'eventuale cronologia della conversazione vi viene aggiunto l'ultimo messaggio inviato dall'utente (che sarà l'unico messaggio presente in caso la cronologia fosse vuota). Infine la risposta dell'assistente viene generata effettuando una chiamata asincrona all'API di Google, utilizzando il modello e i parametri specificati nella configurazione dell'agente.

Funzionalità di grounding

Poiché la Gemini API consente di utilizzare la funzionalità di grounding[9], in `GoogleAIProvider` sono state fornite le implementazioni per i metodi `toggleSearchGrounding` e `isSearchGroundingEnabled` (che si limitano a modificare una flag interna all'agente `isGroundingEnabled`) per fornire l'accesso a tale funzionalità.

Per mostrare le informazioni sulle fonti utilizzate qualora un agente con il grounding attivo effettui una ricerca web, è stato creato un metodo privato `appendGroundingMetadata` (la cui implementazione è riportata nel Listato 4.9).

```
private appendGroundingMetadata(
    responseText: string,
    response: GenerateContentResponse
): string {
    const groundingContent =
```

[9] ai.google.dev/gemini-api/docs/google-search

```
6           response.candidates?.[0]?.groundingMetadata?.searchEntryPoint?.renderedConte
            ↪ nt;
7
8       if (!groundingContent) {
9           return responseText;
10      }
11
12      return `${responseText}\n\n---\n\n**Search Results
        ↪ Used:**\n\n${groundingContent}`;
13  }
```

Listato 4.9: Implementazione di `appendGroundingMetadata`. In seguito ad una ricerca web da parte dell'agente la risposta include un campo `groundingMetadata` contenente le query di ricerca utilizzate, che vengono appese alla risposta fornita.

4.6 Servizi secondari

Oltre ai servizi che gestiscono le funzionalità principali dell'estensione, ovvero `ConfigService`, `AIService` e `ChatService` (i cui dettagli verranno presentati in Sezione 4.9), sono stati introdotti dei servizi che svolgono una funzione di supporto. Questi servizi secondari offrono principalmente un punto di accesso centralizzato a funzionalità comuni, semplificando la logica dei vari componenti che ne fanno uso e favorendo la modularità dell'intera architettura.

4.6.1 FileService

Essendoci vari componenti dell'estensione che svolgono operazioni di lettura o scrittura su file, è stato creato un servizio apposito `FileService` che consente agli utilizzatori di effettuare tali operazioni senza doversi interfacciare con l'API di VS Code e i relativi dettagli implementativi.

In particolare, `FileService` espone dei metodi per leggere e modificare l'intero contenuto di un file (o solo il testo attualmente selezionato), funzionalità particolarmente utilizzate dai comandi legati all'interazione contestuale (descritti in Sezione 4.7). Presenta inoltre dei semplici metodi di utility per estrarre metadati come il nome o l'estensione di un file a partire dal suo percorso (*path*).
Un'ultima funzionalità esposta dal servizio, tramite il metodo `getUserDefined`

`Instructions`, è dedicata al recupero delle istruzioni personalizzate che l'utente può opzionalmente definire all'interno del file `.ai/agentInstructions.md` per modificare il comportamento dell'assistente.

4.6.2 RateLimitService

L'estensione, al fine di interagire con i modelli AI, utilizza le API fornite da provider esterni, che possono essere soggette a costi e limiti di utilizzo. Per questo motivo è desiderabile un meccanismo in grado di limitare la frequenza di esecuzione delle feature che ne fanno uso (*rate limiting*).

A tale scopo è stato creato il servizio `RateLimitService`, che espone due metodi:

- `configureRateLimit`, che consente di registrare una feature (identificata da una chiave univoca) e associarvi l'intervallo di attesa minimo che deve trascorrere tra due invocazioni consecutive.
- `isRequestAllowed`, per verificare se una data feature possa o meno effettuare una nuova richiesta.

Come si può vedere dal codice riportato nel Listato 4.10, l'implementazione del servizio mantiene internamente una mappa (riga 7) che associa ciascuna feature registrata ad un `RateLimitEntry`, contenente l'intervallo minimo di tempo tra due richieste successive (`timeoutMs`) e il timestamp dell'ultima richiesta andata a buon fine (`lastRequestTime`) (righe 1-4).
Chiaramente l'approccio adottato lascia al chiamante il compito di verificare se una richiesta possa essere effettuata o meno prima di procedere, chiamando il metodo `isRequestAllowed`.

```
1 type RateLimitEntry = {
2     timeoutMs: number;
3     lastRequestTime: number;
4 };
5
6 export class RateLimitServiceImpl implements RateLimitService {
7     private rateLimits = new Map<string, RateLimitEntry>();
8
9     public configureRateLimit(key: string, timeBetweenRequests: number): void {
```

```
10          this.rateLimits.set(key, {
11              timeoutMs: timeBetweenRequests,
12              lastRequestTime: 0,
13          });
14      }
15
16      public isRequestAllowed(key: string): boolean {
17          const entry = this.getEntryOrThrow(key);
18          const now = Date.now();
19
20          // check if enough time has passed since the last request
21          const timeSinceLastRequest = now - entry.lastRequestTime;
22          if (timeSinceLastRequest < entry.timeoutMs) {
23              const waitTime = entry.timeoutMs - timeSinceLastRequest;
24              return false;
25          }
26
27          // allow request and update last request time
28          entry.lastRequestTime = now;
29          return true;
30      }
31
32      private getEntryOrThrow(key: string): RateLimitEntry {
33          const entry = this.rateLimits.get(key);
34          if (!entry) {
35              throw new Error(`No configuration found for '${key}'`);
36          }
37          return entry;
38      }
39  }
```

Listato 4.10: Implementazione di RateLimitService.

Nel contesto dell'estensione realizzata, `RateLimitService` è utilizzato solo dalla feature di inline completion (approfondita in Sezione 4.8), essendo questa l'unica in grado di effettuare delle chiamate al provider AI senza richieste esplicite da parte dell'utente. Tuttavia, il servizio è stato volutamente progettato in maniera scalabile, per poter estendere con facilità il meccanismo di rate limiting a qualsiasi funzionalità futura che ne possa avere bisogno.

4.6.3 LoggingService

Per supportare le attività di diagnostica e debugging è stato introdotto un semplice servizio `LoggingService` che funge da wrapper attorno alle funzionalità di logging fornite dall'API di VS Code, esponendo ai vari componenti dell'estensione i metodi per i principali livelli di logging: `debug`, `info`, `warn` e `error`.

L'implementazione di `LoggingService` mantiene internamente un riferimento all'oggetto `LogOutputChannel`[10], sul quale indirizza tutti i messaggi ricevuti per far sì che vengano mostrati su un apposito pannello di output all'interno dell'IDE.

4.7 Interazione contestuale

VS Code ha a disposizione svariati comandi "built-in" per interagire con l'editor, eseguire azioni in background e gestire l'interfaccia utente.
Oltre a poter liberamente invocare ciascuno di questi comandi nativi, un'estensione può anche definire e registrare dei comandi aggiuntivi, al fine di esporre le proprie funzionalità agli utenti o implementare della logica interna[11].

Sfruttando questo meccanismo di estendibilità è stato possibile registrare i comandi di interazione contestuale, ovvero quelli che consentono di interagire con l'assistente all'interno dell'editor. In seguito alla loro registrazione con l'API di VS Code, trattata in Sezione 4.7.3, l'utente può invocarli come se fossero comandi nativi, utilizzando l'apposita palette dei comandi (come mostrato in Figura 4.1) o tramite scorciatoie da tastiera (*keyboard shortcuts*).

I comandi di interazione contestuale implementati sono:

1. `EditCommand`, che consente di modificare il file corrente sulla base delle istruzioni fornite dall'utente in linguaggio naturale.
2. `RefactorCommand`, che esegue automaticamente il refactoring del codice, senza che l'utente debba fornire alcuna indicazione.
3. `DocsCommand`, che genera automaticamente la documentazione per il codice.

[10] code.visualstudio.com/api/references/vscode-api#LogOutputChannel
[11] code.visualstudio.com/api/extension-guides/command

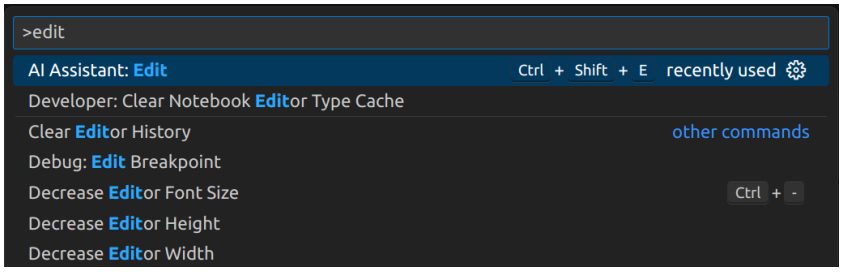

Figura 4.1: Palette dei comandi (*command palette*) di VS Code, dalla quale è possibile accedere sia ai comandi nativi dell'IDE che a quelli registrati dalle varie estensioni.

Occorre far presente che in fase di implementazione è stata effettuata una leggera modifica all'architettura logica presentata in Sezione 3.3.1 durante la fase di design, aggiungendo all'interfaccia `Command` un metodo `getId` necessario per l'effettiva registrazione con l'API di VS Code.

4.7.1 Implementazione di BaseEditorTransformer

`BaseEditorTransformer`, riportato nel Listato 4.11, effettua l'override dei metodi `canExecute` (che ritorna `True` solamente se l'utente ha un editor di testo attivo) e `execute`, che contiene l'effettiva chiamata al template method `applyTransformation` (riga 15). Quest'ultimo metodo viene ereditato dall'interfaccia `EditorTransformer` e il compito di fornirne l'implementazione viene delegato alle sottoclassi.

```
1  export abstract class BaseEditorTransformer
2      extends BaseCommand
3      implements EditorTransformer
4  {
5      public canExecute(): boolean {
6          return this.serviceContainer
7              .resolve("FileService")
8              .hasActiveTextEditor();
9      }
10
11     public async execute(): Promise<void> {
12         if (!this.canExecute()) {
13             return;
14         }
15         await this.applyTransformation();
```

```
16      }
17
18      ...
```

Listato 4.11: Implementazione dei metodi `canExecute` e `execute` di `BaseEditorTransformer`.

La logica comune a tutte le sottoclassi è contenuta nel metodo protetto `aiCodeProcess`. Il suo flusso di esecuzione, riportato nel Listato 4.12, è suddiviso come segue:

1. **Estrazione del codice**: in seguito alla risoluzione delle dipendenze utilizzando il service container ereditato da `BaseCommand` viene verificato tramite il metodo `getEditorSelection` di `FileService` se l'utente abbia selezionato o meno del codice (riga 8). In caso affermativo, il comando opererà esclusivamente sulla selezione, altrimenti verrà processato l'intero contenuto dell'editor.

2. **Configurazione dell'agente**: viene istanziato l'agente da utilizzare (righe 12-14) e assemblato il relativo prompt, unendo eventuali istruzioni dell'utente con il codice da elaborare. Le istruzioni di sistema, ovvero quelle necessarie a definire il comportamento dell'agente, vengono similmente arricchite con le eventuali preferenze globali dell'utente, ottenute dal file di configurazione `.ai/agentInstructions.md` tramite `FileService` (riga 21).

3. **Generazione del contenuto**: mentre l'agente genera il contenuto viene utilizzato il metodo `vscode.window.withProgress` per mostrare il messaggio di progresso `progressText` (righe 26-31). Al termine della generazione, dato che gli agenti tendono ad utilizzare il formato Markdown nelle loro risposte, il risultato ottenuto viene ripulito da eventuali marcatori di blocco di codice tramite un metodo di utility (riga 39), per far sì che il codice inserito nell'editor sia "pulito".

4. **Sostituzione e Conferma**: il codice originale viene sostituito con quello generato, rispettando il contesto iniziale (selezione o intero editor). Infine viene mostrato un messaggio di successo dove l'utente può scegliere se man-

tenere le modifiche o annullarle, invocando il comando nativo di VS Code `undo` (righe 50-57).

```
protected async aiCodeProcess(
        systemInstructions: string,
        userPrompt: string | undefined,
        progressText: string,
        successMessage: string
    ): Promise<void> {
        ... // services resolution via serviceContainer
        const selectedCode = fileService.getEditorSelection();
        const codeToTransform = selectedCode || fileService.getEditorContent()!;
        const isFullFile = !selectedCode;

        const agent = aiService.createAgent(
            configService.getConfig().baseModel
        );
        // build the prompt with optional user request and code
        const prompt = userPrompt
            ? `User request: ${userPrompt}\n\nCode:\n${codeToTransform}`
            : codeToTransform;
        // combine system instructions with user-defined instructions if available
        const userInstructions =
            fileService.getUserDefinedInstructions();
        if (userInstructions) {
            systemInstructions += `\n\nUser preferences:\n${userInstructions}`;
        }

        const processedCode = await vscode.window.withProgress(
            {
                location: vscode.ProgressLocation.Notification,
                title: progressText,
                cancellable: false,
            },
            async () => {
                const transformedCode = await aiService.generateContent(
                    agent,
                    prompt,
                    undefined, // no conversation history
                    systemInstructions
                );
                return Utils.removeCodeBlockMarkers(transformedCode);
            }
```

```
41          );
42
43          if (isFullFile) {
44              await fileService.replaceFileContent(processedCode);
45          } else {
46              await fileService.replaceSelectedText(processedCode);
47          }
48
49          // ask user to keep or undo the changes
50          const result = await vscode.window.showInformationMessage(
51              successMessage,
52              "Keep Changes",
53              "Undo"
54          );
55          if (result === "Undo") {
56              await vscode.commands.executeCommand("undo");
57          }
58      }
```

Listato 4.12: Implementazione del metodo `aiCodeProcess` all'interno di `BaseEditorTransformer`.

Nota: L'implementazione effettiva di `aiCodeProcess` presente nella codebase delega le varie responsabilità appena descritte a metodi privati dedicati: il codice sopra riportato è stato compattato per brevità espositiva.

4.7.2 Implementazione concreta di un comando di interazione contestuale

Grazie all'architettura adottata, l'implementazione concreta dei comandi di interazione contestuale diventa triviale, poiché è sufficiente fornire un'implementazione al template method `applyTransformation` e richiamare al suo interno il metodo protetto `aiCodeProcess` ereditato da `BaseEditorTransformer`.

Come mostrato nel Listato 4.13, all'interno di `EditCommand` l'implementazione del template method si limita a richiedere all'utente le modifiche desiderate tramite un input box (righe 10-14) e ad invocare `aiCodeProcess` con il prompt acquisito e delle appropriate istruzioni di sistema (righe 19-24).

```
export class EditCommand extends BaseEditorTransformer {
    private readonly id = "edit";
    private readonly userInstructions =
        "Describe what edits you want to make. Select some text to edit a specific
        ↪  section only, or leave unselected to edit the entire file.";
    private readonly systemInstructions =
        "You are a code editing assistant. The user will provide code and a
        ↪  description of the edits they want. Apply the requested edits to the
        ↪  code. Return the edited code ONLY, without any additional explanations.";

    public async applyTransformation(): Promise<void> {
        // show input box to get user prompt
        const userPrompt = await vscode.window.showInputBox({
            placeHolder: "Enter your prompt...",
            prompt: this.userInstructions,
            ignoreFocusOut: true,
        });
        if (!userPrompt) {
            return; // user cancelled
        }

        await this.aiCodeProcess(
            this.systemInstructions,
            userPrompt,
            "Editing...",
            "Code Edited!"
        );
    }

    public getId(): string {
        return this.id;
    }
}
```

Listato 4.13: Implementazione di `EditCommand`.

Le implementazioni di `RefactorCommand` e `DocsCommand`, omesse per brevità, poiché non richiedono alcun input utente sono ancora più semplici e si limitano ad invocare direttamente `aiCodeProcess`, fornendogli delle appropriate istruzioni di sistema.

4.7.3 Registrazione dei comandi

La registrazione dei comandi viene effettuata all'avvio dell'estensione dal metodo privato `registerCommands` di `ExtensionManagerImpl`.
Come mostrato nell'implementazione riportata nel Listato 4.14, tale processo è standardizzato per tutti i comandi e reso facilmente estendibile: innanzitutto viene creato un array contenente i comandi da registrare (righe 2-7), fornendo a ciascuno di essi il service container per la risoluzione delle dipendenze a runtime.

Per ogni comando da registrare viene poi invocato il metodo `vscode.commands.registerCommand`[12] dell'API di VS Code (riga 10), che prende come argomenti l'identificatore univoco del comando (ottenuto tramite `getId`) e una funzione di callback. Quest'ultima è stata definita in modo tale da consentire l'esecuzione di ciascun comando solo dopo la verifica dei prerequisiti necessari tramite `canExecute` (righe 12-16).

Infine l'oggetto `Disposable`[13] restituito dalla registrazione viene aggiunto al contesto dell'estensione (riga 17): si tratta di una best practice dello sviluppo per VS Code necessaria a garantire che i comandi vengano correttamente de-registrati alla disattivazione dell'estensione, rilasciando eventuali risorse ancora in uso ed evitando possibili memory leak.

```
1  private registerCommands(): void {
2      const commands = [
3          new RefactorCommand(this.serviceContainer),
4          new EditCommand(this.serviceContainer),
5          new DocsCommand(this.serviceContainer),
6          new NewChatCommand(this.serviceContainer),
7      ];
8
9      commands.forEach((command) => {
10         const disposable = vscode.commands.registerCommand(
11             `${Constants.EXTENSION_ID}.${command.getId()}`,
12             async () => {
13                 if (command.canExecute()) {
14                     await command.execute();
15                 }
```

[12] code.visualstudio.com/api/extension-guides/command#registering-a-command
[13] code.visualstudio.com/api/references/vscode-api#disposables

```
16              }
17          );
18          this.context.subscriptions.push(disposable);
19      });
20  }
```

Listato 4.14: Implementazione del metodo `registerCommands` all'interno di `ExtensionManagerImpl`. `NewChatCommand`, non trattato in dettaglio, è un semplice comando che consente di creare una nuova chat.

4.8 Inline completion

L'implementazione della feature di inline completion, che consente all'utente di ricevere completamenti di codice all'interno dell'editor, dipende totalmente dall'API messa a disposizione dallo specifico IDE nel quale l'assistente AI viene integrato.

Nel contesto di VS Code è stata adottata l'architettura riportata in Figura 4.2, che fa uso di un provider concreto `InlineCompletionProvider` che estende un'apposita interfaccia `InlineCompletionItemProvider`[14] fornita dall'API dell'IDE. Quest'ultima definisce il metodo `provideInlineCompletionItems`, che viene utilizzato dall'ambiente per richiedere i suggerimenti da fornire all'utente ogni volta che interrompe la digitazione o richiede un completamento in maniera esplicita tramite un apposito comando nativo.

4.8.1 Attivazione e parametri di configurazione

Essendo l'inline completion in grado di effettuare delle chiamate al provider AI senza esplicita richiesta da parte dell'utente è stata data la possibilità di disattivare la feature assegnando il valore `false` al parametro `enableInlineCompletions`, contenuto nel file di configurazione `.vscode/settings.json`.
Inoltre, nel caso in cui la feature venisse mantenuta attiva, sono stati dichiarati ulteriori parametri configurabili che consentono all'utente di limitare il numero di chiamate effettuate verso il provider AI e ridurre i costi d'uso. Questi sono:

[14] code.visualstudio.com/api/references/vscode-api#InlineCompletionItemProvider

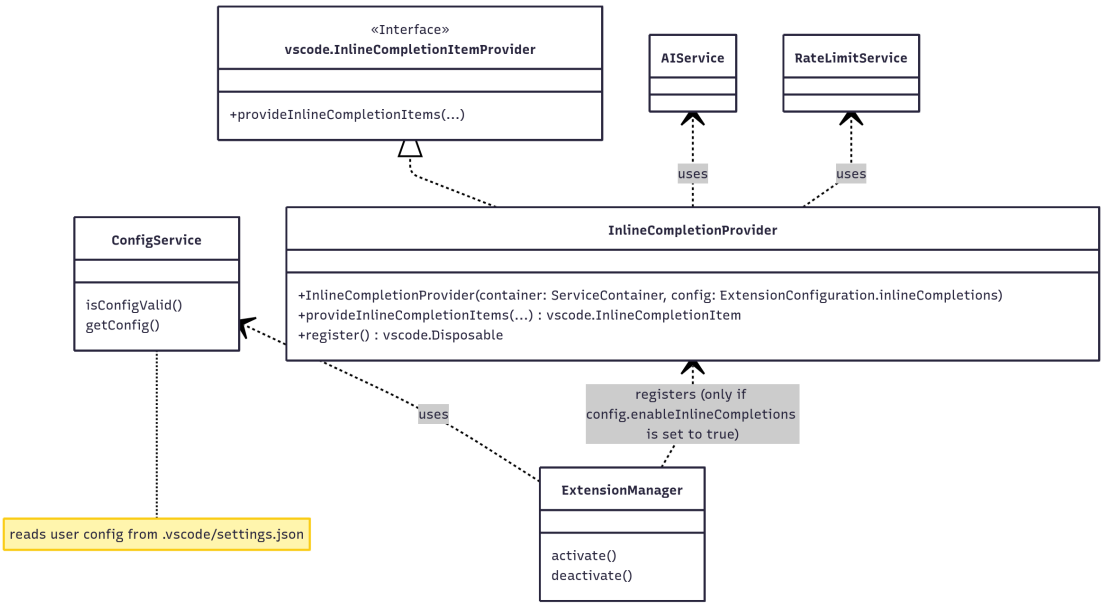

Figura 4.2: Architettura completa di `InlineCompletionProvider`.

- `inlineCompletions.timeBetweenRequests`, che definisce l'intervallo minimo tra due richieste consecutive (utilizzato per il rate limiting).
- `inlineCompletions.idleDelay`, che specifica il tempo che deve trascorrere dal momento in cui l'utente interrompe la digitazione a quando viene effettuata una richiesta di completamento.
- `inlineCompletions.model`, che definisce il modello AI da utilizzare per le inline completions, consentendo all'utente di adottare un modello più veloce ed economico rispetto a quello configurato per le altre funzionalità dell'estensione.

Questi parametri vengono passati a `InlineCompletionProvider` tramite il costruttore e utilizzati all'interno del metodo `register` (riportato nel Listato 4.15), che si occupa della configurazione del rate limiting mediante `RateLimitService` (utilizzando il valore del parametro di configurazione `timeBetweenRequests`) e dell'effettiva registrazione del provider con l'API di VS Code.
Quest'ultima viene effettuata tramite il metodo `registerInlineCompletionItemProvider`, a cui vengono passati come argomenti un selettore generico (per far

sì che i suggerimenti siano forniti indipendentemente dal linguaggio di programmazione) e il provider stesso (righe 8-11).
Inoltre, analogamente a quanto visto nel caso della registrazione dei comandi per l'interazione contestuale, anche `register` restituisce un `Disposable` (righe 13-15), al fine di de-registrare il provider in seguito alla disattivazione dell'estensione.

```
1  public register(): vscode.Disposable {
2      this.rateLimitService.configureRateLimit(
3          SERVICE_KEY,
4          this.config.timeBetweenRequests
5      );
6
7      const providerRegistration =
8          vscode.languages.registerInlineCompletionItemProvider(
9              { pattern: "**" },
10             this
11         );
12
13     return new vscode.Disposable(() => {
14         providerRegistration.dispose();
15     });
16 }
```

Listato 4.15: Implementazione del metodo `register` di `InlineCompletionProvider`.

Come mostrato nel Listato 4.16, in seguito all'avvio dell'estensione l'attivazione della feature di inline completion è a carico di `ExtensionManagerImpl`. Questo controlla tramite `ConfigService` se la proprietà `enableInlineCompletions` sia impostata a `true`, procedendo solo in tal caso all'inizializzazione e registrazione del provider (righe 2-8).

```
1  if (config.enableInlineCompletions) {
2      this.inlineCompletionProvider = new InlineCompletionProvider(
3          this.serviceContainer,
4          config.inlineCompletions
5      );
6      const inlineCompletionDisposable =
7          this.inlineCompletionProvider.register();
8      this.context.subscriptions.push(inlineCompletionDisposable);
9  } else {
```

```
10      logger.info("Skipping inline completion provider activation");
11  }
```

Listato 4.16: Logica per l'attivazione delle inline completions in `ExtensionManagerImpl`.

4.8.2 Meccanismo di richiesta dei completamenti

In seguito alla registrazione di `InlineCompletionProvider`, VS Code invoca il suo metodo `provideInlineCompletionItems` ogni volta che vengono richiesti dei completamenti.

Come mostrato nell'implementazione riportata nel Listato 4.17, il comportamento di tale metodo varia a seconda del tipo di *trigger* (innesco) ricevuto: se il suggerimento è stato richiesto esplicitamente dall'utente (riga 9) allora viene immediatamente chiamato il metodo privato `getCompletions`, responsabile della generazione del completamento.

Nel caso di un trigger automatico invece, prima di invocare `getCompletions` viene atteso un tempo variabile dettato dal parametro di configurazione `idleDelay`: se in tale lasso di tempo l'utente riprende a scrivere il valore di `token.isCancellationRequested` risulterà `True` e la richiesta non verrà effettuata (righe 20-23).

```
1   public provideInlineCompletionItems(
2       document: vscode.TextDocument,
3       position: vscode.Position,
4       context: vscode.InlineCompletionContext,
5       token: vscode.CancellationToken
6   ): vscode.ProviderResult<
7       vscode.InlineCompletionItem[] | vscode.InlineCompletionList
8   > {
9       // triggered explicitly by the user
10      if (context.triggerKind === vscode.InlineCompletionTriggerKind.Invoke) {
11          return this.getCompletions(document, position, token);
12      }
13      // triggered automatically
14      if (this.idleTimer) {
15          clearTimeout(this.idleTimer);
16      }
17      return new Promise((resolve) => {
18          this.idleTimer = setTimeout(async () => {
```

```
            if (token.isCancellationRequested) {
                resolve([]);
                return;
            }

            const result = await this.getCompletions(
                document,
                position,
                token
            );
            resolve(result);
        }, this.config.idleDelay);
    });
}
```

Listato 4.17: Implementazione del metodo `provideInlineCompletionItems`.

4.8.3 Generazione dei completamenti

Innanzitutto, per ogni chiamata ricevuta, `getCompletions` verifica tramite il metodo `isRequestAllowed` di `RateLimitService` che sia trascorso un tempo sufficiente dall'ultimo completamento fornito, indipendentemente dal tipo di trigger. Se la richiesta è consentita, viene costruito il prompt da fornire all'agente responsabile della generazione del completamento, che verrà effettuata tramite `AIService`. La creazione di tale prompt è a carico del metodo privato `createCompletionPrompt`, riportato nel Listato 4.18, nel quale vengono specificate sia le istruzioni per l'agente che il contesto per la generazione, fornendo il linguaggio del file e il testo presente prima e dopo il cursore.

```
private createCompletionPrompt(
    fileLanguage: string,
    prefix: string,
    suffix: string
) {
    return [
        `You are given a ${fileLanguage} file, and your job is to suggest code
        ↪  completion to put exactly at the current cursor position.`,
        `Return ONLY the code to insert at the cursor, without any additional text
        ↪  or Markdown formatting.`,
```

```
9           `If the code is already complete or there isn't enough context, return
            ↪  nothing.\n`,
10          `The code before the cursor is:\n${prefix}`,
11          `${suffix.trim().length === 0
12                  ? "There is no code after the cursor"
13                  : `The code after the cursor is:\n${suffix}`
14          }`,
15      ].join("\n");
16  }
```

Listato 4.18: Implementazione del metodo privato `createCompletionPrompt` di `InlineCompletionProvider`, nel quale viene costruito il prompt che verrà poi utilizzato dall'agente responsabile della generazione del completamento.

4.9 ChatService: Pannello di chat

Definite le implementazioni per i servizi responsabili della lettura della configurazione utente, l'interazione con i modelli AI e la gestione delle operazioni su file, è possibile procedere con l'implementazione della funzionalità di chat.
Per realizzare il pannello di chat è stata utilizzata la Webview API[15], una tecnologia necessaria per superare i limiti dei componenti nativi di VS Code che consente di creare interfacce utente complesse tramite l'utilizzo di tecnologie web standard come HTML, CSS e JavaScript.

Una conseguenza diretta dovuta all'implementazione del pannello di chat (a cui d'ora in poi ci si riferirà con il termine *frontend*) tramite Webview, è che questa esiste come un elemento completamente separato rispetto al resto dell'architettura dell'estensione (d'ora in poi indicata come *backend*).
Come indicato anche dalla documentazione ufficiale, l'unico modo per consentire la comunicazione tra frontend e backend (e viceversa) è tramite *message passing*, cioè tramite lo scambio di messaggi.

Al fine di gestire questo scambio di informazioni in maniera centralizzata l'implementazione concreta di `ChatService` (che agisce da intermediario tra frontend e backend) espone i seguenti metodi:

[15] code.visualstudio.com/api/extension-guides/webview

- `setWebview(webview)`, necessario per stabilire la connessione con il frontend e instaurare la comunicazione bidirezionale necessaria per inviare e ricevere messaggi (si faccia riferimento alla Sezione 4.9.1 per ulteriori dettagli).
- `handleWebviewMessage(data)`, metodo verso il quale vengono indirizzati i vari messaggi provenienti dal frontend, come la ricezione di un messaggio utente, l'aggiunta di un file al contesto o l'attivazione/disattivazione del grounding. In base al tipo di messaggio ricevuto verrà poi invocata la logica appropriata.
- `clearChat()`, che consente di resettare la chat, cancellando la cronologia della conversazione e rimuovendo eventuali file aggiunti al contesto da parte dell'utente.

4.9.1 Inizializzazione del pannello di chat

La creazione di una Webview è un processo a più step che va effettuato tramite l'API di VS Code e consiste principalmente nel registrare il provider responsabile della creazione del relativo pannello (che verrà effettivamente istanziato e configurato solamente quando l'utente decide di aprirlo).

Per fare in modo che `ChatService` si possa occupare della sola logica applicativa è stata adottata un'architettura che consente di astrarre il processo di creazione e registrazione di una Webview.
Tale architettura, riportata in Figura 4.3, si basa su due componenti principali:

- **ChatWebviewManager**, che si occupa della *registrazione* del pannello di chat. Implementa l'interfaccia `WebviewManager`, che espone un singolo metodo `activate` necessario per la creazione di una Webview generica (il parametro `ExtensionContext`[16] contiene le varie informazioni necessarie per la registrazione).
- **ChatViewProvider**, che si occupa dell'*istanziazione* del pannello di chat. Implementa l'interfaccia `vscode.WebviewViewProvider`, che espone an-

[16] code.visualstudio.com/api/references/vscode-api#ExtensionContext

ch'essa un singolo metodo `resolveWebviewView`, in cui andranno definite le operazioni da svolgere quando l'utente apre il pannello di chat.

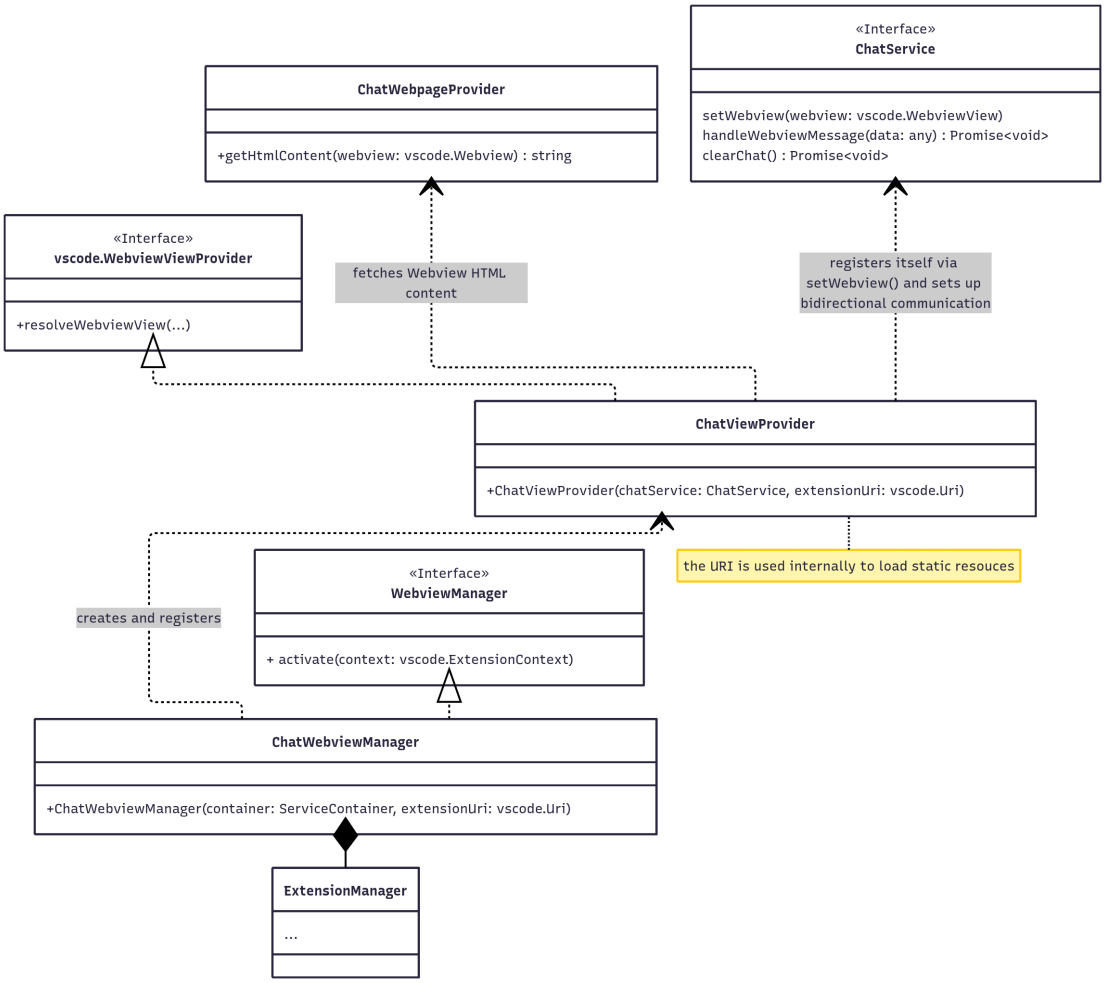

Figura 4.3: Architettura estesa adottata in fase di implementazione per creazione e gestione del pannello di chat, con inizializzazione della comunicazione frontend-backend.

Creazione della Webview

In seguito all'apertura da parte dell'utente del pannello di chat (cliccando sulla relativa icona nella barra laterale dell'IDE), VS Code invoca il metodo `resolveWebviewView` di `ChatViewProvider`, che si occupa di creare e configurare la Webview che verrà mostrata all'utente all'interno del pannello di chat (e che verrà utilizzata

da `ChatService` per la gestione dello scambio di messaggi tra frontend e backend). Come si può osservare dal codice riportato nel Listato 4.19, quando l'utente apre il pannello di chat, `resolveWebviewView` svolge tre operazioni fondamentali:

1. **Configurazione della Webview**: viene abilitata l'esecuzione di JavaScript (necessaria per l'invio di messaggi al backend) e viene impostato il root path della Webview a quello dell'estensione (riga 19), un accorgimento relativo alla sicurezza che ne impedisce l'accesso ai file esterni.

2. **Aggiunta del contenuto HTML**: il contenuto HTML dell'intera interfaccia utente del pannello di chat (presentato in maggior dettaglio in Sezione 4.9.2) viene ottenuto tramite il metodo `getHtmlContent` della classe `ChatWebpageProvider` e iniettato all'interno della Webview (riga 22).

3. **Registrazione con ChatService**: il metodo privato `setupMessageHandling` registra un listener per l'evento Webview `onDidReceiveMessage`[17] per far sì che i messaggi provenienti dalla Webview vengano inoltrati al metodo `handleWebviewMessage` di `ChatService`. Il canale di comunicazione tra frontend e backend viene infine completato passando la Webview come parametro al metodo `setWebview` di `ChatService` (riga 9).

```
1  // called when chat panel is opened
2  public resolveWebviewView(
3      webviewView: vscode.WebviewView,
4      _context: vscode.WebviewViewResolveContext,
5      _token: vscode.CancellationToken
6  ): void {
7      this.view = webviewView;
8      this.setupWebview();
9      this.chatService.setWebview(webviewView);
10 }
11
12 private setupWebview(): void {
13     if (!this.view) {
14         return;
15     }
16     const webview = this.view.webview;
17     webview.options = {
```

[17] code.visualstudio.com/api/references/vscode-api#Webview

```
18          enableScripts: true, // enable JS
19          localResourceRoots: [this.extensionUri], // needed to load local resources
20      };
21
22      webview.html = this.getHtmlContent();
23      this.setupMessageHandling(webview);
24  }
25
26  private setupMessageHandling(webview: vscode.Webview) {
27      this.messageHandlerDisposable?.dispose(); // dispose previous listener if any
28      this.messageHandlerDisposable = webview.onDidReceiveMessage(
29          async (data) => {
30              await this.chatService.handleWebviewMessage(data);
31          }
32      );
33  }
```

Listato 4.19: Implementazione del metodo `resolveWebviewView` di `ChatViewProvider`.

4.9.2 Contentuto HTML della Webview

La classe `ChatWebpageProvider` (la cui implementazione completa è omessa per brevità) ha come unico scopo di fornire, tramite il metodo `getHtmlContent`, il codice HTML responsabile per la creazione dell'interfaccia associata al pannello di chat.

Come si può vedere dalla Figura 4.4, tale interfaccia è divisa in tre aree principali:

- Un'area per la visualizzazione dei messaggi, la cui resa grafica è migliorata utilizzando le librerie JavaScript `marked.js` e `highlight.js`, necessarie rispettivamente a formattare il codice Markdown restituito dal modello e fornire la colorazione della sintassi per i blocchi di codice (*syntax highlighting*).

- Un'area contenente il file attualmente aperto nell'editor, che può essere aggiunto o rimosso dal contesto cliccando su un'apposita icona, ed eventuali altri file allegati dall'utente.

- Un'area di input che consente all'utente di scrivere i propri messaggi, con dei pulsanti per allegare un file al contesto della conversazione, per attivare o disattivare la funzionalità di grounding e per inviare il messaggio.

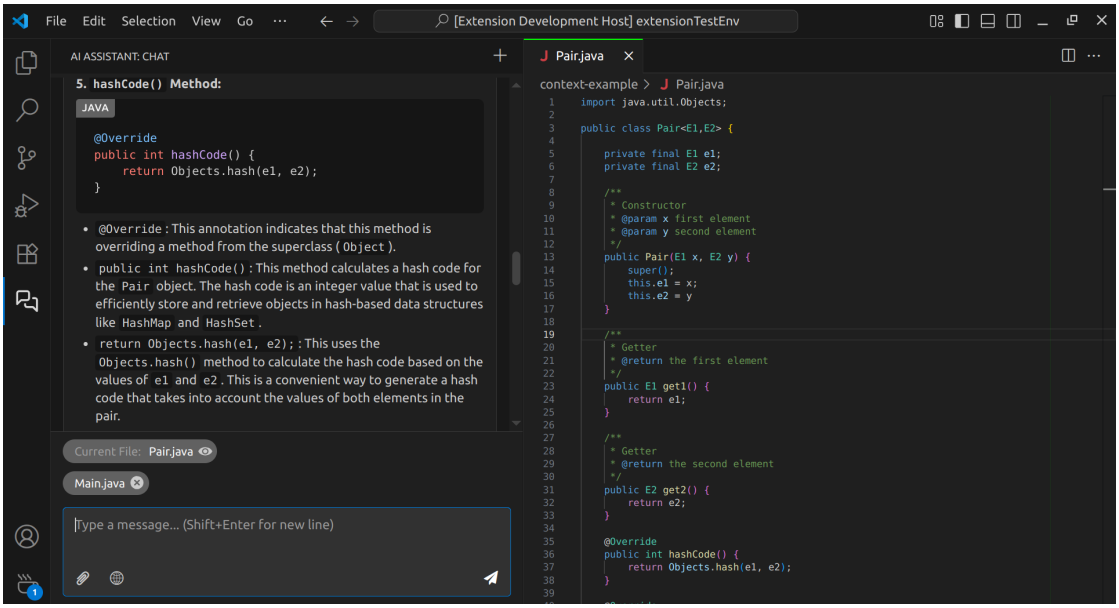

Figura 4.4: Interfaccia utente del pannello di chat.

Per quanto riguarda l'effettiva logica del frontend, questa è contenuta all'interno di un file JavaScript `main.js` (che viene iniettato nell'HTML della Webview) responsabile sia dell'aggiornamento dinamico dell'interfaccia che della cattura degli eventi generati dalle azioni dell'utente (ad esempio l'attivazione del grounding), inviando un apposito messaggio al backend che si occuperà della loro gestione.

Content Security Policy

Come per un sito web, anche la creazione di una Webview richiede di adottare alcune importanti misure di sicurezza, necessarie a limitare il contenuto che può essere caricato o eseguito al suo interno.
Questo perché, sebbene sia integrata nell'editor, una Webview è di fatto un browser web e come tale è soggetta agli stessi rischi, in particolare a quelli legati al *Cross-Site Scripting* (XSS), cioè l'iniezione di codice JavaScript potenzialmente malevolo.

Per mitigare tale minaccia, all'interno del tag `<meta>` del codice HTML è stata introdotta una Content Security Policy[18] (CSP), riportata nel Listato 4.20, che adotta un approccio *"deny-by-default"* che consente di bloccare il caricamento di qualsiasi risorsa non esplicitamente autorizzata.

```
<meta http-equiv="Content-Security-Policy" content="
    default-src 'none';
    script-src 'nonce-${nonce}';
    style-src ${webview.cspSource};
    img-src ${webview.cspSource};
    font-src ${webview.cspSource};
">
```

Listato 4.20: Content Security Policy adottata dalla Webview del pannello di chat.

L'elemento fondamentale per la prevenzione di attacchi XSS è l'utilizzo di un *nonce* (number used once), ossia un token che viene rigenerato casualmente da un metodo `getNonce` ad ogni nuova istanziazione della Webview.
Questo token deve essere associato a tutti gli script tramite l'attributo `nonce`, inclusi il file responsabile della logica del frontend e le librerie esterne per la miglior resa grafica dei messaggi. Così facendo, l'esecuzione di qualunque script che non presenti un nonce con l'esatto valore generato da `getNonce` verrà bloccata, impedendo eventuali tentativi di iniezione di codice.
Infine, per quanto riguarda le altre tipologie di risorse (fogli di stile, immagini e font), la CSP ne limita il caricamento solo da fonti interne all'estensione (`webview.cspSource`).

4.9.3 Comunicazione bidirezionale frontend-backend

Stabilito il canale di comunicazione bidirezionale tra il frontend e il backend si può descrivere il meccanismo basato su scambio di messaggi e mediato da `ChatService` che consente all'utente di interagire con la chat.
Ciascuno di questi messaggi è un oggetto con una proprietà `type` che ne definisce la natura e un contenuto variabile contenente i dati necessari per la sua elaborazione.

[18] w3.org/TR/CSP3

Comunicazione frontend → backend

La logica frontend contenuta nel file `main.js` della Webview cattura le interazioni dell'utente con l'interfaccia di chat (tramite dei listener che vengono aggiunti sui componenti HTML d'interesse) e converte tali eventi in messaggi da inviare al backend, utilizzando il metodo `vscode.postMessage`.

Poiché ci sono più azioni che l'utente può compiere interagendo con l'interfaccia di chat, a ciascuna di esse è stato associato un certo tipo di messaggio, che verrà poi elaborato lato backend. Questi sono:

- `userMessage`, inviato quando l'utente manda un messaggio all'assistente.
- `requestFileSelection`, inviato quando l'utente clicca sul pulsante per aggiungere un file al contesto.
- `toggleGrounding`, inviato quando l'utente clicca sul pulsante per attivare o disattivare la funzionalità di grounding.

Comunicazione backend → frontend

Come già visto nella Sezione 4.9.1, i messaggi ricevuti dal frontend passano tramite il listener registrato in `ChatViewProvider`, che li inoltra al metodo `handleWebviewMessage` di `ChatService`. Tale metodo, in base al tipo di messaggio ricevuto, si occupa poi di invocare la logica opportuna.
Ad esempio, in seguito alla ricezione di un messaggio utente (ovvero un messaggio frontend di tipo `userMessage`), viene creato un prompt "arricchito" contenente sia il testo inviato dall'utente che il contesto della chat (file e cronologia), invocando `AIService` per generare la risposta dell'assistente.

Terminata l'elaborazione di un messaggio ricevuto, il backend invierà a sua volta un messaggio al frontend (tramite il metodo `webview.postMessage`) per consentire l'aggiornamento dell'interfaccia del pannello di chat.

4.9.4 Persistenza dello stato della chat

Poiché il contenuto di una Webview viene eliminato quando l'utente chiude il pannello ad essa associato o l'IDE stesso, è necessario implementare un meccanismo che consenta di ripristinare lo *stato* della chat, ovvero la cronologia dei messaggi e i file aggiunti al contesto, in seguito alla riapertura del pannello da parte dell'utente.

Per farlo sono stati valutati due possibili approcci: mantenere lo stato della chat lato backend oppure lato frontend. Dopo varie considerazioni si è scelto di adottare quest'ultima soluzione, poiché la prima introduce il problema di dover sincronizzare il frontend con lo stato presente lato backend ogni volta che la chat viene riaperta. Tale approccio comporterebbe anche un maggior flusso di messaggi scambiati tra i due lati, introducendo della complessità aggiuntiva, poiché il frontend dovrebbe richiedere lo stato salvato lato backend ad ogni riapertura della chat, oltre che ad inviargli vari messaggi di sincronizzazione in seguito alle azioni dell'utente che ne modificano lo stato, come la rimozione di un file dal contesto.

Al fine di gestire lo stato della chat lato frontend è stato utilizzato il meccanismo di persistenza per le Webview fornito dall'API di VS Code[19], che consente di salvare e ripristinare lo stato tramite i metodi `vscode.setState` e `vscode.getState`.
Come si può vedere dal codice riportato nel Listato 4.21, in seguito all'apertura del pannello di chat il frontend tenta di recuperare lo stato precedente tramite `getState`, creandone uno vuoto in caso questo non esista. Ogni volta che lo stato viene modificato (in seguito all'invio di un nuovo messaggio, l'aggiunta/rimozione di un file dal contesto o il reset della chat), il nuovo stato aggiornato viene salvato tramite `setState`.

```
1 (function () {
2     const vscode = acquireVsCodeApi();
3     const oldState = vscode.getState();
4
5     let messages = oldState.messages || [];
6     let activeFiles = oldState.activeFiles || [];
7     ...
8     // request current file state from backend
9     vscode.postMessage({
```

[19] code.visualstudio.com/api/extension-guides/webview#persistence

```
        type: "requestCurrentFile"
    });
    ...
    // UI initialization
    const messagesContainer = document.querySelector("#chat-messages");

    if (messages.length > 0) {
        showPreviousMessages();
    }
    updateActiveFilesUI();
    messagesContainer.scrollTop = messagesContainer.scrollHeight;
```

Listato 4.21: Ripristino dello stato della chat lato frontend.

Aggiornamento del file corrente

Un ultimo elemento necessario al ripristino della chat è il file corrente, cioè quello attualmente aperto nell'editor. Questo dato, essendo dinamico, non può essere recuperato dallo stato in quanto potrebbe risultare non aggiornato.
Si supponga, ad esempio, che l'utente chiuda la chat con un determinato file aperto per poi riaprirla solo dopo aver selezionato un altro file: se l'informazione relativa al file corrente fosse stata memorizzata nello stato, alla seconda apertura della chat verrebbe ripristinato il valore associato al vecchio file anziché quello attualmente aperto.

Per ovviare al problema, come si può vedere dall'implementazione riportata nel Listato 4.22, ogni volta che la chat viene riaperta il frontend invia un messaggio di tipo `requestCurrentFile` al backend. Tale messaggio viene inoltrato a `ChatService` che, tramite la classe `ChatContextManager`, risponde con il file attualmente aperto nell'editor inviando un messaggio di tipo `updateCurrentFile`.
Inoltre, per far sì che il file corrente mostrato nella chat rifletta sempre quello effettivamente aperto nell'IDE, all'attivazione di `ChatContextManager` viene registrato un listener per l'evento VS Code `onDidChangeActiveTextEditor` (invocato ogni volta che l'utente cambia il file attivo), così da poter immediatamente aggiornare il frontend.

```
/* Inside ChatServiceImpl */
public async handleWebviewMessage(data: any): Promise<void> {
```

```
    switch (data.type) {
        case "requestCurrentFile":
            this.contextManager.sendCurrentFileToWebview();
            break;
        ...

/* Inside ChatContextManagerImpl */
// called by ChatService after the chat panel gets created
public setWebview(webview: vscode.WebviewView): void {
    this.webview = webview;
    this.editorChangeDisposable = vscode.window.onDidChangeActiveTextEditor(
        this.updateCurrentFile,
        this
    );
    this.updateCurrentFile(vscode.window.activeTextEditor);
}
...
private updateCurrentFile(editor?: vscode.TextEditor): void {
    if (...) { // check if there's an open file
        const filePath = editor.document.uri.fsPath;
        if (this.webview) {
            this.webview.webview.postMessage({
                type: "updateCurrentFile",
                filePath: filePath,
            });
        }
        ...
    }
}
```

Listato 4.22: Meccanismo di aggiornamento del file corrente mostrato nella chat.

Capitolo 5

Dimostrazione

In questo breve capitolo verranno illustrate le funzionalità dell'assistente AI realizzato, fornendo una serie di esempi concreti volti a dimostrare il rispetto dei vari requisiti definiti in fase di analisi.

5.1 Interazione contestuale

Come richiesto dal RF1, l'assistente espone vari comandi di interazione contestuale, implementati in modo tale che l'utente possa limitare la zona d'intervento dell'assistente alla sola porzione di codice selezionata nell'editor (RF2).
In Figura 5.1 viene mostrato l'utilizzo del comando di "Edit", che consente modificare il codice sulla base di istruzioni fornite in linguaggio naturale.

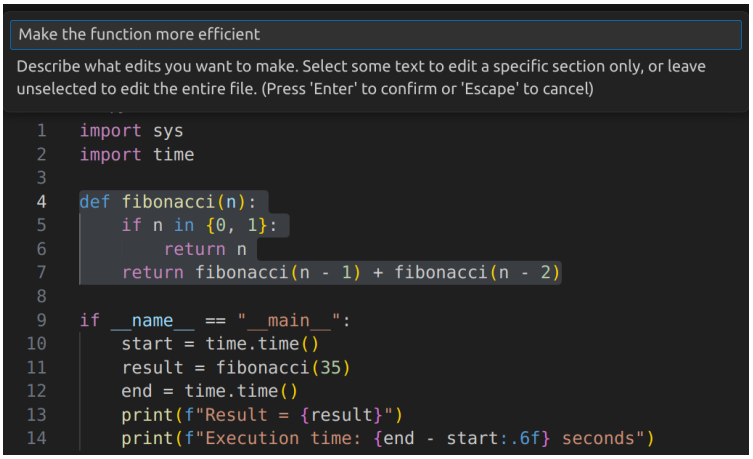

Figura 5.1: L'utente seleziona la funzione `fibonacci`, invoca il comando di "Edit" e inserisce nell'apposita input box la richiesta di renderla più efficiente.

Al termine dell'elaborazione, a prescindere dal comando di interazione contestuale invocato, l'assistente presenta all'utente le modifiche effettuate, offrendo la possibilità di confermarle o annullarle, come mostrato in Figura 5.2.

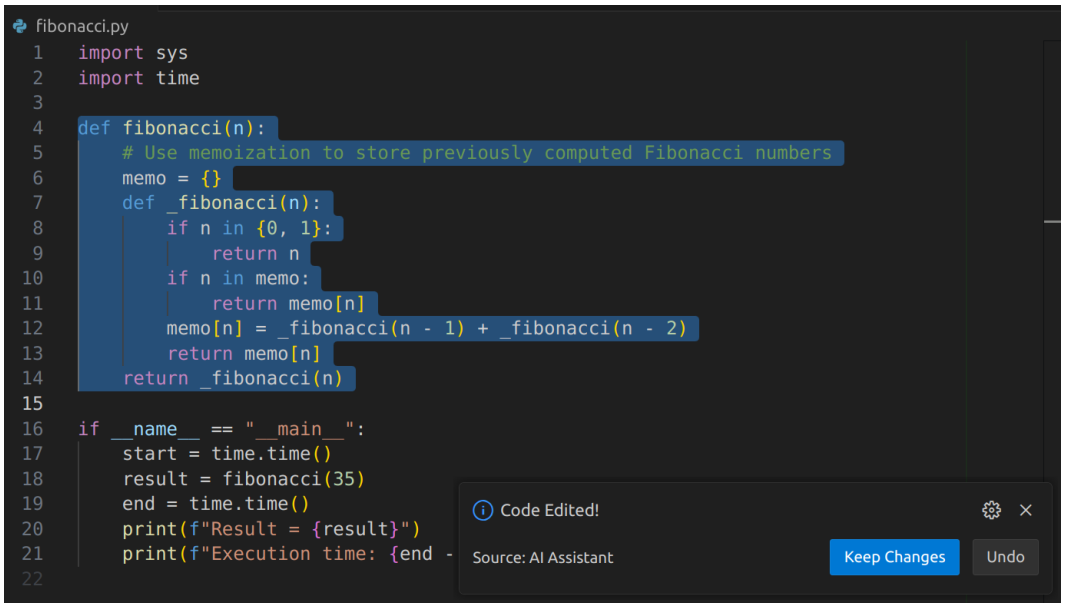

Figura 5.2: In seguito alla richiesta di ottimizzazione dell'utente, l'assistente propone un'implementazione più efficiente del metodo `fibonacci`.

5.2 Chat

5.2.1 Mantenimento della cronologia di conversazione

In Figura 5.3 viene mostrato come l'assistente integri il contesto del file attivo con la cronologia della conversazione per sostenere un dialogo coerente con l'utente, in linea con il RF3.

5.2.2 Aggiunta di risorse al contesto

Come richiesto dal RF4, l'utente può arricchire il contesto della conversazione allegando dei file all'interno della chat, in modo che l'assistente possa utilizzarne il contenuto per fornire risposte pertinenti.

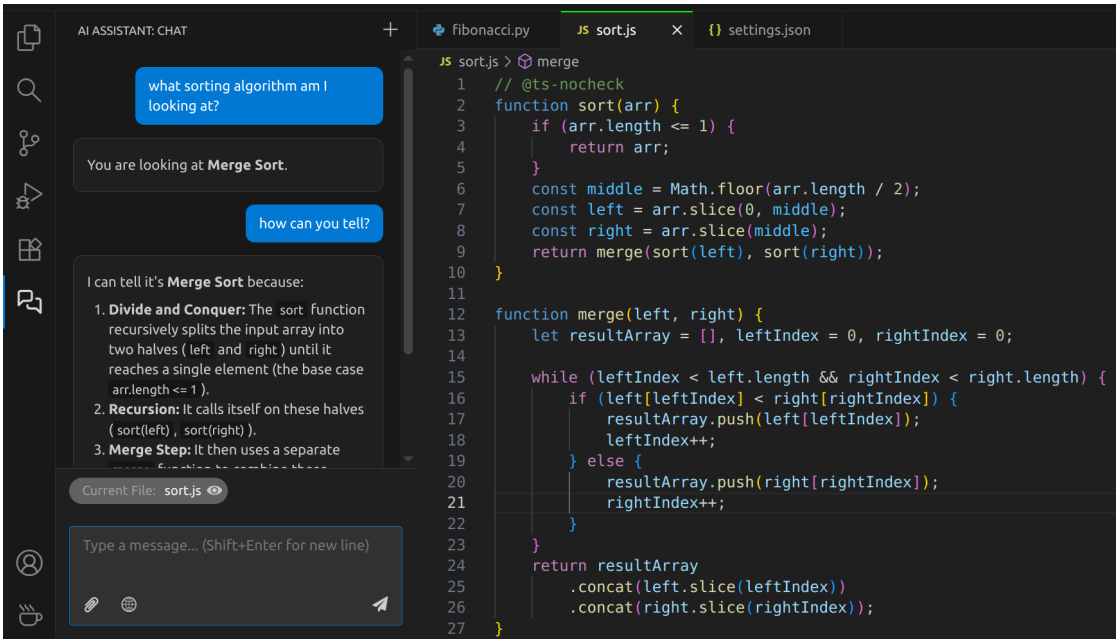

Figura 5.3: Dimostrazione del mantenimento della cronologia della chat. Innanzitutto l'utente richiede all'assistente quale sia l'algoritmo presente nel file corrente `sort.js`, che viene correttamente identificato dall'assistente come merge sort. Grazie al mantenimento della cronologia della conversazione, l'assistente comprende la domanda successiva ("how can you tell?") come una richiesta di spiegazioni sulla sua precedente affermazione, fornendo le relative motivazioni.

Per dimostrare questa capacità, è stato preparato uno scenario nel quale viene richiesto all'assistente di generare un diagramma UML che rappresenti l'intera architettura di una gerarchia di classi Java distribuita su file sorgente distinti. Tale operazione richiede necessariamente che l'assistente acceda al contenuto di tutti i file forniti, per poter mappare correttamente le relazioni di ereditarietà tra le diverse classi.

In Figura 5.4 viene mostrata la richiesta dell'utente e in Figura 5.5 il diagramma (corretto) prodotto dall'assistente.

[1] mermaid.js.org/

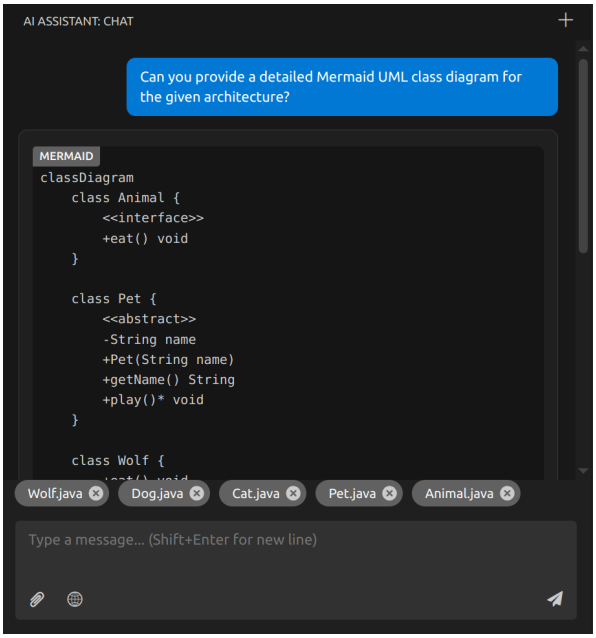

Figura 5.4: L'utente allega diversi file sorgente Java (visibili sotto l'area di input) al contesto della conversazione e richiede all'assistente di generare un diagramma UML in formato Mermaid[1] della gerarchia delle classi.

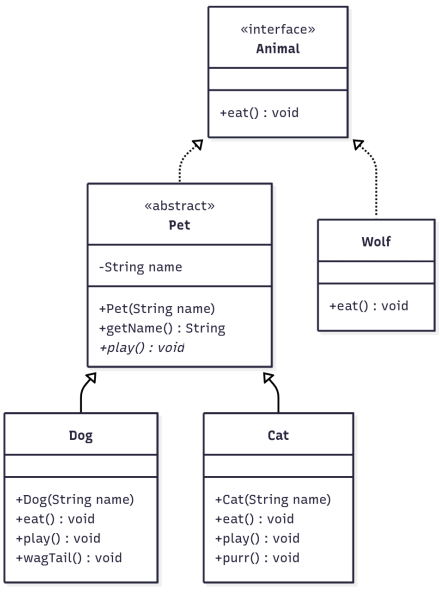

Figura 5.5: Diagramma UML generato dall'assistente sulla base dei file Java aggiunti al contesto, che modella correttamente la gerarchia delle classi.

5.2.3 Grounding

Come richiesto dal RF6, l'assistente è dotato di una funzionalità di grounding che, se attiva, gli consente di effettuare ricerche sul web qualora ritenuto necessario. Per dimostrare tale capacità, l'assistente è stato interrogato su un'informazione estremamente specifica (l'identità di un particolare utente GitHub).
Come mostrato in Figura 5.6, se il grounding è attivo l'assistente reperire le informazioni necessarie dal web, citando la query di ricerca utilizzata in fondo alla risposta fornita. Quando la stessa domanda viene posta con il grounding disattivato, l'assistente non è più in grado di fornire una risposta.

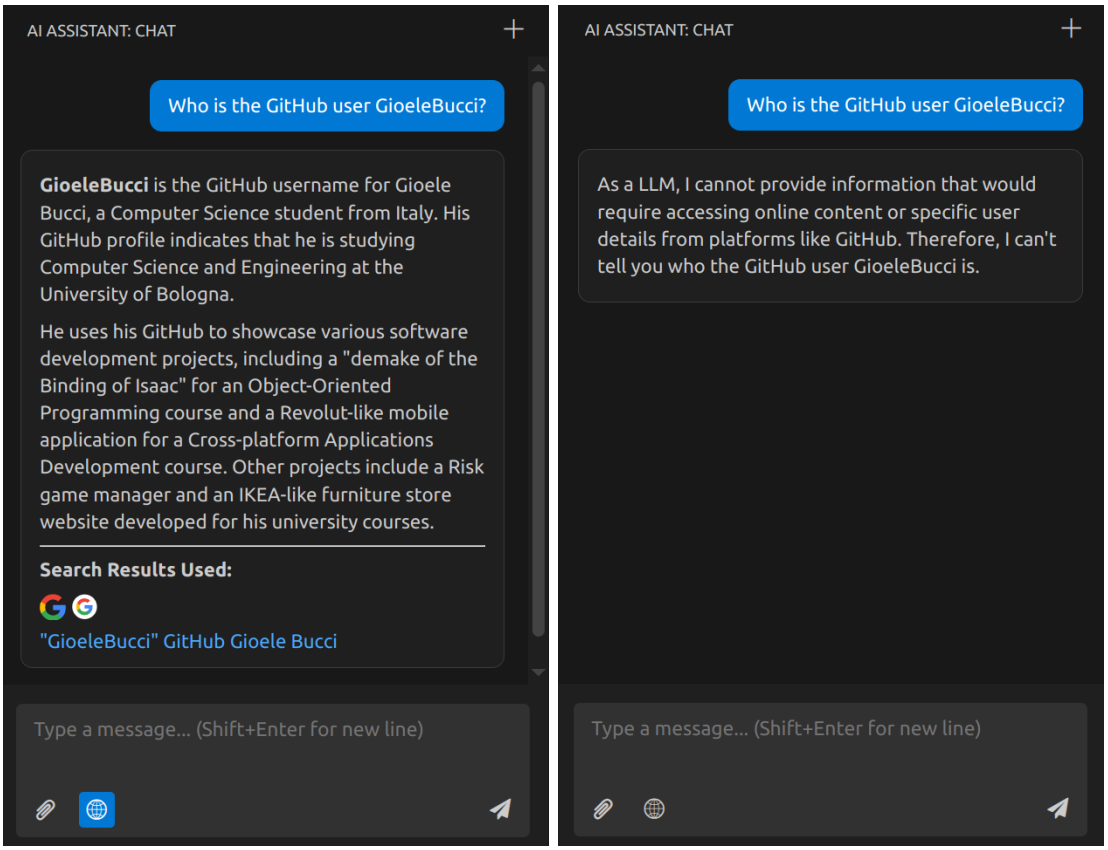

Figura 5.6: Dimostrazione dell'utilizzo della funzionalità di grounding all'interno della chat: a sinistra la funzionalità è attiva, mentre a destra è stata disattivata.

5.3 Inline completion

Come richiesto dal RF7, l'assistente è in grado di fornire suggerimenti di completamento del codice in tempo reale, analizzando il contesto del file corrente (incluso il codice già presente, i commenti e il linguaggio di programmazione) per anticipare l'intento del programmatore e proporre frammenti di codice pertinenti.
In Figura 5.7 è riportato un esempio di tale funzionalità in azione.

Figura 5.7: Esempio di inline completion per un file Java.

5.4 Istruzioni personalizzate

Come richiesto dal RF8, l'utente ha la possibilità di fornire direttive aggiuntive per personalizzare il comportamento dell'assistente, dichiarandole all'interno di un apposito file di configurazione (`.ai/agentInstructions.md`).
Nell'esempio riportato in Figura 5.8 si può vedere come l'output del comando di interazione contestuale "Docs" (che genera documentazione) sia stato modificato sulla base delle indicazioni fornite dell'utente.

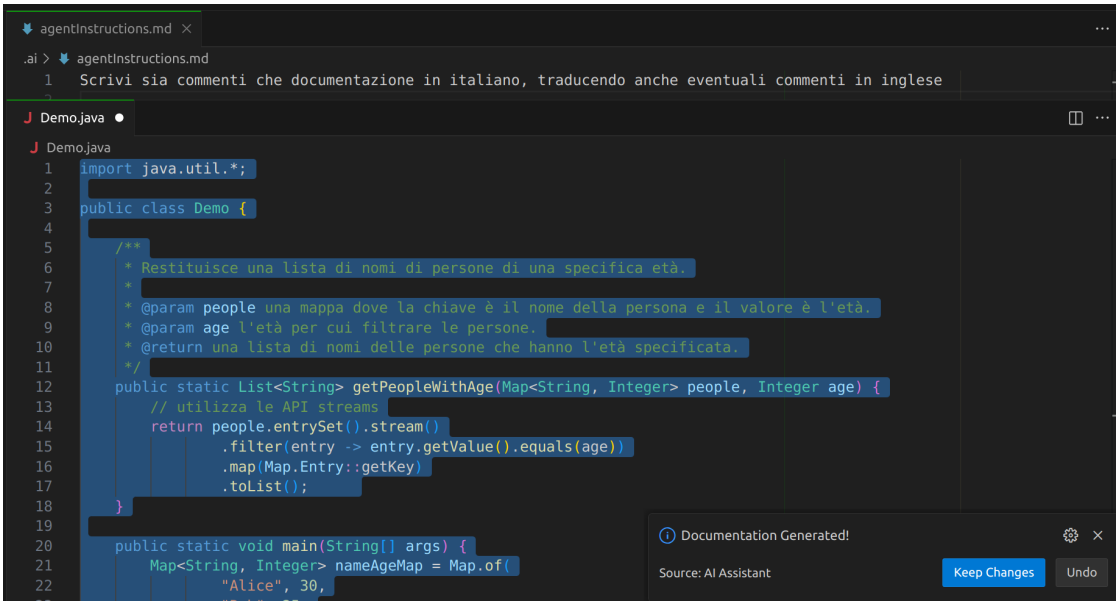

Figura 5.8: Dimostrazione dell'effetto delle istruzioni personalizzate. In alto, il file di configurazione `agentInstructions.md` dove l'utente ha richiesto che la scrittura della documentazione sia fatta in italiano. In basso, il risultato: in seguito all'esecuzione del comando "Docs", l'output dell'assistente rispetta la regola sulla lingua imposta dall'utente.

Capitolo 6

Conclusioni

L'obiettivo principale della tesi era quello di approfondire la conoscenza degli assistenti alla programmazione basati sull'intelligenza artificiale attraverso un approccio prevalentemente pratico. In seguito all'identificazione delle funzionalità principali di un simile prodotto già diffuso sul mercato (GitHub Copilot), è stato realizzato un prototipo ampiamente soddisfacente in grado di replicare con successo le principali funzionalità che caratterizzano un AI-assisted IDE moderno.

L'effettiva integrazione dell'assistente all'interno di un IDE affermato e maturo come Visual Studio Code si è rivelato uno dei compiti più onerosi, che ha richiesto un'approfondita fase di studio preliminare necessaria a comprendere le modalità di integrazione con l'ambiente. A questa sfida tecnica si è affiancata anche quella di rimanere al passo con la rapida evoluzione del settore dell'intelligenza artificiale: le API per l'interazione con i modelli sono soggette a continui cambiamenti e, anche durante il periodo di realizzazione di questo progetto, sono emersi nuovi standard e tecnologie volti a facilitare l'interazione tra strumenti AI-based e servizi esterni, come il Model Context Protocol (MCP[1]).

L'affermazione di questi standard sta inevitabilmente plasmando la nuova generazione di assistenti AI, nei quali le funzionalità agentiche stanno assumendo un ruolo sempre più prominente e sofisticato. Alla luce di ciò, è ragionevole identifi-

[1] modelcontextprotocol.io/docs/getting-started/intro

care come naturale evoluzione del progetto l'introduzione di tali capacità.
Un primo passo consisterà nel consentire all'assistente di interagire più liberamente con l'ambiente di sviluppo locale, eseguendo comandi e manipolando file in maniera autonoma per portare a termine compiti complessi. Successivamente, grazie alla diffusione di standard emergenti come MCP, sarà possibile estendere ulteriormente le capacità dell'assistente oltre i confini dell'IDE, abilitando l'integrazione con servizi esterni come sistemi di controllo di versione, piattaforme di documentazione o strumenti di CI/CD.

Bibliografia

[CDJ+24] Zheyuan Cui, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, and Tobias Salz. The effects of generative ai on high skilled work: Evidence from three field experiments with software developers. *SSRN*, 2024.

[MFP+25] Nestor Maslej, Loredana Fattorini, Raymond Perrault, Yolanda Gil, Vanessa Parli, Njenga Kariuki, Emily Capstick, Anka Reuel, Erik Brynjolfsson, John Etchemendy, Katrina Ligett, Terah Lyons, James Manyika, Juan Carlos Niebles, Yoav Shoham, Russell Wald, Tobi Walsh, Armin Hamrah, Lapo Santarlasci, Julia Betts Lotufo, Alexandra Rome, Andrew Shi, and Sukrut Oak. Artificial Intelligence Index Report 2025. Technical report, AI Index Steering Committee, Institute for Human-Centered AI, Stanford University, 2025.