UNIVERSITÀ DEGLI STUDI DELL'AQUILA

**Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica**

Tesi di Laurea Triennale in Informatica

# Sistema multi agente per code refactoring basato su attributi di qualità

Relatore
**Prof. Juri Di Rocco**

Correlatore
**Dr. Riccardo Rubei**

Laureando
**Ercole Rutolo**
**279065**

**Anno Accademico 2024-2025**

## Abstract

Ad oggi esistono numerosi studi e ricerche su come effettuare un code refactoring efficace, basandosi sulla qualità statica del codice. Tuttavia con questo studio di tesi, si prova a fare Code Refactoring di progetti pre-selezionati utilizzando come approccio la nuova frontiera degli LLM, ovvero un sistema multi agente. Questa modalità di lavoro offre non solo un nuovo metodo di studio, osservando come e se l'Intelligenza Artificiale applicata ad un sistema multi agente riesce a eseguire code refactoring, ma consente di espandere questo articolo di tesi in lavori futuri, sia nel campo del Software Engineering (quindi in particolare del code refactoring) e sia nel campo dei sistemi multi agente, andando a studiare a fondo nuove tecniche e nuove possibilità. L'obiettivo di questa tesi, quindi, è quello di effettuare un code refactoring efficace, basando i risultati su attributi di qualità come Security, Reliability, Maintainability, Duplications, Coverage, Complexity e utilizzando un approccio multi agente.
I progetti su cui è stato applicato l'approccio sono tutti open source, in Java, gestiti con Maven. In particolare sono stati selezionati progetti commerciali di Apache Commons e progetti di studenti ai primi anni di Università. Ciò serve a osservare principalmente come l'approccio proposto impatta su un tipo di progetti più strutturati e ben formati rispetto a progetti generalmente con più problematiche e più semplici. Infatti, quello che ci si aspettava era che l'approccio effettuasse un refactoring più efficace e significativo per i progetti studenteschi, rispetto a progetti Apache. Dopo aver costruito il dataset finale di progetti, di cui 11 Apache e 11 degli studenti, averne fatto l'analisi statica del codice grazie a SonarQube e dopo aver sviluppato ed eseguito il vero e proprio approccio multi agente, con framework CrewAI, i risultati sono stati quelli previsti. Si è visto, quindi, come l'approccio abbia effettuato refactoring dei progetti, attraverso l'analisi statica post refactoring e il confronto tra i valori prima e dopo, basando i risultati sul miglioramento o meno di determinati attributi di qualità. I risultati seguono ciò che ci si aspettava. Infatti sono mostrati, in percentuale, le modifiche di qualità subite per ogni progetto, e si è notato come per i progetti Apache ci siano stati più peggioramenti che miglioramenti, mentre per i progetti degli studenti l'andamento è stato opposto. I risultati, inoltre, portano anche degli spunti per quanto riguarda l'affidabilità e la validità dell'approccio proposto, sia dato un certo numero di classi per progetto e sia quando si specifica un particolare attributo di qualità su cui deve concentrarsi mentre effettua il refactoring. Questi risultati, in generale, hanno suggerito che il refactoring ben pianificato può contribuire positivamente all'affidabilità di un sistema software, in particolare quelli complessivamente meno complessi o ancora in via di sviluppo, costruendo e personalizzando l'approccio multi agente in base alle proprie esigenze specifiche. Questa tesi, infine, non vuole proporre un approccio multi agente perfetto, che migliori tutti gli attributi di tutti i progetti. Piuttosto, vuole dimostrare quanto gli LLM possano aiutare uno sviluppatore o un team di sviluppo nel fare code refactoring in modo automatico, cercando di migliorare il codice il più possibile, diminuendo il lavoro manuale pur mantenendo le funzionalità del progetto.



Keywords: multi-agent system, LLM, software engineering, code refactoring, quality attributes

# Indice

# Elenco delle figure

# Elenco delle tabelle

# Capitolo 1

# Introduzione

Si immagini un classico scenario in cui un team di sviluppatori abbia concluso con successo un progetto, rilasciandolo non affetto da errori o bugs e adempiendo tutti i requisiti funzionali richiesti. In apparenza, il progetto potrebbe essere utilizzato dai committenti senza particolari accorgimenti o precauzioni. Tuttavia, il progetto potrebbe manifestare delle problematiche non facilmente riscontrabili o prevedibili in fase di sviluppo e che essenzialmente pertengono ad aspetti non legati alla mera compilazione, bensì all'utilizzo vero e proprio. In fase di utilizzo infatti, potrebbero manifestarsi una serie di problemi di diversa gravità legati ad aspetti quali le performance, la sicurezza, ecc. La soluzione immediata sarebbe dunque quella di modificare il codice, rendendolo più pulito, più leggibile, più semplice, più sicuro. Tuttavia, se il codice presentasse numerosi problemi, allora questa procedura, se fatta manualmente, potrebbe essere lunga e dispendiosa. Di conseguenza, questo articolo di tesi mira a sviluppare un tool a supporto degli sviluppatori durante la fase di risoluzione dei problemi e di riscrittura del codice. Con questo approccio automatizzato, le tempistiche per eseguire tali modifiche verranno ridotte. Questo attraverso un sistema multi agente che, automaticamente, tenta di migliorare la qualità statica del codice. Per sistema multi agente (detto anche MAS ovvero Multi-Agent System) si intende un insieme di agenti intelligenti autonomi che interagiscono tra di loro per svolgere dei task al fine di raggiungere un determinato obiettivo. L'utilizzo di questo approccio è diventato sempre più comune per svolgere e creare software in modo programmatico con una divisione in step del lavoro. Si è scelto di utilizzare uno dei più comuni framework per creare e personalizzare un sistema multi agente, ovvero CrewAI.

Numerosi sono gli articoli in cui si testano singoli LLM e/o sistemi multi agenti su diversi campi di applicazione. A tal proposito, sono state effettuate molte ricerche sull'applicazione degli LLM e degli LLM basati su sistemi multi agente nel campo della Software Engineering, in particolare da David Lo [11] [10] [19], da Mark Harman [7] e da altri ricercatori [22] [29] [33] [35] [18], osservando quanto siano efficaci in alcune fasi del SE, con relativi open problems e possibili sviluppi futuri.
In questo lavoro di tesi ci si è voluto concentrare principalmente su uno dei più rilevanti task nel panorama SE, ovverosia il code refactoring. Per code refactoring si intende il processo di miglioramento di codice, che, però, mantiene tutte le sue funzioni principali, mettendo tutta l'attenzione, quindi, su come è scritto il codice piuttosto che su cosa fa. L'obiettivo di base del code refactoring è quello di migliorare la leggibilità e la manutenibilità del codice, eliminare possibili errori o vulnerabilità e rimuovere possibili duplicati. Quindi, in definitiva, il code refactoring è utile quando si vuole migliorare la qualità del codice, come evidenziato nei seguenti articoli [13] [14]. La scelta è ricaduta su tale task del SE perchè gli LLM, in generale, hanno discreti risultati su di esso e ci si aspetta lo stesso anche in questa tesi. Anche in questo

caso, sono numerose le ricerche scientifiche e gli articoli che trattano LLM, sistemi multi agenti o, in generale, l'applicazione del machine e deep learning con il code refactoring [34], [26], [23], [32], [25], [4]. Per cui, l'obiettivo di questo approccio basato su sistema multi agente è quello di effettuare un efficace e valido code refactoring di alcune classi di determinati progetti, guidando l'azione di refactoring in base alla qualità del codice [3]. L'analisi qualitativa del codice è stata effettuata attraverso l'utilizzo di SonarQube, uno strumento molto utile nell'analisi statica del codice. Tale strumento viene impiegato per estrarre attributi di qualità come Security, Reliability e Maintainability, e metriche come Bugs, Code Smells e Vulnerabilities. Dunque, verrà fatta un'analisi qualitativa del codice prima e dopo il refactoring, osservando quanto il sistema multi agente sia stato efficace nel migliorare gli attributi di qualità calcolati da SonarQube. I progetti che sono stati selezionati su cui effettuare code refactoring sono stati scelti in base a determinate caratteristiche e criteri, che verranno meglio specificati nella fase di validazione. Comunque, il dataset di progetti sarà formato principalmente da due tipi di progetti: progetti strutturati e commercali di Apache e progetti creati e sviluppati da studenti ai primi anni di Università, così da identificare potenziali differenze e approfondirne i risultati.

Per concludere questa introduzione: nel prossimo capitolo, il Capitolo 2, verranno presentati, spiegati e approfonditi il framework utilizzato per costruire il sistema multi agente e il software per effettuare l'analisi statica di ogni progetto; nel Capitolo 3 sarà spiegato l'approccio creato per poter effettuare refactoring attraverso gli agenti AI; nel Capitolo 4 verrà approfondito il metodo con cui è stato raccolto il dataset di progetti e verranno dichiarate e descritte le Research Question a cui questa tesi tenterà di rispondere; nel Capitolo 5 saranno presentate le tabelle contenenti i risultati dell'approccio, valutando se sia stato efficace nel fare refactoring, rispondendo alle Research Questions, ed effettuando un'approfondimento e una discusione dei risultati stessi; nel Capitolo 6 saranno presentati diversi approcci già esistenti che affrontano il code refactoring classico senza aiuto di LLM, poi il refactoring con l'utilizzo di un LLM. Inoltre, in questa sezione vengono riportati lavori recenti in cui sono stati impiegati meccanismi basati su multi-agent systems, dimostrandone la validità come in questo lavoro di tesi; nel Capitolo 7 verranno formulati i possibili sviluppi per l'approccio utilizzato in questa tesi, e di come poterne migliorare i risultati; nel Capitolo 8 infine, saranno presentate le conclusioni di questo lavoro di tesi.
Inoltre, per poter visualizzare e testare l'intero codice sviluppato si consulti il seguente URL:
*https://github.com/Reccolino/Multi-agent-system-for-code-refactoring-based-on-quality-attributes*

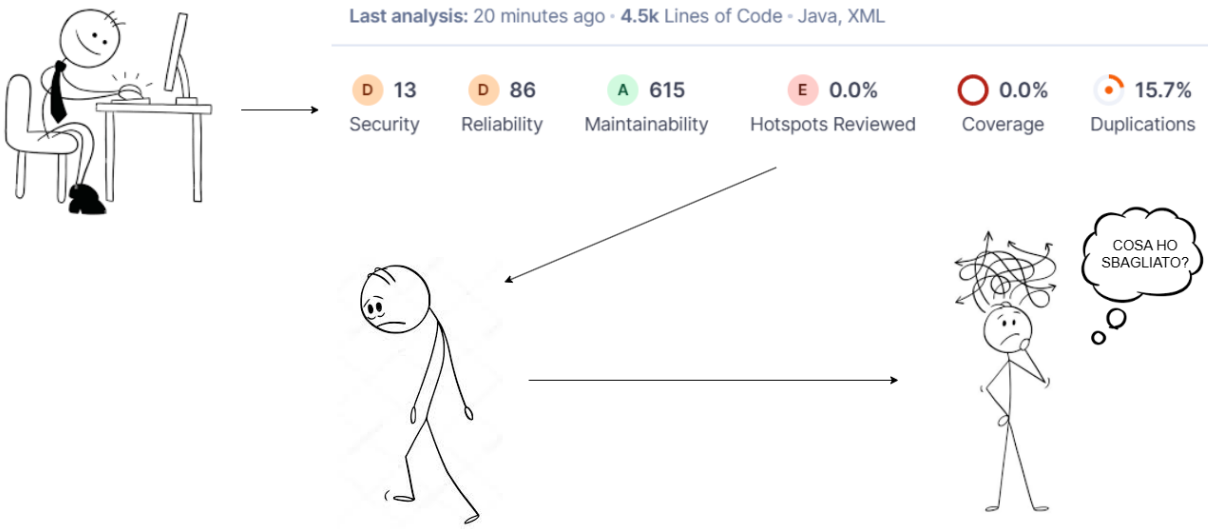
# Capitolo 2

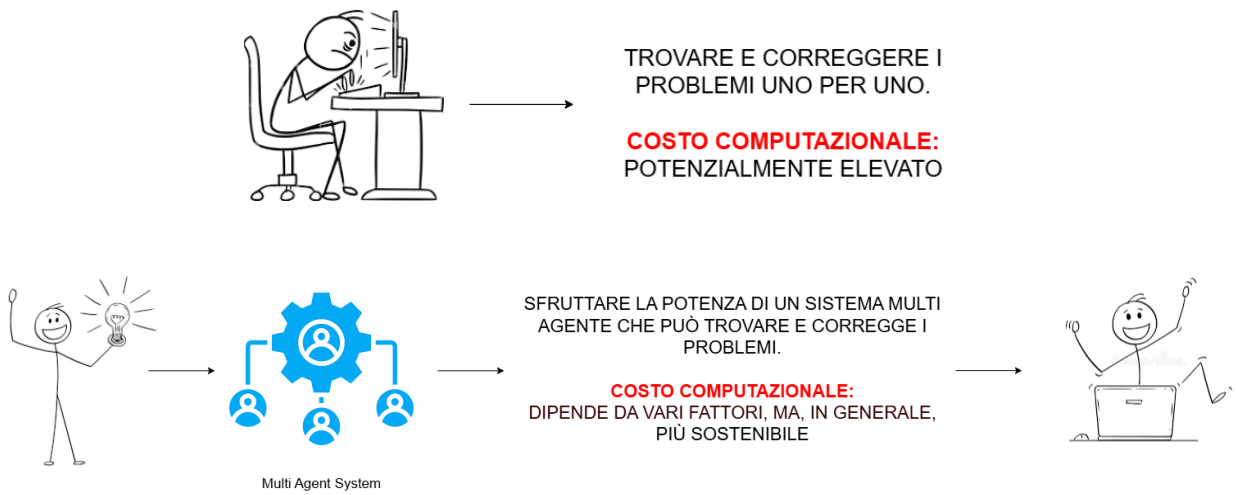
# Background

Di seguito verranno spiegati i framework e i software utilizzati in questo lavoro di tesi. In particolare nella Sezione 2.2 si farà un'introduzione e un approfondimento a CrewAI, il framework utilizzato per costruire il sistema multi agente, mentre in Sezione 2.3 si spiegherà cos'è SonarQube, il software utilizzato per eseguire l'analisi statica del codice prima e dopo il code refactoring. Mentre, nella prossima Sezione 2.1 verrà introdotto il problema che questa tesi si propone di affrontare, fornendo un esempio applicato alla realtà un esempio applicato alla realtà.

## 2.1 Problema

Come già annunciato nel Capitolo 1, l'obiettivo di questo lavoro di tesi è quello di aiutare e seguire gli sviluppatori nell'applicazione del code refactoring. In particolare, le immagini che seguono, dovrebbero far capire come l'approccio proposto potrebbe funzionare:






## 2.2 CrewAI

CrewAI[1] è un framework open-source per la creazione e la gestione di sistemi multi-agente [31]. Il termine "Crew" si riferisce proprio al fatto che CrewAI consente la creazione di agenti IA autonomi, ognuno dei quali svolge determinate attività detti task, che insieme, formano una crew, ovvero un gruppo di agenti che collaborano tra loro. Ogni agente funziona attraverso l'uso di un Large Language Model (LLM). Per LLM si intende un modello di Intelligenza Artificiale avanzato progettato per comprendere e generare linguaggio naturale in modo simile agli esseri umani. CrewAI consente una vasta personalizzazione, a partire dagli agenti e tasks fino alla crew, ai tools, e agli LLM utilizzati. La figura 2.1 ne è un esempio riassuntivo:

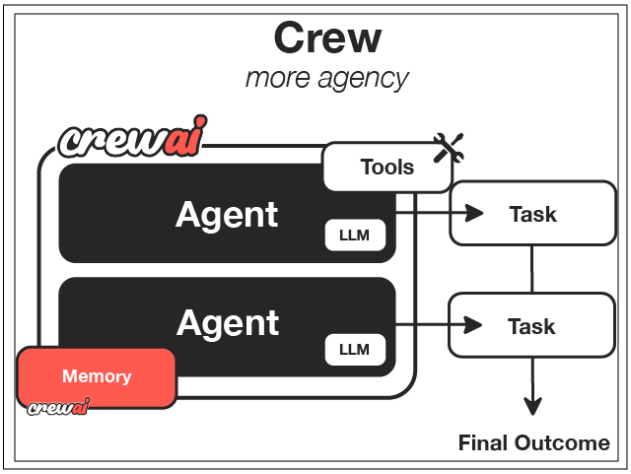



**Figura 2.1:** Concetto di Crew
Fonte: `https://docs.crewai.com/guides/crews/first-crew`

Una (o più) crew in CrewAI sono composte, come già accennato, principalmente da 2 elementi: gli agenti e i task. Ogni agente è un'entità autonoma che esegue delle task specifiche, prendendo decisioni in base al proprio ruolo, obiettivo e conoscenza. I task sono le mansioni e le attività che ogni agente deve eseguire. Ci sono, inoltre, tanti altri parametri opzionali che consentono la modellazione e personalizzazione di una crew. Come per esempio il Process, ovvero l'ordine con cui eseguire i task della crew, che può

[1] `https://www.crewai.com/`

essere Sequential (processo di default), in cui i task vengono eseguiti una dopo l'altro secondo un ordine stabilito, oppure Hierarchical, in cui i task vengono eseguiti secondo uno schema gerarchico. L'ordine di esecuzione, quindi, non è in funzione degli agenti, ma è in funzione dei task, eseguiti da determinati agenti. Oppure la Memory, cioè la funzione di una crew di mantenere memoria di ciò che è stato fatto che può essere short-term memory, long-term memory o entity-memory. Comunque, ci sono tanti altri parametri opzionali che possono essere utilizzati in base alle proprie esigenze. In questo lavoro di tesi, si è scelto di usare una crew semplice, con solo agenti e task senza ulteriore configurazione perchè oltre che essere funzionale, per poter usare altri parametri era necessario usufruire di modelli LLM e Embedder avanzati e a pagamento. La struttura con cui CrewAI gestisce, definisce e valida i dati di agenti, tasks e, in generale, di una crew, è definita dal modello Pydantic, una libreria Python che garantisce coerenza, sicurezza e auto-documentazione. Mentre, per quanto riguarda l'uso degli LLM, CrewAI si affida ad un backend astratto per la gestione di questi ultimi, chiamato LiteLLM. Grazie ad esso, è possibile utilizzare un buon numero di modelli, come quelli di Anthropic, di OpenAI, di Mistral, di Google, fino all'uso di ulteriori piattaforme che forniscono ulteriori modelli, come OpenRouter, HuggingFace o Ollama in locale. Tutto, ovviamente, seguendo la struttura predefinita di LiteLLM. Oppure, più semplicemente, si può utilizzare il provider di default di CrewAI, ovvero OpenAI.

Per concludere, la scelta di questo framework è dovuta alla semplice gestione di agenti e task e alla facilità di apprendimento e utilizzo. Tuttavia, è possibile espandere questa tesi, utilizzando diversi framework (o utilizzandoli insieme) basati su sistemi multi agenti. Di seguito si riportano alcuni articoli che trattano altri sistemi multi agenti popolari, facendo, inoltre, dei confronti proprio con CrewAI [2], [20], [5].

### 2.2.1 Agenti

Gli agenti [2] sono alla base dei sistemi multi-agente come CrewAI, in quanto sono i principali elementi che svolgono le attività richieste. Un agente di Intelligenza Artificiale si riferisce a un sistema o programma in grado di eseguire autonomamente attività per conto di un utente o di un altro sistema, progettando il proprio flusso di lavoro e utilizzando gli strumenti disponibili. Gli agenti di IA possono comprendere un'ampia gamma di funzionalità che vanno oltre l'elaborazione del linguaggio naturale, tra cui il processo decisionale, la risoluzione di problemi, l'interazione con ambienti esterni e l'esecuzione di azioni. I tre parametri fondamentali e obbligatori che modellano e impattano sul profilo di un agente in CrewAI sono il **Role**, ovvero il ruolo da svolgere dall'agente, che lo identifica all'interno del sistema, il **Goal**, cioè una breve descrizione dell'obiettivo dell'agente e la **Backstory**, quindi tutto ciò che riguarda la conoscenza che noi vogliamo mettere a disposizione dell'agente, che può essere utilizzata per raggiungere il suo obiettivo. Altri parametri opzionali, ma comunque importanti ai fini del lavoro di tesi sono stati l'**LLM** e, quindi, il modello IA utilizzato per l'agente che, se non specificato, di default è GPT o3-mini di OpenAI. A sua volta, l'LLM scelto per gli agenti può essere personalizzato, andando a modificare e definire parametri come Temperatura, Top P, Max Tokens, Frequency Penalty e Presence Penalty, Seed, ecc... Per cui, in definitiva, un agente in CrewAI è un'entità autonoma che esegue dei task specifici, prendendo decisioni in base al Role, al Goal e alla Backstory, utilizzando eventualmente dei Tools, cioè delle capacità ulteriori, per raggiungere i propri obiettivi e un LLM per generare risposte adeguate e interagire con l'ambiente.

[2] https://docs.crewai.com/concepts/agents

### 2.2.2 Tasks

I task [3] sono le mansioni e le attività che ogni agente deve eseguire. Così come per gli agenti, anche le task sono personalizzabili. I 3 parametri fondamentali per modellare un task in CrewAI sono il **Description**, cioè una descrizione dettagliata di ciò che l'agente deve fare, l' **Expected Output**, ovvero ciò che ci si aspetta che l'agente restituisca e l'**Agent** cioè l'agente che si deve occupare di quello specifico task. Un altro parametro opzionale, ma fondamentale ai fini del lavoro di tesi è il **Tools**, cioè una lista di strumenti che consentono di dare una competenza specifica aggiuntiva agli agenti per eseguire quello specifico task, aiutandoli a raggiungere ancor di più il loro obiettivo. Per i Tools, CrewAI mette a disposizione una serie di strumenti predefiniti, come ad esempio il tool per fare richieste HTTP, il tool per interagire con i file, il tool per interagire con i database, e tanti altri [4] oppure è possibile crearne dei propri, andando a definire il nome del tool da utilizzare, la descrizione e le funzioni che lo caratterizzano.
Un particolare da far notare è che un task può essere eseguito da più agenti, in modo da poter sfruttare le competenze di più agenti per raggiungere un obiettivo comune, oppure, al contrario, un agente può eseguire più task, così da poter sfruttare le competenze di un singolo agente per raggiungere più obiettivi. Inoltre, è possibile definire anche Conditional Tasks, che vengono eseguiti solo se certe condizioni, definite dall'utente, sono effettivamente soddisfatte. Un ultimo parametro opzionale importante è il **Context**, una lista di task da cui catturare i contesti. Grazie ad esso si possono far comunicare task fra loro, passando l'output di uno come input di un altro. Di default, però, se il processo della crew è Sequential, allora il parametro context non ha alcun effetto, dato che tramite processo sequenziale, i contesti vengono già passati tra un task e un altro. Mentre se si vuole una maggior sicurezza, flessibilità e coordinazione dei task, questo parametro può risultare molto utile.

### 2.2.3 Flow

Il concetto di flow [5] in CrewAI è leggermente distaccato dal processo fra task e agenti che avviene nella crew. I CrewAI flows sono un modo per poter combinare la potenza degli agenti AI con la flessibilità di codice procedurale. In questo modo è possibile creare dei veri e propri flussi di lavoro, con la possibilità di personalizzare tale flusso, sfruttando le potenzialità degli agenti AI delle Crew. Per esempio, un Flow può essere utilizzato quando si ha la necessità di eseguire del codice che sia esterno alla Crew, come chiamate API o collegamenti a database. Oppure è utile quando si vogliono modellare i dati di input e/o di output della Crew. Un altro campo di applicazione è quello della creazione di flussi condizionali, complessi e dinamici, cosa che la classica Crew in CrewAI non riesce ancora a fare a pieno.

[3] https://docs.crewai.com/concepts/tasks
[4] https://docs.crewai.com/concepts/tools
[5] https://docs.crewai.com/concepts/flows

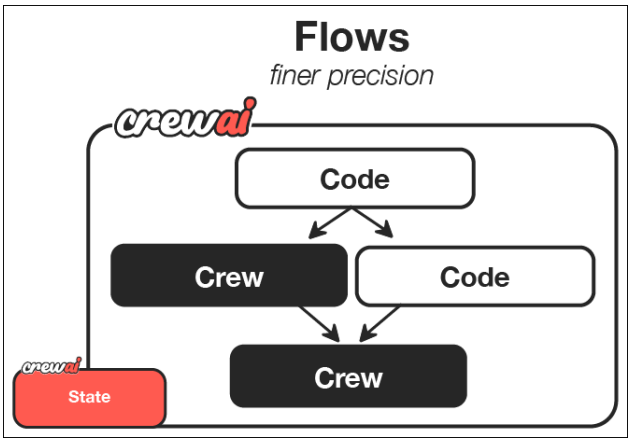



**Figura 2.2:** Concetto di Flow
Fonte: `https://docs.crewai.com/guides/flows/first-flow`

Ci sono diversi modi per costruire un CrewAI flow. In questa ricerca, si è scelto di utilizzare un flow di tipo Strutturato. Questo tipo di flow utilizza modelli Pydantic per definire uno schema per lo stato del flusso, garantendo sicurezza dei tipi, convalida e una migliore esperienza per gli sviluppatori. Tale modello verrà usato per rappresentare lo stato delle variabili e, in generale, della struttura del flow. Si accede allo stato tramite self.state, che si comporta come un'istanza del modello Pydantic. Il vantaggio principale di Pydantic è che garantisce che tutte le variabili siano del tipo giusto (type safety). La scelta della Structured Flow è stata fatta per garantire una maggiore sicurezza e coerenza nella gestione dello stato del flusso di lavoro, in particolare per poter gestire in modo più semplice i dati e le variabili all'interno del flusso, il tutto combinato con la potenza e l'esecuzione della Crew di agenti, in modo tale da poter modellare sia i dati di input che i dati di output proprio della Crew.
I flow funzionano attraverso dei decorator, che permettono di definire il percorso del flusso di lavoro e le funzioni che lo compongono. A tal proposito, di seguito, sono riportati i decorator utilizzati in questa ricerca:

**@start** : Definisce il punto di partenza del flusso di lavoro. Tutti i metodi con tale decorator vengono eseguiti in parallelo quando il flusso viene avviato.

```
class OutputExampleFlow(Flow):
@start()
def first_method(self):
    return "Output from first_method"
```

**@listen** : Definisce tutti quei metodi che devono essere eseguiti subito dopo un altro. Ha la funzione di "listener" dell'output del metodo precedente.

```
@listen(first_method)
def second_method(self, first_output):
    return f"Second method received: {first_output}"
```

**@router** : I metodi con questo decorator hanno la funzione di instradare l'output di un metodo precedente verso uno o più metodi successivi proprio come un vero e proprio router.

```
@router(start_method)
def second_method(self):
    if self.state.success_flag:
        return "success"
    else:
        return "failed"

@listen("success")
def third_method(self):
    print("Third method running")

@listen("failed")
def fourth_method(self):
    print("Fourth method running")
```

Tramite questi decorator è stato possibile creare un Flow strutturato e complesso, grazie alla quale è stato possibile gestire determinate condizioni, determinate esecuzioni e il percorso ben definito.
Inoltre, è stato utile anche la condizione logica **or_**, utilizzata all'interno dei decorator. Questa condizione (insieme a and_, che però non è stata utilizzata in questo approccio) è una funzione che, se inserita all'interno di @listen o di @router, permette di scegliere quale metodo eseguire. Come per esempio:

```
@start()
def start_method(self):
    return "Hello from the start method"

@listen(start_method)
def second_method(self):
    return "Hello from the second method"

@listen(or_(start_method, second_method))
def logger(self, result):
    print(f"Logger: {result}")
```

In questo caso, l'ultimo @listen, si attiverà quando uno qualsiasi dei metodi presenti nell'or_ è completato e viene chiamato una volta per ogni trigger attivato. Mentre, se si fosse usato l'and_ , l'ultimo @listen si sarebbe attivato quando tutti i metodi presenti nella condizione sono completati e sarebbe stato chiamato un unica volta.
In definitiva, questo metodo di lavoro ha permesso l'esecuzione e l'effettiva realizzazione del sistema multi-agente per il code refactoring, in quanto è stato utile per modellare i dati di input e di output della Crew, come i progetti, le classi, i path, gli errori e i tempi di esecuzione. Senza di esso non sarebbe stato possibile realizzare tale sistema, in quanto, con solo l'utilizzo della Crew, spesso, tra gli agenti, si perdeva l'integrità di alcuni dati fondamentali, come path o classi, che compromettevano l'intera esecuzione del sistema. In questo modo, il flow ha permesso di gestire questi dati fondamentali in modo sicuro e programmatico , passandoli come input per la crew, che si è occupata principalmente di effettuare code refactoring. Inoltre, il flow ha contribuito a rendere il sistema multi-agente più flessibile e dinamico, andando a gestire eventuali flussi iterativi e condizionali, cosa che la crew, da sola, non poteva fare. O meglio, era in grado di gestire delle condizioni, grazie all'uso di Conditional Task ma ciò iniziava e finiva all'interno della crew stessa,

senza la possibilità di rieseguirla dall'inizio passando i dati da una esecuzione all'altra. Per questo motivo, il flow è stato utile soprattutto per gestire questa evenienza, ovvero far ripartire l'esecuzione di una crew, tenendo conto di alcuni dati restituiti dall'esecuzione precedente della stessa.

## 2.3 SonarQube

SonarQube[6] è una piattaforma open-source per l'analisi statica del codice, che permette di analizzare il codice sorgente di un progetto software e di fornire informazioni utili sulla qualità del codice stesso. SonarQube si può utilizzare sia sul web, tramite la versione Cloud, che in locale, tramite l'installazione di un server che gira su una porta specifica (di default 9.000). La versione in locale permette di effettuare più configurazioni rispetto a quella Cloud che pone dei limiti in termini di numeri di progetti e di personalizzazione (se si utilizza la versione gratuita).

### 2.3.1 Attributi di qualità

Per poter effettuare un'analisi statica di un software, SonarQube si basa su vari **attributi di qualità**. Gli attributi di qualità sono le caratteristiche del codice che SonarQube analizza e misura per ogni file del progetto software per poterne valutare la qualità [11]. Di seguito, i più importanti su cui si è basato questo lavoro di tesi:

**Security:** Raggruppa tutte quelle regole che misurano la sicurezza del codice. Viene misurata attraverso le *vulnerabilities*, ovvero possibili errori di sicurezza come SQL injection, password non protette, API key in vista, ecc. Inoltre, se uno snippet di codice contiene un potenziale problema di sicurezza che richiede una revisione manuale, viene identificato come *security hotspot*, il che contribuisce alla valutazione degli Hotspots Reviewed.
*Risponde alla domanda:* quanto è sicuro il codice?

**Reliability:** Raggruppa tutte quelle regole che misurano la correttezza funzionale del codice, ossia la probabilità che esso contenga errori di programazione che possono manifestarsi a runtime. Viene misurata tramite il conteggio di *bugs*, ovvero pattern di codice che, se non corretti, rischiano di causare comportamenti imprevisti o crash.
*Risponde alla domanda:* quanto è affidabile il codice?

**Maintainability:** Raggruppa tutte quelle regole che misurano quanto è facile: comprendere, estendere e modificare il codice nel tempo. Si misura attraverso due indicatori principali: i *code smells*, ovvero porzioni di codice che hanno problemi come metodi troppo lunghi, duplicazioni, e, in generale, problemi che abbassano la qualità strutturale del progetto e il *technical debt ratio*, ovvero il rapporto tra il tempo stimato per risolvere i code smells e il tempo stimato per scrivere il codice.
*Risponde alla domanda:* quanto è difficile mantenere, modificare e migliorare il codice?

**Complexity:** Si basa sul concetto di complessità del codice, che può influenzare la comprensibilità e la manutenibilità. Viene misurata attraverso due metriche: la *cognitive complexity*, ovvero quanto è difficile da comprendere il codice per un essere umano e la *cyclomatic complexity*, cioè quanto è profondo e complesso

[6] `https://www.sonarsource.com/`

un ramo (if, for, while, ecc.).
*Risponde alla domanda:* quanto è complesso il codice ?

**Duplications:** Considera tutte le regole che misurano la presenza di codice duplicato (linee, blocchi e/o file) all'interno del progetto. Viene misurato attraverso la *duplicated lines density*, cioè la percentuale di linee duplicate rispetto al totale del codice.
*Risponde alla domanda:* quanto codice è duplicato all'interno dell'intero progetto ?

**Coverage:** Verifica, in percentuale, quanto codice all'interno del progetto è coperto da test automatici. Viene misurato tramite la *code coverage* ovvero il rapporto tra linee di codice eseguite durante i test e linee non eseguite durante i test.
*Risponde alla domanda:* quanto codice è coperto dai test automatici ?

Attributi come Security, Reliability e Maintainability sono misurati attraverso un **Rating** che va dalla A (migliore) alla E (peggiore), mentre gli altri attributi sono misurati attraverso valori numerici o percentuali, come Duplications, Coverage e Complexity. Si porrà particolare attenzione ai primi tre attributi e rispettive metriche, perchè sono i più significativi e importanti ai fini del lavoro [7]. Da specificare che SonarQube offre moltri altri attributi di qualità e metriche, che, però, non sono state prese in considerazione per questo caso di studio, perchè ritenute ridondanti. Di seguito, nella Tabella 2.1, un riassunto degli attributi e da quale metrica sono influenzati:

| **Attributo** | **Metrica Principale** |
|---|---|
| Security | vulnerability |
| Reliability | bug |
| Maintainability | code smell, debt ratio |
| Complexity | cognitive/cyclomatic complexity |
| Duplications | duplicated lines density |
| Coverage | code coverage |

**Tabella 2.1:** Sintesi degli attributi di qualità del codice e rispettive metriche

### 2.3.2 Metriche

Ogni attributo di qualità raggruppa al suo interno delle **metriche** [12], che sono i valori numerici o percentuali che rappresentano lo stato di quel particolare attributo. Quindi, per esempio, come già visto anche nella Tabella 2.1, la Security è misurata attraverso le vulnerabilities [9], la Reliability è misurata attraverso i bugs, mentre la Maintainability è misurata attraverso le code smells [16], e così via. Queste metriche vengono identificate come **Issues**, nell'analisi statica del codice, e influiscono sul Rating dei corrispettivi attributi di qualità. Ogni Issues può essere identificato come: Blocker, Critical, Major, Minor e Info, in ordine di gravità. Particolare attenzione nei Blocker e nei Critical, che saranno usati successivamente anche per l'approccio e nei risultati: un Blocker Violation è un problema che ha una probabilità significativa di gravi conseguenze involontarie sull'applicazione che dovrebbe essere risolto immediatamente. Ciò include bug che portano a crash di produzione e difetti di sicurezza che consentono agli aggressori di estrarre

dati sensibili o eseguire codice dannoso; un Critical Violation, invece, è un problema con un forte impatto sull'applicazione che dovrebbe essere risolto il prima possibile. Si distingue invece il caso dei code smells: SonarQube li traduce in minuti di lavoro per calcolare il technical debt ratio, cioè il tempo medio per la loro correzione. Di seguito la Tabella 2.2 di come queste metriche influiscano sul Rating dei rispettivi attributi di qualità (Security, Reliability, Maintainability):

| Rating | Descrizione |
|---|---|
| A | Nessun bug/vulnerability o almeno uno di tipo **Info**. technical debt ratio $\leq 5\%$. |
| B | Almeno un bug/vulnerability di tipo **Minor**. technical debt ratio tra il 6% e il 10%. |
| C | Almeno un bug/vulnerability di tipo **Major**. technical debt ratio tra l'11% e il 20%. |
| D | Almeno un bug/vulnerability di tipo **Critical**. technical debt ratio tra il 21% e il 50%. |
| E | Almeno un bug/vulnerability di tipo **Blocker**. technical debt ratio superiore al 50%. |

**Tabella 2.2:** Criteri di assegnazione dei Rating in SonarQube

Quindi SonarQube non lavora per quantità, ma piuttosto ci concentra maggiormente sulla qualità.

### 2.3.3 Gestione del codice

È importante ora parlare di come SonarQube lavora effettivamente con questi attributi di qualità. In particolare SonarQube analizza un progetto basandosi su due tipi di codice: il New Code e l'Overall Code. Il New Code è il codice che è stato aggiunto o modificato entro un intervallo di tempo definito (detto "New Code Period" o "Leak Period"), mentre l'Overall Code è il codice totale del progetto, ovvero la somma del New Code e del codice già esistente. Il New Code è il codice su cui SonarQube si concentra maggiormente, in quanto è quello che ha subito modifiche e potrebbe introdurre nuovi problemi e può essere personalizzato in base alle esigenze del team di sviluppo del progetto. Quindi per New Code si può intendere il codice che è stato aggiunto o modificato negli ultimi 30 giorni, negli ultimi 6 mesi, o più semplicemente il codice che è stato aggiunto o modificato dopo una certa data di riferimento, come la data di creazione del progetto o la data dell'ultima analisi. In questo lavoro di tesi ci si concentrerà maggiormente sull'Overall Code, in quanto, contiene tutti i valori di attributi di qualità e metriche utili ai fini della validazione dei risultati. Il New Code, seppur importante, offre, in questo caso, solo un'analisi sul nuovo codice che è stato scritto o trasformato, e quindi consente solo l'analisi statica di tale codice. Per cui, per fare un esempio, è possibile che nel fare code refactoring l'approccio aggiunga solamente righe commentate, per cui tali righe

non influiranno sull'Overall Code che rimarrà invariato, ma influirà solo sul New Code che, al contrario, rileverà che non ci sono stati peggioramenti e, anzi, che quelle righe commentate rispettano determinati valori di attributi e metriche (perchè difatti sono righe commentate e quindi non hanno influenza su sicurezza, affidabilità ecc...). Detto ciò, è necessario parlare dei Quality Gates e dei Quality Profiles.

### 2.3.4 Quality gates and profiles

I Quality Gates[7] sono dei set di regole che il New Code o l'Overall Code di un progetto deve rispettare per essere considerato di qualità accettabile. In pratica, sono dei criteri che un progetto deve soddisfare per essere considerato "pronto" o "accettabile". Per capire, SonarQube fornisce un Quality Gate di default, chiamato Sonar Way, che contiene delle regole predefinite per ogni attributo di qualità. Per esempio, il Quality Gate di default prevede che un progetto abbia almeno l'80% di code coverage, che non contenga Issues, che abbia un duplicates lines density $\leq 3\%$ e che abbia tutti i Security hotspot revisionati. È anche possibile specificare regole per l'Overall Code. Mentre, per quanto riguarda la personalizzazione, è possibile creare Quality Gates personalizzati, quindi, per esempio, andando ad accettare degli Issues, o andando a modificare la percentuale di Coverage Test o creando delle nuove regole come l'accettazione o meno di un certo Rating per un qualche attributo di qualità. Il soddisfacimento di una Quality Gate implica lo stato del New Code, che può essere "Passed" (superato) o "Failed" (non superato), secondo le regole imposte dallo stesso Quality Gates. Inoltre, è possibile creare più Quality Gates e assegnarli a progetti diversi, mentre il Quality Gate di default è assegnato a tutti i progetti che non ne hanno una specifica assegnata.
I Quality Profiles[8] , invece, sono dei set di regole che definiscono le metriche e le regole di qualità per un determinato linguaggio di programmazione. Per esempio, SonarQube fornisce un Quality Profile di default per Java che contiene delle regole predefinite per ogni attributo di qualità. Per cui, le metriche seguono le regole definite dalla Quality Profile assegnata al progetto. È possibile utilizzare Quality Profiles diverse da quelle di default, andando a eliminare delle regole o andando a crearne di nuove, sia personali che non. Infatti, SonarQube, fornisce dei plugin da scaricare per determinati linguaggi di programmazione. In questo lavoro di tesi si è scelto di usare i seguenti plugin Java: **Checkstyle**, per la cura dello stile e delle conventions, **PMD**, per identificare codice non utilizzato e ulteriori bugs e code smells nel codice sorgente e **FindBugs**, per identificare ulteriori bugs e vulnerabilità nel bytecode. Ognuno di essi, quindi, contiene delle regole specifiche. Perciò, è possibile utilizzare uno di questi plugin oppure combinarli o ereditarli tra di loro. Per cui la personalizzazione delle regole per il proprio Quality Profile è molto ampia e permette di adattare SonarQube alle esigenze del proprio progetto.
È importante notare che più regole ci sono attive in un Quality Profile, più è probabile che il progetto non ne rispetti alcune e, di conseguenza, conterrà più bugs, vulnerabilities e/o code smells (quindi Issues) e ciò significa che non supererà il Quality Gate.

In questo lavoro di tesi ci si concentrerà solo ed esclusivamente sulle Quality Profiles, in quanto contribuiscono in modo importante ai valori di qualità dell'Overall Code dei progetti. Mentre il Quality Gate verrà tralasciato perchè, come già detto, ci si vuole concentrare maggiormente sull'Overall Code piuttosto che sul New Code, ignorando il soddisfacimento o meno di vincoli qualitativi solo del New Code. Questo

[7] https://docs.sonarsource.com/sonarqube-server/latest/quality-standards-administration/managing-quality-gates/introduction/

[8] https://docs.sonarsource.com/sonarqube-server/latest/quality-standards-administration/managing-quality-profiles/introduction/

implica il fatto che alla fine dell'approccio è possibile che alcuni New Code siano identificati come "Failed" perchè non hanno rispettato vincoli determinati del Quality gates, tuttavia il refactoring è comunque stato ritenuto efficace perchè , in generale, l'approccio è riuscito a migliorare (o a non peggiorare) l'Overall Code e, di conseguenza, ad aver migliorato alcuni attributi o metriche di qualità.
La scelta definitiva delle Quality Profiles è ricaduta su un Profile personalizzato, adattato a tutti i progetti analizzati, che eredita tutte le regole dal Quality Profile di default per Java di SonarQube, Sonar Way, e che eredita alcune delle regole dai plugin Java sopra citati, in particolare: regole che identificano Issues Blocker e Issues Critical per la Security, la Reliability e la Maintainability, e regole che identificano tutti problemi sotto la famiglia del Correctness, CWE (Common Weakness Enumeration) e Bad Practise del codice [24]. Nel primo caso perchè si vuole cercare di correggere errori più gravi e importanti con il code refactoring, dando priorità a ai 3 attributi di qualità principali. Mentre nel secondo caso, si è fatta una scelta implementativa, andando a scegliere tutte quelle regole facenti parte della famiglia di uno dei 3 tag sopracitati. Da specificare che, le regole deprecate, nonostante identificassero Issues di tipo Blocker e Critical, non sono state considerate. Per concludere, il risultato del Quality Profile finale personalizzato è osservabile dalle Figure 2.3 e 2.4:

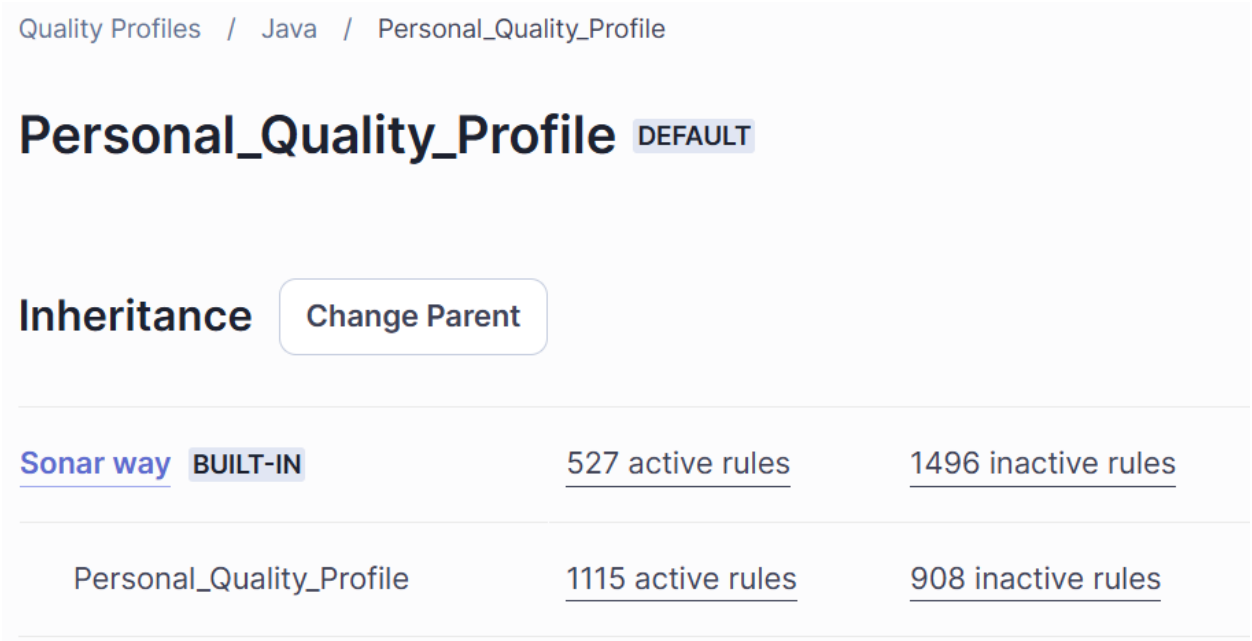



**Figura 2.3:** Quality Profile

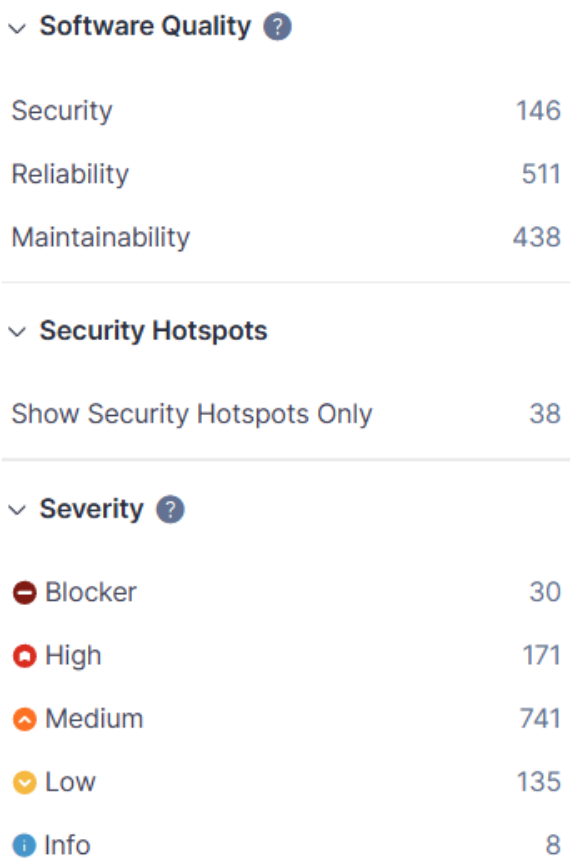



**Figura 2.4:** Regole Attive

### 2.3.5 Descrizioni API

SonarQube mette a disposizione delle API [9] per interagire con la piattaforma e per estrarre i risultati dell'analisi statica del codice. Queste API permettono di creare progetti, caricare codice, eseguire analisi, estrarre metriche e molto altro. Per tutte le chiamate è stato necessario inserire l'API Key di SonarQube come Header della chiamata, prendendolo tramite environment del sistema, in modo da autenticarsi, in questo modo:

```
HEADER: Final[dict[str, str]] = {
    "Authorization": f"Bearer {os.getenv("SONAR_LOCAL_API_TOKEN")}",
}
```

**Nota**: In questa sezione le response vengono sorvolate.

Sono state utilizzate le seguenti chiamate API (le più importanti), in Python, importando il modulo requests:

```
param = {
    "name": f"Progetto_{repository}",
    "project": f"Progetto_{repository}"
}
```

[9] https://next.sonarqube.com/sonarqube/web_api/api/

```
url = "http://localhost:9000/api/projects/create"
#per eliminare mettere delete al posto di create
response = requests.post(url, headers=HEADER, params=param)
```

**Listing 2.1:** Creazione ed Eliminazione di un progetto

```
param = {
      "component": f"Project_{project}",
      "metricKeys": "ncloc,bugs,vulnerabilities,code_smells,coverage,"
                    "test_success_density, test_failures,"
                    "duplicated_lines_density,"
                    "reliability_rating,sqale_rating,security_rating,"
                    "cognitive_complexity,"
                    "blocker_violations,critical_violations"
    }
url = "http://localhost:9000/api/measures/component"
response = requests.get(url, headers=HEADER, params=param)
```

**Listing 2.2:** Estrazione delle metriche

```
param = {
        "component": f"Project_{project}",
        "metricKeys":  "vulnerabilities",  #code_smells, bugs, ecc...
        "qualifiers": "FIL",
        "s": "metric",
        "metricSort": "vulnerabilities",  #code_smells, bugs, ecc...
        "ps": 10,
        "asc": "false"
    }
url = "http://localhost:9000/api/measures/component_tree"
response = requests.get(url, headers=HEADER, params=param)
```

**Listing 2.3:** Estrazione classi da un progetto in base a metriche

In questa chiamata si sta dicendo a SonarQube di restituire le 10 classi con più vulnerabilities ordinate in modo decrescente.

# Capitolo 3

# Approccio

L'obiettivo di questo capitolo è quello di presentare, descrivere e approfondire l'approccio adottato. Nello specifico, di come sono state strutturate e sviluppate la crew e il flow del sistema multi agente, al fine di fare un code refactoring efficace per tutti i progetti del dataset selezionato (vedere la Sezione 4.2). La Figura 3.1 di seguito raffigura in modo intuitivo lo schema progettato:

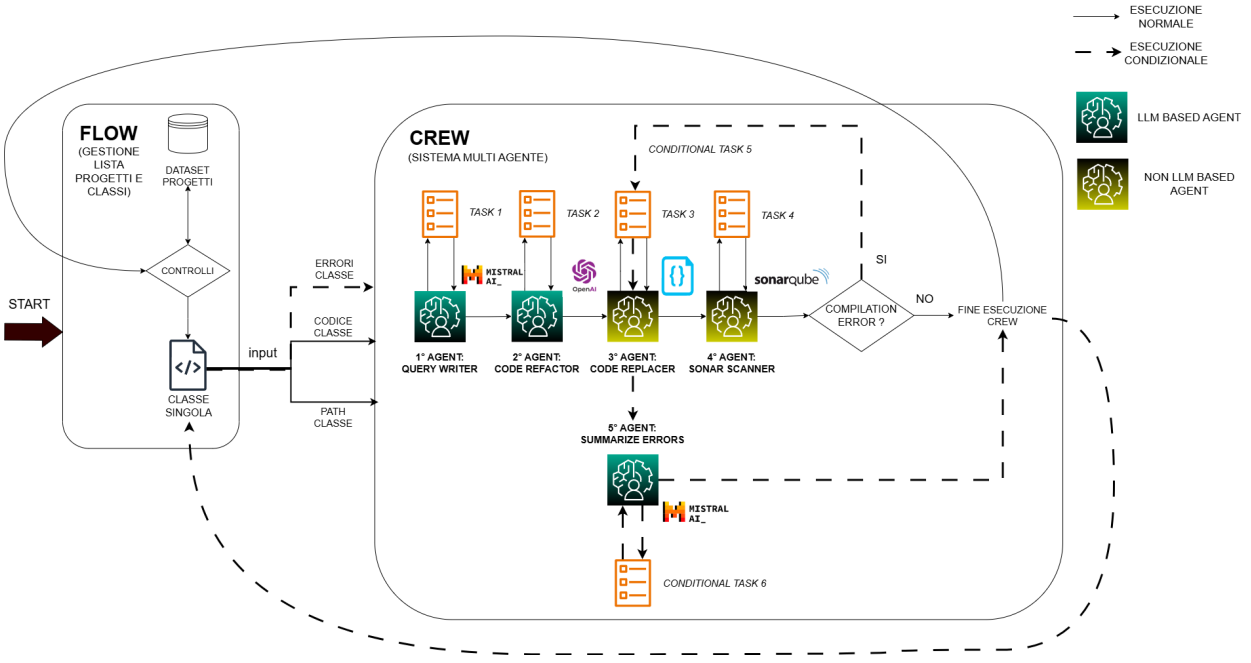



**Figura 3.1:** Schema Approccio

**1° step: creazione e sviluppo della crew** Per prima cosa è stato necessario pensare al vero e proprio sistema multi agente e, quindi, a come creare una crew che si adattasse alle esigenze e specifiche della ricerca.

Inizialmente lo schema degli agenti era stato pensato in una crew costituita da 4 agenti, che, in sequenza, eseguissero tali compiti: esecuzione test case al codice iniziale, generazione di prompt ottimale, code refactoring del codice, verifica attributi di qualità con SonarQube. A questo schema iniziale, dopo aver identificato problemi ed errori, sono state apportate delle variazioni. Infatti, quello pensato per eseguire i test case, è stato scartato, perchè sarà un altro agente (*sonar_agent*) più generico a farlo. Dopodichè, anche l'agente pensato inizialmente per verificare attributi di qualità è stato scartato. Questo perchè, alla fine del-

l'approccio, non bisogna verificare se il code refactoring abbia soddisfatto o meno dei vincoli preimpostati su attributi di qualità, ma bisogna solo osservare se ci sono stati dei miglioramenti o dei peggioramenti. A questo punto, si è riformulato lo schema degli agenti. Ovviamente sono rimasti i 2 agenti più importanti, ovvero quello con il compito di generare il prompt e quello di effettuare code refactoring vero e proprio (*query_writer* e *code_refactor*). Mentre, in sequenza, sono stati creati 3 agenti, con rispettivi compiti: fare code replace del codice (*code_replacer*), eseguire comando da terminale Maven e sonar-scanner (*sonar_agent*) e fare un riassunto degli eventuali errori restituiti dall' agente precedente (*errors_summarizer*). A tal proposito, verranno approfonditi nella Sezione 3.1 i 5 agenti, come sono stati costruiti e i task che devono eseguire.

**2° step: creazione e sviluppo del flow** Dopo aver iniziato il flusso di lavoro solamente con la crew, ci si è resi conto che bisognava modificare qualcosa. In particolare, l'acesso ai progetti, alle classi e ai path locali, non poteva essere fatto attraverso la sola crew. O comunque, anche se si è trovato il modo, l'agente interpretava spesso il codice e il path a modo suo, andando a invalidare l'intero flusso. Per cui la costruzione del flow, meglio approfondita nella Sezione 3.2, ha favorito questi aspetti, dove l'integrità dei dati è fondamentale in questo approccio di lavoro.

**3° step: esecuzione** Una volta testati e sviluppati in modo corretto crew e flow e una volta aver selezionato e ottenuto correttamente il dataset dei progetti, è stato possibile partire con le esecuzioni. Il flusso dell'approccio seguirà i particolari descritti nel sottocapitolo delle RQs nel prossimo Capitolo 4.

## 3.1 Sviluppo crew

La struttura della crew è formata da tre file: due Yaml file, in cui vengono descritti e formalizzati agenti e task e un Python file, in cui viene strutturata la crew vera e propria e gestiti i vari tools, LLM, callback e variabili. Come già preannunciato, sono stati sviluppati 5 agenti, ognuno dei quali con il compito di eseguire una o più task. Ogni parametro e descrizione di agenti e task sono stati testati e formulati attraverso l'Ablation Study. A differenza del seguente articolo [21], in questo lavoro di tesi, l'Ablation Study è avvenuto senza tenere traccia ogni volta dei risultati. Per cui, si è adottato un approccio non formale, in cui si eseguiva il codice, si osservavano i risultati senza salvarli, e si modificava di conseguenza una description di un task o una backstory di un agente.
In particolare:

**Query Writer:** questo agente è stato pensato come un maestro del prompt, che avrà il compito di creare un prompt completo ed efficace, passandolo come input al prossimo agente che effettuerà code refactoring. Serve quindi a istruire il prossimo agente, aiutandolo a fare un ottimo code refactoring basandosi sul prompt descrittivo che riguarda gli attributi di qualità generato da questo. Inoltre, questo agente (insieme al prossimo), cambierà tra RQ1-RQ2 e RQ3. Maggiori dettagli si avranno nel Capitolo 4.

Mentre per quanto riguarda il task che questo agente eseguirà (`task1`), si è costruito in modo tale da approfondire e perfezionare ancor di più gli attributi di qualità su cui si vuole fare particolare attenzione.

**Code Refactor:** questo è l'agente più importante ai fini del risultato. È colui responsabile del miglioramento o peggioramento degli attributi di qualità del codice. È stato pensato come uno sviluppatore Java con più di 50 anni di esperienza, specializzato nel refactoring e nello scovare errori inerenti agli attributi di qualità passati dal prompt generato dal precedente task.
Mentre il task che questo agente deve eseguire (`task2`), è stato costruito in modo meticoloso, cercando di elencare tutte le regole che un code refactoring efficace, di base, dovrebbe rispettare. Per cui, si è creato in modo tale che l'agente prevenga la struttura originale, e, quindi, mantenga nomi di classi e metodi, i costruttori, i package, la signature i parametri e i return dei metodi, la logica e le funzionalità, le licenze (se presenti) e le gerarchie tra classi (interfacce, classi astratte, ereditarietà). In più si sono dati degli input che riguardano il refactoring, in particolare di essere aggressivo, di seguire bene il prompt passato dal task precedente, non introdurre nessun tipo di errori, non effettuare casi di test, aggiungere commenti o header solo se necessario e di non ritornare nulla se non il codice refattorizzato.
Come contesto del task è stato inserito quello del task precedente. In questo modo ci si assicura che l'output dell'agente e task precedente passino come input all'agente e task attuale, in modo che Code Refactor segua le linee guida sulle metriche e attributi di qualità fornite dall task precedente.

**Code Replacer:** La scelta di implementare questo agente è dovuto al fatto principale che il codice refattorizzato dall'agente e task precedente deve essere effettivamente compilato ed eseguito insieme a tutto il resto del progetto. SonarQube, in automatico, non offre la possibilità di modificare del codice direttamente su piattaforma, per questo è stato necessario sostituire il vecchio codice con il nuovo codice all'interno del progetto situato in locale. L'agente è stato pensato per effettuare una sostituzione di codice locale in modo completamente sicuro. A tal proposito, essendo un lavoro chirurgico, in cui l'agente non può permettersi di sbagliare path e di costruirne uno nuovo o errato , perchè poi viene meno tutta l'esecuzione della crew, si è deciso di costruire un tool personalizzato per il task (`task3`) da lui eseguito, in cui si effettua la sostituzione del codice originale con quello nuovo appena refattorizzato dal task dell'agente precedente. Per fare ciò, al task viene dato il tool personalizzato, contenente lo snippet di sostituzione. In più, per evitare errori di path, al tool viene passato il path corretto come input, così che l'agente non possa sbagliarsi su quale codice sostituire e dove effettuare la sostituzione soprattutto. Il passaggio del path come input non avviene in automatico come un classico passaggio di parametri, ma bensi' sarà proprio l'agente a capire, grazie alla conoscenza fornita, che il path dovrà essere usato come input del tool per eseguire il task. Altrimenti, avrebbe potuto sbagliare il path di destinazione e, di conseguenza, la sostituzione del codice, invalidando i risultati.
Questo agente, inoltre, a differenza di tutti gli altri ha il compito di eseguire 2 task. Un task è quello appena descritto, in cui si sostituisce il vecchio codice con il nuovo. L'altro task è chiamato `conditional task5`, e si differenzia dall'altro task solo nel fatto che sostituisce il codice nuovo, questa volta, con il codice vecchio. Questo perchè, se nel prossimo task risulta che il codice refattorizzato ha errori in compilazione, allora bisogna ripristinare lo stato precedente. Per questo, si ripristina il vecchio codice. Il motivo è che se si fosse mantenuto il codice refattorizzato, che ha sollevato errori in compilazione, tutti gli altri refactoring sarebbero stati inefficaci, perchè si sarebbero mantenuti proprio quegli errori della classe precedentemente refattorizzata. Allora, per sorvolare questo problema, si riporta nel progetto la vecchia classe con il vecchio codice. L'esecuzione o meno del conditional task dipende da una funzione nel file Python, nella quale verrà confrontato un attributo. In particolare se il campo "valid" dell'attributo sarà False o None, allora il conditional task5 verrà eseguito, altrimenti no. Maggiori dettagli si avranno nella

spiegazione del prossimo agente.

**Sonar Agent:** È l'agente responsabile dell'esecuzione dei comandi da terminale post refactoring e post code replace. In particolare, della compilazione, pulizia e report dei test del progetto tramite comandi Maven e dello scanner con SonarQube tramite sonar-scanner. Inoltre, per la RQ3, è responsabile della chiamata API a SonarQube per la restituzione del valore di una metrica in particolare, così che si usi tale valore come oggetto di re-iterazione di una classe. (Maggiori dettagli implementativi riguardo ciò, verranno approfonditi più tardi. ) Anche in questo caso, come per il task precedente, è stato costruito un tool personalizzato per il task (`task4`) che questo agente deve eseguire Il motivo è che l'agente, in autonomia, non riesce ad eseguire comandi da terminale e a fare chiamate API. Per cui, il tool creato, che alla fine non è altro che un metodo deterministico, serve proprio a guidare l'agente su quello che deve fare, cosa che prima, da solo, non sarebbe riuscito a fare. In particolare nel tool viene eseguitio il comando da terminale già descritto 4.2 e verranno catturati gli errori in compilazione, se presenti, del nuovo codice refattorizzato. In più, verrà effettuata la chiamata API a SonarQube per poter ottenere il valore di una metrica per un progetto, utile per la RQ3. Questo agente e rispettivamente il suo task, sono i responsabili delle iterazioni e dei task successivi. Questo perchè se il comando da terminale restituisce errori in compilazione, significa che il nuovo codice generato non funziona, e che, quindi, bisogna tornare all'agente Code Replacer che, come già accennato, rieffettua un code replace, in particolare ripristina il vecchio codice, eseguendo la conditional task5 già citata. Tutto ciò che riguarda l'esecuzione o meno dei conditional task, parte proprio dal task eseguito da questo agente, poichè, nel file Python, si gestiscono gli errori restituiti, eventualmente, da Maven, andandoli a catturare e a salvare in un oggetto Pydantic. Inoltre, è l'unico task che restituirà una *callback*. Nella funzione di callback si salverà il risultato restituito dal tool, sottoforma di *output pydantic* (e, quindi, se il refactoring è valido, eventuali errori, e valore di una metrica nel caso della RQ3). Questo perchè durante l'iterazione della crew non è possibile accedere direttamente all'output pydantic. Per salvarlo correttamente durante l'esecuzione, allora, si è salvato tale output pydantic in un attributo d'istanza della crew, chiamato *refactoring_output*. Questo attributo di istanza verrà analizzato, nelle condizioni dei conditional task, e da qui si sceglierà se proseguire con l'iterazione fra gli agenti. In particolare se concludere l'eseuzione della crew per la classe attuale perchè il refactoring non ha restituito errori in compilazione, o se eseguire i 2 conditional task, uno che rieffettua code replace, come trattato nella spiegazione dell'agente Code Replacer3.1, e un altro che verrà spiegato nel prossimo agente. Nel caso in cui il comando da terminale non restituisce errori in compilazione, in ogni caso si concluderà l'esecuzione della crew per quella determinata classe e si ritornerà nel Flow, dove verranno fatte nuove considerazioni, in base alla RQ. Non si approfondirà ancor di più il tool e di come avviene il passaggio fra i vari dati nello spcifico, per maggiori informazioni si visiti l'url GitHub presente nel Capitolo 1.

**Errors Summarizer:** È, potenzialmente, l'ultimo agente della crew. È pensato come un esperto analista e responsabile in riassunti che va dritto al punto. In questo caso, è il responsabile del riassunto degli errori restituiti, eventualmente, dall'agente e task precedenti (task4). Il task da lui eseguito (`conditional task6`) è costruito come il conditional task5. Per cui questo conditional task6, brevemente, segue il flusso impostato: se task4 ha restituito errori di compilazione nella callback e quindi il campo "valid" Pydantic è False o None, allora il conditional task6 verrà eseguito, altrimenti no. Tale controllo viene effettuato in una funzione apposita che determinerà l'esecuzione o meno del conditional task6.

Per concludere questo paragrafo, è bene specificare che alla crew, ogni iterazione, viene dato in input un dizionario, contenente: codice della classe da refattorizzare, path della classe, riassunto degli errori. Grazie a questi input, l'intera crew aveva la possibilità di utilizzare questi valori, grazie alla tecnica del templating, ovvero quella di inserire la chiave del dizionario tramite parentesi graffe, in modo che la crew potesse accederne al contenuto. L'inserimento di tali valori, come si poteva già osservare nelle rapide descrizioni dei vari task creati, è stato pensato in modo tale da mantenere l'integrità dei dati, accedendo senza errori e con massima sicurezza a valori di classe, path ed errori in compilazione giusti e precisi. Ulteriori approfondimenti su come verranno personalizzati gli agenti e i task, verranno descritti nella Sezione 4.1 del Capitolo 4. Inoltre, sono riportati, nella Tabella 3.1, il riassunto sui parametri opzionali per ogni task, precedentemente introdotti testualmente:

| Task | Tool personalizzato | Output Pydantic | Condition | Callback |
|---|---|---|---|---|
| task1 | ✗ | ✗ | ✗ | ✗ |
| task2 | ✗ | ✗ | ✗ | ✗ |
| task3 | ✓ | ✗ | ✗ | ✗ |
| task4 | ✓ | ✓ | ✗ | ✓ |
| conditional_task5 | ✓ | ✗ | ✓ | ✗ |
| conditional_task6 | ✗ | ✓ | ✓ | ✗ |

**Tabella 3.1:** Tabella riassuntiva parametri opzionali tasks

## 3.2 Sviluppo Flow

Si è scelti di implementare un flow di tipo Strutturato, come già visto nel Capitolo precedente 2.2.3. Il flow finale è rappresentato dalla Figura 3.2, costruito grazie al metodo plot() di CrewAI:

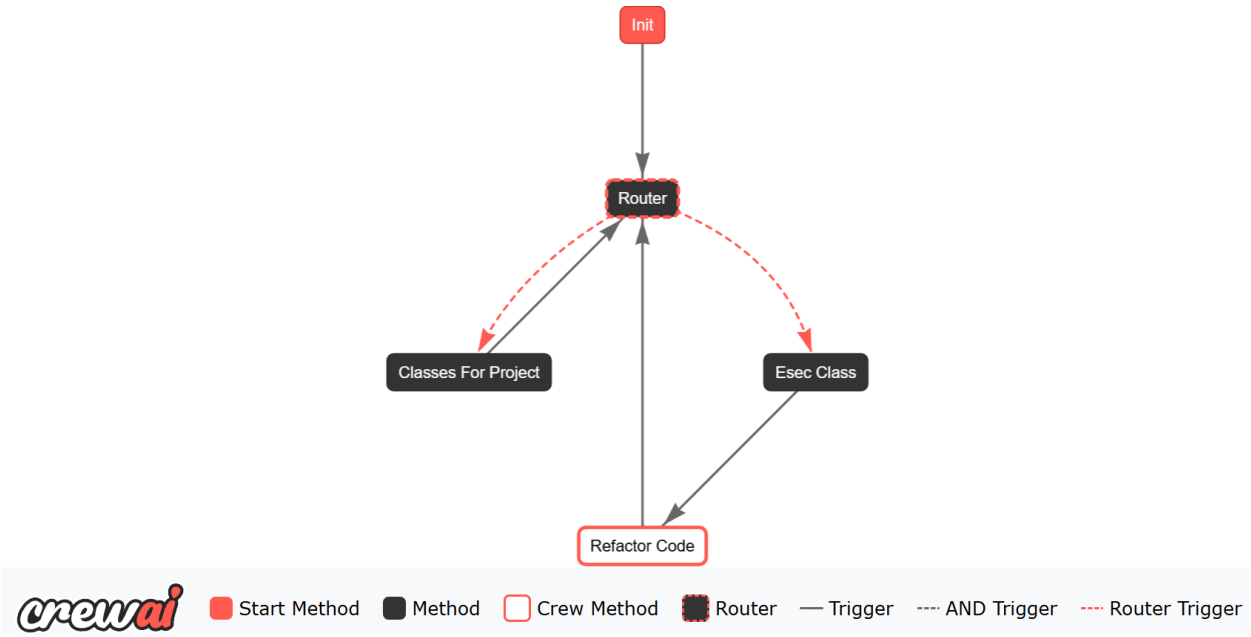



**Figura 3.2:** Struttura Flow

Inanzitutto c'è da parlare delle variabili a cui i metodi del flow potevano accedere e, eventualmente, utilizzare. È stato necessario creare una classe che estende BaseModel, ovvero la classe di partenza della

struttura Pydantic di Python. Sono state dichiarate le seguenti variabili:

```
ATTEMPTS_MAX: int = 3                #attempt max for every class
attempts_tot: int = 0                #total attempts in end execution
time_for_project: List[float] = []     #array for keep projects time

class ExampleFlow(BaseModel):
    directory: str = ""      #directory where execution takes place
    project_list: List[str] = []        #list of projects
    current_project: Optional[int] = 0         #indicates which project is
       being processed
    classes: List[dict] = []         #list of classes of a project
    current_class: Optional[int] = 0        #indicates which class of the
       project is being refactored
    path_class: str = ""       #path of the class to pass to the Crew for code
        replacement and to run sonar-scanner
    code_class: str = ""       #code of the class to pass to the Crew for
       refactoring
    errors: Optional[str] = ""       #errors (compilation or others) to pass
       to the Crew to guide the agent to avoid committing them
    is_validate: Optional[bool] = True       #flag that checks if the terminal
        command for a class returned Build Success or Build Failure
    attempts: Optional[int] = 0       #counter to keep track of the number of
       attempts on a single class
    value_metric_pre: Optional[int] = 0   #FOR RQ3
    value_metric_post: Optional[int] = 0   #FOR RQ3
    project_start_times: List[float] = []    #list of start execution time for
        every project
```

in cui le prime 4 sono attributi non di classe, usate per tenere traccia dei tempi di esecuzione per progetto e tentaivi totali, per dichiarare e assegnare i tentativi massimi per classe e per inizializzare il numero di classi da refattorizzatel, prese in modo randomico dal metodo *random.sample* di Python. Mentre tutte le altre variabili sono utilizzate per gestire e controllare il processo di refactoring all'interno e all'esterno della crew. In particolare, ogni metodo o step del flow può accedere a tali variabili attraverso *self.state*. Esso è un dizionario mutabile e condiviso dagli step del flow e ogni attributo che si imposta su self.state viene mantenuto nel contesto della run del flow. CrewAI crea e inietta direttamente questo oggetto proprio per facilitare la condivisione e mantenere il contesto delle variabili all'interno dell'esecuzione del flow. A questo punto, bisogna parlare e approfondire gli step del flow.

**Init:** È il metodo di partenza, come identificato dal decorator @start. Qui vengono inizializzate alcune variabili di base, come la lista dei progetti, la lista contenente i tempi di partenza di esecuzione di ogni progetto e la lista contenente i tempi totali di esecuzione per progetto. Questo sarà sempre il primo step ad essere eseguito e non verrà più richiamato all'interno del flow.

**Router:** È il metodo responsabile dell'instradamento del flow, come identificato dal decorator @router. In particolare il decorator è questo: **@router(or_("init", "classes_for_project", "refactor_code"))**, ciò

significa che questo metodo verrà eseguito dopo gli step presenti nella condizione logica or_ . Al suo interno avviene principalmente la scelta fra il caricamento delle classi e l'esecuzione della singola classe, con rispettivi percorsi.
Nella prima iterazione, è lo step subito chiamato dopo Init. Dopodichè, invece, verrà chiamato dagli step di "Classes For Projects" e "Refactored Code" che saranno meglio spiegati successivamente. In ogni caso, il codice interno non cambia, e si effettuano vari controlli. Nel dettaglio, se la lista di classi (*self.state.classes*) è vuota, questo significa che non si hanno ancora classi su cui fare refactoring e che, quindi, bisogna caricare la lista. In tal caso, questo step si occuperà di instradare il percorso verso lo step chiamato Classes For Projects. Invece, se la lista di classi non è vuota, si effettua il controllo sul numero di classi ancora da refattorizzare. In tal caso, si verifica l'indice *self.state.current_class* all'interno della lista di classi e il Router decide cosa fare. Se non sono ancora state eseguite tutte le classi all'interno della lista di classi sono state eseguite, allora questo step instraderà il percorso verso un altro step chiamato Esec Class. In caso contrario, significa che sono state eseguite tutte le classi per un determinato progetto e che, quindi, è possibile calcolare il tempo di esecuzione di tutta la crew per quel progetto e bisogna riniziare il flow, questa volta con un nuovo progetto. Quindi andando ad aumentare *self.state.current_projects*, se ci sono ancora progetti da eseguire, terminando il flow altrimenti. Per maggiori dettagli implementativi, si guardi il seguente codice:

```
@router(or\_("init", "classes_for_project", "refactor_code"))
  def router(self, _=None):
      """
      The router checks the current state and returns the name of the next
         method to execute.
      """

      #If I haven't loaded the classes for a project yet
      if not self.state.classes:
          return "project_roadmap"


      #If I still have remaining classes for this project
      if self.state.current_class < len(self.state.classes):
          return "class_roadmap"


      #I have processed all the classes of this project:
      # move on to the next project (if it exists) or terminate
      time_for_project[self.state.current_project] = time.time() - self.
         state.project_start_times[
          self.state.current_project]
      self.state.current_project += 1
      if self.state.current_project < len(self.state.project_list):
          # resetto le classi per il nuovo progetto
          self.state.classes = []
          self.state.current_class = 0
          return "project_roadmap"
      else:
```

```
            print("All projects have been processed")
```

Questo è l'unico step con la quale il flow può terminare la sua esecuzione.

**Classes For Projects:** Questo step, identificato da **@listen("project_roadmap")** ed eseguito sempre e solo dopo Router, serve principalmente a gestire i singoli progetti. In particolare, si fa partire il tempo di esecuzione per il singolo progetto e dopodichè si recuperano le classi su cui fare refactoring di quel progetto. La metodologia con cui le classi vengono selezionate non verranno ulteriormente approfondire. Basti sapere che per la RQ2 vengono selezionate sempre 10 classi Java, in modo randomico. Mentre per la RQ3, vengono selezionate 10 classi, ordinate in base al valore della metrica scelta (non è detto che ci siano sempre 10 classi in questo caso). La response di questo metodo, fornirà la lista di classi del progetto, caricandole su *self.state.classes* e si ritornerà al Router (che ha "classes_for_projects" all'interno della condizione logica or_ ) che gestirà questa cosa, come già descritto.

**Esec Class:** In questo metodo del flow, identificato con **@listen("class_roadmap")** ed eseguito sempre e solo dopo Router, ha il compito di gestire le singole classi presenti nella lista di classi di un determinato progetto. In particolare, di tenere traccia della classe su cui si vuole fare refactoring e, una volta, identificata attraverso indice *self.state.current_class*, ottenere il codice di quella classe e, per la RQ3, ottenere il valore di una metrica preselezionata per quella specifica classe, attraverso chiamate API a SonarQube. A questo punto è necessario trovare il path della classe, così da darla in input alla crew, per non incorrere errori. Per questo, tale step ha anche il compito di trovare e gestire il path della classe da refattorizzare. Da specificare, però, che tutto ciò avviene se e solo se il numero di tentativi attuali per una classe è minore del numero di tentativi totali per classe ovvero 3. Se così non fosse, si evita tutta la gestione della classe e si va avanti, così da ottimizzare l'efficienza ed evitare esecuzione di codice che poi non porterà a nulla. A questo punto lo step è terminato e si passerà al prossimo.

**Refactor Code** È l'unico Crew Method del flow. Questo perchè al suo interno ha la vera e propria chiamata alla crew sopra descritta. È identificata da **@listen("esec_class")**, per cui viene eseguita sempre e solo dopo Esec Class. In particolare, come prima condizione bisogna verificare, come per Esec Class, se il numero di tentativi attuale per una classe è minore del numero di tentativi totali per classe. Questo perchè a priori, il flow non sa che momento dell'esecuzione sia e che, quindi, potrebbe darsi che in un determinato momento si entri in questo step dopo svariati tentativi andati a vuoto nel cercare di refattorizzate una classe. Per cui, se questa condizione viene meno, così come per Esec Class, tutto il codice interno non verrà eseguito, ma a differenza di prima, in questo step si gestisce tale condizione. Una volta raggiunti i tentativi massimi per classe, si poteva passare alla prossima classe, aumentando il contatore *self.state.current_class* nella lista di classi e riazzerando rispettivamente: *self.state.is_validate*, *self.state.errors = ""*, *self.state.attempts = 0*. Da specificare che, non è stato possibile fare questo controllo nello step precedente Esec Class in modo tale da non entrare mai in questo step. Questo perchè in questo modo si sarebbe dovuto mettere "esec Class" nella condizione or_ del Router, ma questo dava problemi al ciclo di vita corretto del flow. Perciò si è fatto un primo controllo in Esec Class, poi si è passati in ogni caso a questo step Refactor Code e solo in questo caso si è tornati allo step Router. Così che si è mantenuto il flow corretto dove Refactor Code viene eseguito solo dopo Esec Class e che Router non venga mai eseguito subito dopo Esec Class. Proseguendo, se la condizione dei tentativi viene rispettata, allora il codice interno di Refacor Code viene eseguito. Nello specifico si fa partite l'esecuzione della crew precedentemente creata

e sviluppata, dandogli in input: codice classe, path classe, eventuali errori provenienti da esecuzioni precedenti della crew. Dopodichè, a fine esecuzione della crew, sono stati presi i risultati restituiti dall'agente Sonar Scanner, oppure dall'agente Errors Summarizer, se passati per conditional tasks. Tali risultati hanno dato la possibilità di continuare a gestire il flusso e l'approccio fuori dalla crew, ma sempre all'interno del flow. Nel dettaglio, se il refactoring non aveva restituito errori in compilazione, allora nel caso della RQ2 si andava avanti, mentre nel caso della RQ3 si effettuava un ulteriore controllo, ovvero quello di osservare se il valore della metrica preselezionata è stata effettivamente migliorata oppure no. In caso positivo, si andava avanti. Passati questi controlli, si poteva dare per buono il code refactoring e, di conseguenza, si poteva passare alla prossima classe, aumentando il contatore *self.state.current_class* nella lista di classi e riazzerando le variabili già citate sopra. A questo punto, quando l'esecuzione di questo step è terminata, si ritorna a Router che continuerà a gestire il processo del flow.

Per visionare l'intero codice e per ulteriori dettagli implementativi su crew e flow si consulti l'URL GitHub presente alla fine del Capitolo 1.

# Capitolo 4

# Validazione

In questo capitolo verranno presentate le Research Question, nella Sezione 4.1, a cui questo lavoro di tesi vuole rispondere e verrà approfondita e spiegata la scelta del dataset di progetti, nella Sezione 4.2.

## 4.1 RQs

***RQ1: Come impatta il MAS per fare code refactoring su progetti strutturati Apache e su progetti studenteschi ?***

Innanzitutto è stato necessario scegliere quale modelli utilizzare per gli agenti del sistema multi agente. La scelta del modello (o dei modelli) si è basata principalmente su questi fattori: velocità, compatibilità con LiteLLM (wrapper di CrewAI che gestisce e unifica le chiamate ai vari LLM in base a determinati providers [1]), capacità di refactoring, prezzo, numero di token in input e in output, eventuali limiti. Per prima cosa, ne sono stati selezionati diversi, e testati in modo rapido per scoprire le prime differenze. Per fare ciò si è utilizzato l'approccio diverse volte sul dataset di progetti selezionato (vedere Sezione 4.2), cambiando ogni volta il modello e osservando i risultati. I modelli testati sono stati diversi, ma alla fine si è scelto di utilizzarne 2. In particolare, per l'agente Code Replacer si è utilizzato GPT-4o-mini. La motivazione è stata presa in funzione delle sue capacità e del rapporto qualità-prezzo. Come anche descritto da questi siti *https://apxml.com/leaderboards/coding-llms* e *https://blog.promptlayer.com/best-llms-for-coding/*, GPT-4o-mini offre ottimi risultati in termini di AI Assistant Code e di Competitive Coding, offre un contesto di token molto buono per i progetti che bisogna refattorizzare (128k) e ha un costo molto contenuto ($0,10 per milione di token in input e $0,60 per milione di token in output). Per cui è stato considerato come il modello migliore da usare, in particolare per questo agente che è il maggior responsabile del code refactoring vero e proprio. Un'altra valida alternativa considerata, ma non utilizzata perchè incompatibile con LiteLLM e non presente in HuggingFace e OpenRouter, era l'uso di Deepseek Coder V2. Mentre per i restanti 4 agenti, che svolgono compiti più descrittivi e/o deterministici e con minor ragionamento, si è scelto di utilizzare Mistral-Medium, un buon modello, che offre un buon compromesso tra velocità e efficacia nelle risposte e soprattutto gratuito (rientrando nei limiti imposti da Mistral, che offre 1 milione di token input/output gratuiti al mese). Di seguito, la Tabella 4.1 riassume i modelli utilizzati obbligatoriamente per ogni agente del sistema, nonostante, come già accennato nel Capitolo 3, gli agenti Sonar Agent e Code Replacer effettuino codice procedurale senza alcun ragionamento:

[1] `https://docs.litellm.ai/docs/providers`

| Agenti | LLM |
|---|---|
| query_writer | Mistral-Medium |
| code_refactor | GPT-4o-mini |
| code_replacer | Mistral-Medium |
| sonar_agent | Mistral-Medium |
| errors_summarizer | Mistral-Medium |

**Tabella 4.1:** Tabella modelli agenti

Dopodichè è stato necessario decidere il numero di classi per ogni progetto da dare in input all'approccio. La decisione è stata presa cercando di mantenere sempre un tempo di esecuzione ragionevole e mai troppo elevato, testando e cercando di capire, nelle prime esecuzioni di prova, quanto potesse essere il tempo totale di esecuzione e osservando quante modifiche al progetto e agli attributi di qualità venissero effettuate. La scelta finale è ricaduta su 5 classi, prese in modo randomico, che non fossero utilizzate per i test e che non avessero estensioni diverse da '.java'. La motivazione per cui le classi sono state prese in modo casuale e non in modo deterministico è perchè non ci si è voluti soffermare su un insieme o sottoinsieme di classi in particolare. Per questo, ci penserà la RQ3, che verrà fra poco approfondita.

***RQ2: Come variano i risultati aumentando il numero di classi da refattorizzare ?***
Considerando che nella RQ1 si sono refattorizzate 5 classi randomiche per progetto, in questa Research Question si vuole osservare la differenza tra l'approccio applicato a 5 classi e lo stesso approccio applicato a 10 classi randomiche e, quindi, di come cambiano i risultati modificando il numero di classi, raddoppiandolo. Questo dopo aver osservato i risultati della RQ1 e aver analizzato codice post refactoring e tempi di esecuzioni. Le classi, come prima, vengono prese sempre in modo randomico.

Nelle Tabelle 4.2 e 4.3 di seguito, inoltre, viene riassunto la configurazione degli agenti e dei task dell'approccio, utilizzata sia nella RQ1 che nella RQ2, testata tramite Ablation Study, come già citato nel Capitolo 3, nella Sezione 3.1:

| Agente | Ruolo | Goal | Backsory |
|---|---|---|---|
| query_writer | Esperto di Static Analysis e Prompt Generation | Scrivere query intelligenti per rilevare e risolvere problematiche in codice Java, usando metriche Sonar (bugs, vulnerabilities, code smells) e plugin (Checkstyle, PMD, FindBugs). | Forte integrazione con SonarQube, focus su qualità del codice, architettura pulita, rilevamento di antipattern, regole personalizzate, supporto continuo per mantenere affidabilità, sicurezza e manutenibilità. |
| code_refactor | Esperto di Refactoring Java Sicuro | Migliorare il codice seguendo il prompt generato da query_writer, mantenendo firme, pacchetti e struttura originale. | Refactoring avanzato, ottimizzazione performance (JVM, GC, SQL), sicurezza (OWASP), qualità del codice (PMD, Checkstyle, SonarQube), riduzione debito tecnico, mentoring tecnico. |
| code_replacer | Writer Sicuro del Codice | Scrivere in sicurezza il codice refactorizzato nel file originale con codifica UTF-8. | Gestione sicura della scrittura su filesystem, salvaguardia della struttura del codice. |
| sonar_agent | Esecutore Analisi Sonar | Eseguire l'analisi statica tramite sonar-scanner nel percorso specificato, senza mostrare errori. | Esperto nell'esecuzione di comandi Maven + SonarQube da terminale, integrazione CI/CD. |
| errors_summarizer | Analista di Errori Statici | Riassumere in modo chiaro e conciso gli errori provenienti dal terminale. | Analisi testuale avanzata, sintesi precisa dei messaggi d'errore, rimozione del superfluo, chiarezza comunicativa. |

**Tabella 4.2:** Configurazioni sintetiche dei 5 agenti per RQ1 e RQ2

| Task | Descrizione | Agente Responsabile | Expected Output |
|---|---|---|---|
| task1 | Analizza codice Java dato in input per generare un prompt avanzato focalizzato su sicurezza, qualità e regole statiche (SonarQube, Checkstyle, PMD, FindBugs). | query_writer | Crea un prompt dettagliato per l'analisi automatica di sicurezza del codice. Integra plugin statici e suggerisce remediation. |
| task2 | Esegue refactoring aggressivo del codice Java usando come guida solo il prompt di task1. Non altera struttura, firme, import o comportamenti. | code_refactor | Migliora sicurezza, mantenibilità e qualità seguendo rigorosamente le regole definite nel prompt. Nessun test, nessuna firma cambiata. |
| task3 | Sovrascrive il file originale con il codice refattorizzato, mantenendo codifica UTF-8. | code_replacer | Sostituisce il codice nel file specificato senza restituire il contenuto. Scrittura sicura su filesystem. |
| task4 | Esegue il sonar-scanner da terminale sul file refattorizzato. Non ritorna errori né codice. | sonar_agent | Lancia sonar-scanner con Maven, valutando il codice aggiornato secondo i Quality Gates. |
| conditional_task5 | Ripristina il codice originale nel file, se il refactoring ha introdotto errori. | code_replacer | Annulla il refactoring, reinserendo il codice originale nel file target. Fallback di sicurezza. |
| conditional_task6 | Riceve errori da task4 e genera un riassunto testuale chiaro in formato JSON con i problemi rilevati. | errors_summarizer | Ritorna un JSON con "valid" da task4 e una lista "errors" riassunta (tipologia errore + suggerimento). |

**Tabella 4.3:** Configurazione sintetica delle tasks per RQ1 e RQ2

***RQ3: Quanto è buono e affidabile l'approccio per fare code refactoring quando si specifica una metrica di qualità ?***

Allo stesso modo della RQ2, si vuole applicare il sistema multi agente al dataset di progetti, con una sottile ma importante differenza. Prima, nelle RQ1 e RQ2, ci si concentrava sul miglioramento (o peggiormento) generale del progetto, prendendo in input classi in modo casuale. Per cui gli agenti avevano il compito di rilevare la maggior parte degli errori e problemi senza soffermarsi su qualcuno in particolare. Mentre in questa RQ3, si vuole dare al sistema multi agente un compito specifico, ovvero quello di fare code refactoring delle classi progetti specificando la particolare metrica su cui ci si vuole concentrare. In questo modo, si dovrebbe notare la differenza tra le RQs precedenti, in cui si applica l'approccio in maniera generale, e la RQ3, in cui si da particolare attenzione su delle metriche specifiche, così che l'approccio possa rilevare errori inerenti principalmente alla metrica specificata. Le metriche in questione saranno quelle che contribuiscono al rating di Security, Reliability e Maintainability, ovvero rispettivamente, come già specificato, vulnerabilities, bugs e code smells. Inoltre, le classi da refattorizzare non sono più prese in modo randomico, ma verranno selezionate le 5 classi, in ordine decrescente, con il numero più alto nelle rispettive metriche. È importante far notare, per concludere, che l'approccio, questa volta, non rieffettuerà il refactoring solo se si avranno errori di compilazione, per un massimo di 3 tentativi, ma lo rieffettuerà anche solo se non migliorerà la metrica specificata. Per cui si salverà il valore della metrica del progetto prima del refactoring, l'approccio farà refactoring, e se la classe refattorizzata non avrà errori di compilazione e il valore della metrica post refactoring sarà migliorato, migliorando il numero totale del progetto, allora la classe sarà data per buona, altrimenti l'approccio ritenterà il refactoring su quella classe. Quindi l'idea è quella di fare un refactoring che forzi il miglioramento di una metrica, trascurando le altre. Il ragionamento è il seguente della Figura 4.1:

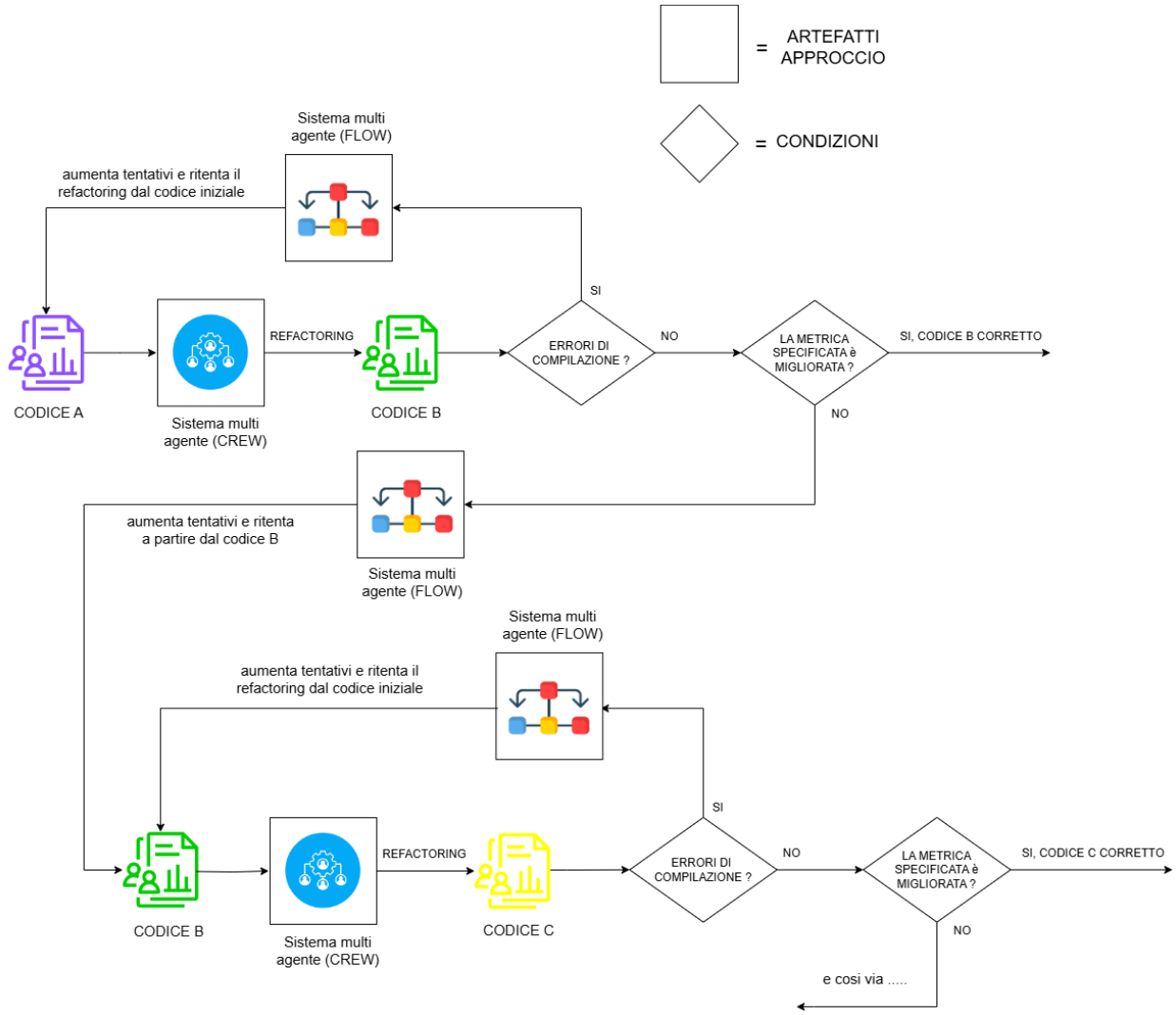



**Figura 4.1:** Funzionamento RQ3

Di conseguenza, sono state modificate anche alcune configurazioni di due agenti e di un task, come rappresentato nelle Tabelle 4.4 e 4.5 per le vulnerabilities, nelle Tabelle 4.6 e 4.7 per i bugs e nelle Tabelle 4.8 e 4.9 per i code smells, per poter indirizare il sistema multi agente verso una particolare metrica:

| Agente | Ruolo | Obiettivo | Backstory |
|---|---|---|---|
| `query_writer` | Esperto di analisi statica orientata alla sicurezza e generatore di prompt | Scrive prompt mirati per rilevare e prioritizzare vulnerabilità nel codice Java, sfruttando SonarQube, OWASP Top 10 e plugin statici come SpotBugs, PMD e Checkstyle. | Creato per rispondere a violazioni di sicurezza dovute a una scarsa analisi statica. Addestrato su migliaia di repository Java vulnerabili. Ottimizzato per sfruttare SonarQube in chiave sicurezza e guidare altri agenti verso una remediation sicura e conforme. |
| `code_refactor` | Esperto senior di refactoring sicuro in Java | Applica modifiche sicure al codice Java mantenendo struttura, logica e API originali, seguendo i prompt prodotti dall'agente di analisi. | Ingegnere Java con oltre 50 anni di esperienza, specializzato in hardening di sistemi legacy e moderni. Conosce profondamente vulnerabilità comuni (es. injection, deserializzazione insicura) e tool statici come SonarQube, PMD e SpotBugs. Focalizzato sulla sicurezza senza compromettere compatibilità o performance. |

**Tabella 4.4:** Configurazioni sintetiche degli agenti `query_writer` e `code_refactor` per RQ3 (vulnerabilities)

| Task | Descrizione | Agente responsabile | Expected Output |
|---|---|---|---|
| `task1` | Genera un prompt dettagliato per istruire un altro agente su come condurre un'analisi statica di sicurezza su codice Java, sfruttando SonarQube, SpotBugs (con FindSecurityBugs), PMD e Checkstyle, con mappatura OWASP e senza modificare direttamente il codice. | `query_writer` | Fornire istruzioni testuali precise per identificare vulnerabilità, hotspot e suggerire strategie di remediation sicura, senza alterare il codice. |

**Tabella 4.5:** Configurazione sintetica di `task1` per RQ3 (vulnerabilities)

| Agente | Ruolo | Obiettivo | Backstory |
|---|---|---|---|
| `query_writer` | Esperto in rilevazione di bug funzionali e generazione prompt | Genera prompt precisi per guidare AI o tool a individuare e prioritizzare bug funzionali e errori logici in Java usando SonarQube, PMD, Checkstyle e SpotBugs. | Sviluppato per sistemi ad alta disponibilità (banche, telecom, automotive), esperto nel guidare AI a rilevare errori runtime, bug di logica e cattivo uso di costrutti Java, focalizzato su correttezza e affidabilità. |
| `code_refactor` | Esperto senior in correzione bug e refactoring Java | Applica refactoring mirati per eliminare bug di logica e errori runtime mantenendo comportamento e contratti invariati, evitando regressioni. | Ingegnere senior con decenni di esperienza in sistemi critici (banche, trading, telecom), specializzato in diagnosi e correzione precisa di bug complessi usando SonarQube, PMD, SpotBugs e Checkstyle, intervenendo solo su ciò che è necessario. |

**Tabella 4.6:** Configurazioni sintetiche degli agenti `query_writer` e `code_refactor` per RQ3 (bugs)

| Task | Descrizione | Agente responsabile | Expected Output |
|---|---|---|---|
| `task1` | Genera un prompt chiaro e operativo che istruisca un altro agente AI a eseguire analisi completa con SonarQube, PMD, SpotBugs e Checkstyle, focalizzandosi esclusivamente su bug di logica, rischi di null pointer, errori di threading, cicli infiniti, uso scorretto di API e codice irraggiungibile, richiedendo l'identificazione e suggerimenti di correzione senza modificare il codice. | `query_writer` | Prompt unico e conciso che indirizzi a usare gli strumenti indicati per individuare bug, con istruzioni precise per la rilevazione e suggerimenti di correzione, senza intervenire direttamente sul codice. |

**Tabella 4.7:** Configurazione sintetica di `task1` per RQ3 (bugs)

| Agente | Ruolo | Obiettivo | Backstory |
|---|---|---|---|
| `query_writer` | Esperto in analisi statica focalizzata su code smells e generazione prompt | Genera prompt precisi per guidare AI o tool a individuare e prioritizzare code smells in Java usando SonarQube, Checkstyle, PMD e SpotBugs, suggerendo remediation efficaci. | Addestrato su migliaia di repository Java per identificare problemi di manutenibilità e complessità, conosce bene SonarQube e plugin Java statici, produce prompt per refactoring pulito e sicuro. |
| `code_refactor` | Esperto senior in refactoring di code smells Java | Applica refactoring mirati per eliminare code smells migliorando leggibilità, semplicità e manutenibilità, mantenendo comportamento e API invariati. | Sviluppatore Java con decenni di esperienza, specializzato in refactoring di sistemi legacy, esperto in best practice, principi SOLID e uso di tool statici per mantenere alta qualità. |

**Tabella 4.8:** Configurazioni sintetiche degli agenti `query_writer` e `code_refactor` per RQ3 (code smells)

| Task | Descrizione | Agente responsabile | Expected Output |
|---|---|---|---|
| `task1` | Genera un prompt dettagliato per istruire un altro agente ad eseguire scansioni con SonarQube, Checkstyle, PMD e SpotBugs per identificare code smells come metodi lunghi, classi grandi, codice duplicato, naming improprio e complessità elevata. Fornisci indicazioni chiare e precise per la priorità delle problematiche e suggerimenti di refactoring mirati a migliorare la manutenibilità senza modificare direttamente il codice. | `query_writer` | Un prompt chiaro, conciso e strutturato che guidi l'analisi statica focalizzata su code smells con gli strumenti indicati, suggerendo azioni concrete di refactoring e manutenzione. |

**Tabella 4.9:** Configurazione sintetica di `task1` per RQ3 (code smells)

## 4.2 Dataset Progetti

Per tutte le RQs è stato necessario trovare e collezionare progetti open-source scritti in Java [28] [15] e gestiti con Maven, per costruire un dataset di progetti, ognuno dei quali caricato su SonarQube per l'analisi statica del codice, al fine di cercare di migliorare gli attributi di qualità. Per prima cosa è stato necessario installare SonarQube in locale, in modo da poterlo utilizzare per l'analisi statica del codice. Dopodichè, è stato necessario decidere quali progetti Java open-source scegliere e come collezionarli. Si è scelti di utilizzare GitHub, come strumento di clonazione dei progetti, in quanto è una delle piattaforme più utilizzate per la gestione dei progetti open-source. In particolare, sono stati scelti i progetti che hanno come linguaggio di programmazione Java, che sono gestiti con Maven e che non risultano troppo piccoli a livello

di linee di codice, in modo da avere un dataset di progetti significativo. La scelta definitiva è ricaduta su progetti sviluppati da Apache Foundation e su progetti sviluppati da studenti ai primi anni di Università. Questo per poter osservare la differenza a livello di risultati tra progetti commerciali, aggiornati e ben strutturati di Apahe e quelli creati da studenti con poca esperienza a livello di progettazione software poichè, si suppone, questi ultimi contengano più problematiche. Infine, per poter gestire tutto in modo programmatico (sia la collezione dei progetti che l'interazione con SonarQube), si è creato uno script in Python che ha seguito questa pipeline:

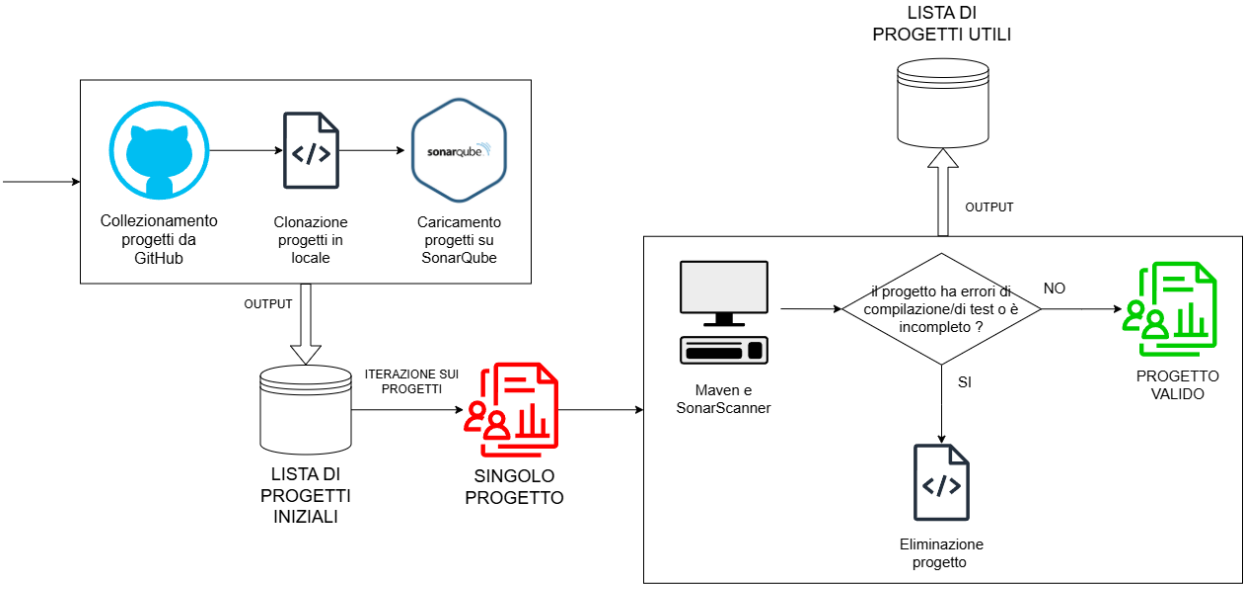



**Figura 4.2:** Pipeline Dataset

Per prima cosa c'è stato bisogno di costruire l'URL per la ricerca dei progetti su GitHub, in modo da poterli collezionare in modo automatico. Per fare ciò, si sono utilizzati i seguenti URL: *https://api.github.com/search/repositories?q=apache+commons+language:java&per_page=15*, per i progetti Apache, che da modo di filtrare i progetti in base al linguaggio di programmazione e al tipo di progetto (in questo caso Apache Commons, ovvero una collezione di librerie Java progettata per fornire funzionalità di base), e *https://api.github.com/orgs/LPODISIM2024/repos?type=private*, per i progetti degli studenti, grazie alla possibilità di accedere all'organizzazione LPODISIM2024, filtrando i progetti solo per quelli privati ( che risultano essere quelli degli studenti, andando a scartare da subito vari pdf o codice utile ai fini dell'insegnamento di Laboratorio Programmazione a Oggetti, ma non utile per il lavoro di tesi).
Avere progetti Java gestiti con Maven, strumento di automazione e gestione sviluppato e mantenuto da Apache Software Foundation, consente di avere la certezza e la sicurezza sulla struttura e sulle dipendenze dei progetti. Inoltre, per poter accedere alle chiamate API di GitHub, è stato necessario inserire l'API Key di Github come Header della chiamata. La risposta di GitHub, in formato JSON, è stata inizialmente letta e interpretata, così da poterne estrarre dati utili ai fini della clonazione. In particolare, per i progetti Apache, il JSON ha restituito un array di repository, ognuno dei quali rappresenta un progetto. Mentre, per i progetti degli studenti, il JSON ha restituito una lista di repository. Per entrambi, ogni progetto ha al suo interno un campo *clone_url*, che rappresenta l'URL per clonare il progetto. A questo punto, è stato possibile iterare sull'array e sulla lista di progetti e clonarli in locale, utilizzando *Repo.clone_from (clone_url, path_destinazione)*, dove Repo è una classe importata dalla libreria *gitpython* e i parametri al suo interno

sono rispettivamente l'URL di ogni progetto restituito dal JSON e il path di destinazione è la directory locale dove mantenere i progetti clonati.

Una volta clonati i progetti in locale, è stato necessario caricarli sul server di SonarQube, in modo da poterli analizzare. Sapendo che il server gira in locale sulla porta 9.000, si è costruito l'URL utilizzando la chiamata a SonarQube, come già visto nella Sezione 2.3.5 delle chiamate API. seguendo la documentazione ufficiale di SonarQube. A questo punto, una volta creata la request con: url, header e parametri, è bastato iterare sui progetti salvati in precedenza localmente e fare una chiamata POST per ogni progetto, in modo da caricarli sul server di SonarQube. Questo però non è bastato, perchè lo scanner dei progetti non avviene in modo automatico insieme alla creazione dello stesso. Allora è stato necessario costruire un metodo che si occupasse di scannerizzare i progetti, utilizzando Maven e SonarQube via terminale, sia alla loro creazione su SonarQube, sia alle successive analisi utilizzando l'approccio. Per fare ciò, è stato necessario prima di tutto creare un metodo per trovare il path del pom.xml (Project Object Model), cuore di un progetto Maven in cui sono definite tutte le configurazioni, informazioni, versioni e dipendenze del progetto, iterando ogni progetto nelle sue sottocartelle, così da evitare problemi di diversa struttura e composizione per progetti di diversa struttura e natura, e poi sono stati eseguiti i vari comandi da terminale, proprio a partire dal file pom.xml. La mancanza del pom ad un progetto implicava la non esecuzione dei comandi da terminale. Quindi, SonarQube non avrebbe effettuato un'analisi statica e, di conseguenza, diventava un progetto non utile ai fini del lavoro. Per questo veniva eliminato subito dopo. Detto ciò, sono utilizzati i seguenti comandi, per ogni progetto precedentemente clonato:

```
mvn clean verify install
```

in cui "mvn clean verify" pulisce, compila e testa il progetto. Tale comando, all'interno di un try-exception, viene gestito e catturato nel caso in cui il progetto contenesse errori di compilazione o di test. In tal caso, il progetto non verrà scannerizzato successivamente da sonar-scanner. Di conseguenza, rimarrà su SonarQube senza alcuna analisi statica del codice e verrà eliminato in seguito, sia in locale sia us SonarQube.

```
mvn jacoco:report
```

grazie alla quale verrà generato il report jacoco (Java Code Coverage, lo strumento di test coverage più usato e supportato da Maven), ovvero un file report .xml dove sono tenuti dati e informazioni su quali classi, metodi, linee e istruzioni sono stati coperti durante l'esecuzione dei test. Sarà utile per poter tenere traccia delle informazioni riguardanti il Coverage. Ovviamente, non tutti i progetti dispongono di jacoco integrato, per cui questo comando non viene gestito e catturato, poichè, i progetti studenteschi non dispongono di questo strumento.

```
mvn org.sonarsource.scanner.maven:sonar-maven-plugin:5.1.0.4751:sonar
-Dsonar.projectKey={param.get("project")}
-Dsonar.projectName={param.get("name")}
-Dsonar.host.url=http://localhost:9000
-Dsonar.token={os.getenv("SONAR_LOCAL_API_TOKEN")}
```

dove viene effettuato il vero e proprio sonar-scanner da parte di SonarQube. Una volta passati come parametri il nome e la chiave del progetto (già caricato su SonarQube), il softwre sarà capace di rintracciarlo, di effettuarne l'analisi statica, sia prima, che durante, che dopo il refactoring, e di caricarne i risultati in piat-

taforma, nella sua interfaccia web. Nel caso in cui il comando precedente passasse e che, quindi, il progetto contiene jacoco.xml, allora al comando di sonar-scanner viene aggiunta un'ulteriore informazione:

```
-Dsonar.coverage.jacoco.xmlReportPaths=target/site/jacoco/jacoco.xml
```

che serve a SonarQube per ritrovare le informazioni riguardante il Coverage .
Tali comandi sono gli stessi che verranno eseguiti anche nell'approccio dall'agente Sonar-Agent 3.1, con l'unica differenza che all'agente verrà aggiunta un'informazione:

```
-Dmaven.test.failure.ignore=true
```

poichè non è interessante sapere al momento dell'esecuzione dell'approccio se il codice nuovo appena refattorizzato passi o meno i test. Con questa riga si ha la certezza che il codice refattorizzato, anche se errato perchè ha modificato delle funzioni del programma, non blocchi l'intera esecuzione dell'approccio.
Quindi, una volta eseguiti i seguenti comandi da terminale nella fase di validazione per ogni progetto, è stato creato un metodo per estrarre i risultati dell'analisi statica, iterando per ogni progetto salvato in locale e utilizzando le API messe a disposizione da SonarQube. In particolare utilizzando una chiamata API già descritta (vedere 2.2), specificando le metriche da estrarre. Le metriche sono state scelte in base a quelle disponibili su SonarQube, a quelle più importanti e a quelle che potevano essere utili ai fini del lavoro (vedi Tabella 4.12).
A questo punto, però, è necessario puntualizzare un particolare, ovvero che tutti i progetti clonati e salvati in locale sono stati caricati correttamente su SonarQube, ma all'esecuzione del comando da terminale NON tutti hanno avuto esito positivo (chi per errori di compilazione, chi per altri motivi) . Quindi, a fronte di ciò, sono stati eliminati tutti quei progetti che, seppur presenti su SonarQube, non hanno avuto esito positivo all'analisi statica e, quindi, erano inutili ai fini del lavoro, andando a controllare il contenuto del JSON. In particolare, se il file JSON di risposta ad ogni progetto (all'ultima chiamata API) era vuoto, perchè di fatto l'esecuzione del sonar-scanner da terminale non era andato a buon fine, o le righe di codice di un progetto erano minori di 800, e, quindi, il progetto era troppo piccolo o ancora in fase di sviluppo, tale progetto veniva eliminato sia da SonarQube sia in locale, con corrispondenza 1:1, così che alla fine dell'iterazione, si avesse un dataset di progetti valido.
Per cui, arrivati a questo punto, sia in locale sia su SonarQube, sono rimasti tutti quei progetti che sono stati compilati correttamente, che sono stati creati correttamente su SonarQube e che hanno più di 800 righe di codice. Perciò, è stato creato un file .csv, contenente i risultati dell'analisi statica di ogni progetto, in modo da avere un report finale delle metriche (le stesse passate come parametri), su cui approfondire il lavoro svolto. In conclusione, tale pipeline ha portato alla creazione di un dataset, contente progetti Java open-source, gestiti con Maven, caricati su SonarQube e correttamente scannerizzati, con le rispettive metriche estratte. Il numero di progetti utilizzati sono osservabili nelle Tabelle 4.10 e 4.11, mentre i valori restituiti dall'analisi statica di SonarQube per ogni progetto prima del refactoring sono nelle Tabelle 4.14, per i progetti Apache, e 4.13, per i progetti degli studenti (il software considera l'intero progetto, quindi l'analisi viene fatta anche per le classi di test).

| Progetti | Valore |
|---|---|
| Clonati | 15 |
| Con errori generici | 4 |
| Con meno di 800 linee di codice | 0 |
| Eliminati (errori o incompletezza) | 4 |
| Salvati correttamente su SonarQube | 11 |
| Utilizzati nell'approccio | 11 |

**Tabella 4.10:** Statistiche sui progetti Apache

| Progetti | Valore |
|---|---|
| Clonati | 30 |
| Con errori generici | 8 |
| Con meno di 800 linee di codice | 11 |
| Eliminati (errori o incompletezza) | 19 |
| Salvati correttamente su SonarQube | 11 |
| Utilizzati nell'approccio | 11 |

**Tabella 4.11:** Statistiche sui progetti studenteschi

Mentre, per quanto riguarda gli attributi di qualità, si sono voluti osservare i seguenti, sia prima che dopo il refactoring applicato dall'approccio:

| Attributi | Descrizione |
|---|---|
| Lines of code | Numero totale di linee di codice nel progetto. |
| Coverage test | Percentuale di codice coperto dai test. |
| Test success density | Densità di successo dei test. |
| Test failures | Numero di fallimenti dei test. |
| Security vulnerabilities | Numero di vulnerabilità di sicurezza. |
| Security Rating | Rating della sicurezza, dalla A (migliore) alla E (peggiore). |
| Reliability bugs | Numero di bug di affidabilità. |
| Reliability Rating | Rating dell'affidabilità, dalla A alla E. |
| Maintainability code smells | Numero di code smells di manutenibilità. |
| Maintainability Rating | Rating della manutenibilità, dalla A alla E. |
| Cognitive Complexity | Complessità cognitiva del codice. |
| Duplicated Lines Density | Densità di linee di codice duplicate. |
| Blocker Violations | Numero di violazioni bloccanti (le più gravi). |
| Critical Violations | Numero di violazioni critiche (le più gravi dopo quelli bloccanti). |

**Tabella 4.12:** Descrizione degli attributi analizzati

| Project | LOC | Vuln. | Security Rating | Bugs | Reliab. Rating | Smells | Maintain. Rating | %Cognit. Compl. | %Duplic. Lines | Blocker Viol. | Critical Viol. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Studente 1 | 1555 | 1 | C | 50 | C | 335 | A | 241 | 2.4 | 46 | 50 |
| Studente 2 | 1093 | 6 | D | 32 | E | 222 | B | 175 | 6.4 | 40 | 66 |
| Studente 3 | 915 | 6 | D | 11 | D | 193 | B | 174 | 6.1 | 29 | 59 |
| Studente 4 | 1416 | 10 | D | 16 | E | 297 | B | 431 | 3.1 | 52 | 85 |
| Studente 5 | 1255 | 11 | D | 34 | D | 292 | B | 215 | 2.9 | 45 | 74 |
| Studente 6 | 1624 | 8 | D | 21 | E | 311 | B | 324 | 2.6 | 45 | 93 |
| Studente 7 | 4463 | 13 | D | 86 | E | 615 | A | 572 | 15.7 | 72 | 103 |
| Studente 8 | 1635 | 4 | C | 18 | D | 257 | A | 76 | 0.0 | 42 | 20 |
| Studente 9 | 2934 | 3 | D | 36 | E | 505 | A | 574 | 1.5 | 59 | 148 |
| Studente 10 | 1634 | 12 | D | 15 | E | 327 | B | 516 | 0.0 | 25 | 52 |
| Studente 11 | 2779 | 11 | D | 39 | E | 233 | A | 511 | 1.1 | 32 | 106 |

**Tabella 4.13:** Tabella qualità progetti studenteschi

| Projects | LOC | %Test Cov. | Vuln. | Security Rating | Bugs | Reliab. Rating | Smells | Maintain. Rating | %Cognit. Compl. | %Duplic. Lines | Blocker Viol. | Critical Viol. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| beanutils | 9588 | 75.1 | 8 | D | 293 | D | 1301 | A | 2041 | 1.3 | 230 | 234 |
| codec | 10050 | 94.7 | 99 | C | 800 | C | 710 | A | 1569 | 1.5 | 105 | 131 |
| compress | 44846 | 83.7 | 77 | D | 959 | E | 3828 | A | 8331 | 1.9 | 585 | 899 |
| configuration | 21065 | 91.0 | 69 | E | 159 | C | 1166 | A | 2566 | 0.1 | 300 | 165 |
| dbcp | 13738 | 68.5 | 73 | D | 183 | E | 1310 | A | 1534 | 0.2 | 111 | 134 |
| dbutils | 3612 | 70.8 | 7 | D | 152 | E | 451 | A | 267 | 0.0 | 251 | 25 |
| lang | 32788 | 95.1 | 24 | D | 1048 | D | 3208 | A | 5932 | 1.3 | 434 | 406 |
| math | 69782 | 74.6 | 89 | D | 1953 | D | 3757 | A | 12579 | 2.7 | 883 | 1027 |
| net | 17992 | 37.1 | 53 | D | 250 | E | 1757 | A | 2971 | 2.5 | 283 | 246 |
| pool | 6401 | 86.9 | 8 | C | 91 | D | 555 | A | 864 | 1.1 | 83 | 75 |
| text | 11224 | 97.3 | 10 | D | 127 | D | 695 | A | 2085 | 6.3 | 125 | 154 |

**Tabella 4.14:** Tabella qualità progetti Apache

In entrambe le tabelle 4.13 e 4.14, sono state escluse diverse colonne. Una di queste è quella di Test Success Density, poichè ogni progetto, per poter essere nel dataset, deve necessariamente passare tutti i test (se presenti) prima di farne refactoring, per cui prima del refactoring questa colonna è stata sempre 100% per tutti i progetti. Un'altra colonna, invece, è quella di Test Failures che, in modo analogo e contrario, presentava tutti 0 per ogni progetto. L'unica colonna per cui le due tabelle differiscono è quella di Test Coverage. Questo perchè, come già accennato, i progetti degli studenti non avevano a disposizione JaCoCo come strumento per poter immagazzinare i dati e le informazioni riguardanti la copertura dei test. Al contrario, i progetti Apache avevano questo strumento incorporato, per cui ne è stato ricavato positivamente il code coverage.

# Capitolo 5

# Risultati

In questo capitolo ci si concentrerà solo ed esclusivamente sui risultati ottenuti dall'approccio. In particolare, verranno enunciate, spiegate e approfondite le risposte alle tre RQs. Per fare ciò, è stato necessario creare dei file .csv e .excel per poter tenere traccia di tutti i valori delle metriche e un file .excel finale contenente i miglioramenti e/o peggioramenti in percentuale dei valori delle metriche tra prima e dopo l'applicazione dell'approccio. Gli attributi di qualità su cui ci si è concentrati sono gli stessi già presenti nell'analisi pre-refactoring (vedi Tabella 4.12). Ciò significa che, complessivamente, sono state calcolate le differenze in grandezza, sicurezza, affidabilità, manutenibilità, duplicati, test e problemi più critici in generale.
Gli attributi che non sono presenti nelle tabelle sono quegli attributi che, nonostante il refactoring, non sono cambiati e, quindi, non hanno subito nè miglioramenti nè peggioramenti.
Inoltre, per ogni risultato verranno presentati ulteriori dati, quali: tentativi totali di refactoring falliti dall'approccio (max 3 per classe) a causa di errori di compilazione, per un determinato dataset di progetti; tempo totale di esecuzione per ogni progetto; tempo totale di esecuzione dell'approccio per il dataset di progetti ( tempo di esecuzione totale + tempo di overhead).

## 5.1 RQ1: Come impatta il MAS per fare code refactoring su progetti strutturati Apache e su progetti studenteschi ?

Di seguito verranno mostrati i risultati, sottoforma di tabella, di quanto l'approccio sia stato efficace nel fare code refactoring per i progetti Apache e per i progetti studenteschi, con relativi tempi di esecuzione e tentativi totali per la Research Question 1. In particolare, è necessario ribadire e specificare che il numero di classi, prese per ogni progetto del dataset, sono scelte in modo casuale, escludendo eventuali file con estensione .xml e non considerando le classi utilizzate per il test.
Di seguito, i risultati nelle Tabelle 5.1 e 5.2:

| Project | LOC | Cover. | Test Success | Test Failures | Bugs | Code Smells | Cognit. Complex. | Duplic. Lines | Critical Violat. | Exec. Time (s) |
|---|---|---|---|---|---|---|---|---|---|---|
| beanutils | +0.41% ↑ | -0.40% ↓ | -0.10% ↓ | – | +0.34% ↑ | – | +0.73% ↑ | – | – | 2450 |
| codec | +0.06% ↑ | – | – | – | – | -0.28% ↓ | +0.06% ↑ | – | – | 5138 |
| compress | +0.04% ↑ | – | – | – | – | -0.10% ↓ | +0.05% ↑ | – | – | 2509 |
| configuration | – | – | – | – | – | – | – | – | – | 1610 |
| dbcp | +0.04% ↑ | – | – | – | – | +0.08% ↑ | – | – | +0.75% ↑ | 1468 |
| dbutils | +0.83% ↑ | -0.14% ↓ | – | – | – | +0.44% ↑ | +1.87% ↑ | – | -4.00% ↓ | 789 |
| lang | +0.08% ↑ | -0.11% ↓ | -0.10% ↓ | +48 ↑ | – | +0.09% ↑ | +0.29% ↑ | – | – | 3075 |
| math | – | – | – | – | – | – | – | – | – | 4989 |
| net | +0.24% ↑ | +0.27% ↑ | – | – | – | -0.28% ↓ | -0.27% ↓ | -8.00% ↓ | -0.81% ↓ | 1176 |
| pool | +0.34% ↑ | -0.23% ↓ | – | – | – | – | +0.81% ↑ | – | – | 1447 |
| text | +0.05% ↑ | – | -0.50% ↓ | +7 ↑ | – | +0.14% ↑ | -0.29% ↓ | – | – | 1031 |
| **Tempo totale approccio:** | | | | | | | | | | **28927 (8h)** |
| **Tentativi totali falliti: 76** | Ciò significa che l'approccio ha fallito il refactoring 76 volte, a causa di errori di compilazione | | | | | | | | | |

**Tabella 5.1:** Tabella RQ1 progetti Apache

| Project | Lines of Code | Bugs | Code Smells | Cognitive Complexity | Duplicated Lines Density | Critical Violations | Execution Time (s) |
|---|---|---|---|---|---|---|---|
| Studente 1 | +0.96% ↑ | – | -2.99% ↓ | -2.49% ↓ | -4.17% ↓ | -20.00% ↓ | 393 |
| Studente 2 | +3.48% ↑ | – | -1.80% ↓ | +1.14% ↑ | -7.81% ↓ | -4.55% ↓ | 422 |
| Studente 3 | +3.28% ↑ | – | -6.22% ↓ | +4.60% ↑ | -65.57% ↓ | -5.08% ↓ | 1045 |
| Studente 4 | +1.55% ↑ | – | – | +1.62% ↑ | – | – | 360 |
| Studente 5 | – | – | – | – | – | – | 496 |
| Studente 6 | +3.51% ↑ | – | -2.25% ↓ | -2.47% ↓ | -69.23% ↓ | -4.30% ↓ | 451 |
| Studente 7 | +0.07% ↑ | – | -0.49% ↓ | -0.35% ↓ | -1.27% ↓ | -1.94% ↓ | 662 |
| Studente 8 | +0.37% ↑ | -5.56% ↓ | -0.78% ↓ | – | – | – | 442 |
| Studente 9 | +0.44% ↑ | – | -0.20% ↓ | +0.87% ↑ | -6.67% ↓ | – | 844 |
| Studente 10 | +2.14% ↑ | -6.67% ↓ | -4.89% ↓ | -0.19% ↓ | – | -9.62% ↓ | 669 |
| Studente 11 | +1.04% ↑ | +2.56% ↑ | +0.43% ↑ | +0.98% ↑ | – | – | 368 |
| **Tempo totale approccio:** | | | | | | | **6171 (1.7h)** |
| **Tentativi totali falliti: 18** | Ciò significa che l'approccio ha fallito il refactoring 18 volte, a causa di errori di compilazione | | | | | | |

**Tabella 5.2:** Tabella RQ1 progetti studenteschi

**Discussione risultati RQ1** In generale, i risultati lasciano diversi punti di conversazione e di studio. Innanzitutto, è necessario parlare dei miglioramenti e peggioramenti, soprattutto in relazione al tipo di progetto su cui l'approccio ha tentato di fare refactoring. In particolare, è possibile osservare che per i

progetti Apache i risultati non sono ottimi. Solo in rare eccezioni l'approccio è riuscito a migliorare alcuni valori ma, complessivamente, non è riuscito a migliorarne tanti e, anzi, ha peggiorato alcuni valori, come nel caso del Coverage, Test Success Density e Test Failures. Ciò è dovuto, probabilmente, al fatto che il refactoring effettuato non viene catturato da alcuni casi di test (e quindi la percentuale totale di Coverage diminuisce), venendo saltato (diminuendo il Test Success Density) o addirittura introducendo nuovi errori di test, a causa di modifiche alle funzionalità del codice, concetto che va contro le regole base del code refactoring (aumentando i Test Failures, che prima del refactoring erano a 0). Diversi invece sono i risultati per quanto riguarda i prgetti degli studenti. Come ci si aspettava, essendo progetti nativi e meno strutturati come quelli sviluppati da Apache Foundation, contengono errori più semplici da trovare e da correggere. Difatti, complessivamente, l'approccio ha avuto discreti risultati, soprattutto per quanto riguarda code smells e Duplicates Lines. Inoltre, è possibile osservare che i problemi corretti dall'approccio sono problemi importanti (basti osservare le percentuali diminuite nelle Critical Violations). Mentre, per quanto riguarda, Coverage e Tests per i progetti studenteschi, non avendo JaCoCo integrato, non è possibile verificarne l'effettiva modifica (anche se sono solo in 3 ad avere effettivamente dei test case). Alcune cose accomunano i due progetti: in generale, le linee di codice tendono ad aumentare e i Rating degli attributi Security, Reliability, Maintainability e il numero di Blocker Violations rimangono invariati, non subendo alcun miglioramento o peggioramento. Questo a testimonianza del fatto che l'approccio, nonostante modifichi e migliori o peggiori delle metriche che contribuiscono ai rating, non modifica in alcun modo i rating degli attributi principali. Questo implica sia che non è riuscito a trovare errori gravi (ma per questo bastava vedere che la colonna Blocker Violations non era presente, in quanto non ha subito alcuna modifica) e sia che non ne ha aggiunti. È anche possibile che l'approccio sia stato sfortunato, avendo preso classi randomiche che non contenevano problemi e proprio per questo si è deciso di formulare la RQ2 e, successivamente, la RQ3. Mentre, per quanto riguarda i tentativi e i tempi, si osservi come ci sia un gran distacco tra i progetti Apache e quelli degli studenti. Questo, molto probabilmente, dovuto al fatto che i primi sono progetti molto più strutturati, complessi e lunghi. Cosa per cui l'approccio tende a metterci più tempo per: creare il prompt, passare il codice della classe, effettuare refactoring ed eseguire comandi da terminale. Infatti, se si pone particolare attenzione anche sui tentativi totali, si può notare che, a parità di progetti e di classi refattorizzate, ci sia una differenza di tentativi provati dall'approccio. Questo ancora a testimonianza di quanto siano differenti le classi tra i progetti Apache e i progetti studenteschi.

**Answer to RQ1:** i progetti strutturati e commerciali Apache hanno subito pochi cambiamenti con il refactoring automatizzato, con prevalenza di modifiche peggiorative. Mentre, per i progetti sviluppati dagli studenti, l'approccio sembra aver portato dei miglioramenti attraverso il refactoring con una prevalenz di modifiche migliorative, in particolare per bugs, code smells e duplicates lines. Questo, come ci si aspettava, dovuto al fatto che i primi progetti sono molto più complessi, studiati e aggiornati, mentre i secondi tendono ad essere generalmente più semplici.

## 5.2 RQ2: Come variano i risultati aumentando il numero di classi da refattorizzare ?

Tenendo conto dei risultati della precedente RQ, si è voluti aumentare il numero di classi, per osservare le eventuali differenza e se l'approccio, date un numero maggiore di classi, migliorava o peggiorava la risposta precedente, per rispondere alla Research Question 2.

Per questo, si è raddoppiato il numero di classi per progetto, passando da 5 classi a 10 classi. Le classi, come prima, sono state selezionate in modo casuale e randomico.
Di seguito, i risultati, visibili nella Tabella 5.3:

| Project | LOC | Cover. | Test Succ. | Test Fail. | Vulner. | Bugs | Code Smells | Cognit. Comp. | Duplic. Lines | Crit. Viol. | Block. Viol. | Exec. Time (s) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| beanutils | +0.34% ↑ | -2.67% ↓ | -9.50% ↓ | – | – | +1.37% ↑ | +0.15% ↑ | +0.73% ↑ | – | +1.71% ↑ | – | 5975 |
| codec | +0.07% ↑ | -0.11% ↓ | – | – | – | – | -0.28% ↓ | +0.32% ↑ | – | – | – | 4717 |
| compress | +0.13% ↑ | – | – | – | – | +0.63% ↑ | +0.03% ↑ | +0.14% ↑ | – | -0.11% ↓ | – | 3944 |
| configuration | – | – | – | – | – | – | – | – | – | – | – | 3898 |
| dbcp | +0.08% ↑ | -0.15% ↓ | -0.70% ↓ | +3 ↑ | – | – | +0.23% ↑ | +0.46% ↑ | – | – | – | 10185 |
| dbutils | +0.25% ↑ | -0.28% ↓ | – | – | – | – | – | +1.12% ↑ | – | – | – | 4721 |
| lang | +0.25% ↑ | -0.11% ↓ | – | – | – | – | +0.62% ↑ | +0.47% ↑ | – | – | – | 6062 |
| math | – | – | – | – | – | – | – | – | – | – | – | 9108 |
| net | +0.18% ↑ | – | – | – | – | -0.40% ↓ | +0.23% ↑ | +0.47% ↑ | – | – | – | 5975 |
| pool | -0.19% ↓ | +0.23% ↑ | – | – | – | – | – | – | -9.09% ↓ | – | – | 3873 |
| text | – | – | – | – | – | – | – | – | – | – | – | 14283 |
| **Tempo totale approccio:** | | | | | | | | | | | | **72788 (20.2h)** |
| **Tentativi totali falliti: 173** | Ciò significa che l'approccio ha fallito il refactoring 173 volte, a causa di errori di compilazione | | | | | | | | | | | |
| Studente 1 | -0.39% ↓ | – | – | – | – | -36.00% ↓ | -4.78% ↓ | -8.30% ↓ | -54.17% ↓ | -6.00% ↓ | +4.35% ↑ | 1247 |
| Studente 2 | +3.20% ↑ | – | – | – | – | – | -8.56% ↓ | -5.71% ↓ | -100.00% ↓ | -19.70% ↓ | – | 1002 |
| Studente 3 | +1.64% ↑ | – | – | – | – | – | +1.55% ↑ | +2.87% ↑ | -63.93% ↓ | +5.08% ↑ | – | 578 |
| Studente 4 | +0.92% ↑ | – | – | – | – | – | – | +0.70% ↑ | – | – | – | 618 |
| Studente 5 | +1.20% ↑ | – | – | – | -18.18% ↓ | -8.82% ↓ | -3.42% ↓ | +0.47% ↑ | -3.45% ↓ | -10.81% ↓ | +2.22% ↑ | 834 |
| Studente 6 | +0.92% ↑ | – | – | – | – | – | – | +0.31% ↑ | – | -1.08% ↓ | – | 842 |
| Studente 7 | -0.11% ↓ | – | – | – | – | – | -0.33% ↓ | -0.70% ↓ | -1.27% ↓ | -0.97% ↓ | – | 1227 |
| Studente 8 | +0.67% ↑ | – | – | – | – | – | -4.28% ↓ | +9.21% ↑ | – | -10.00% ↓ | – | 919 |
| Studente 9 | +0.55% ↑ | – | – | – | – | -11.11% ↓ | -2.18% ↓ | -4.70% ↓ | – | -4.73% ↓ | – | 1793 |
| Studente 10 | +0.80% ↑ | – | – | – | -25.00% ↓ | -6.67% ↓ | -2.45% ↓ | – | – | -3.85% ↓ | – | 1538 |
| Studente 11 | +0.32% ↑ | – | – | – | – | – | – | +0.59% ↑ | – | – | – | 6231 |
| **Tempo totale approccio:** | | | | | | | | | | | | **16867 (4.7h)** |
| **Tentativi totali falliti: 66** | Ciò significa che l'approccio ha fallito il refactoring 66 volte, a causa di errori di compilazione | | | | | | | | | | | |

**Tabella 5.3:** Tabella unificata RQ2 (Apache + Studenti)

**Discussione risultati RQ2** I risultati di questa RQ2 seguono quelli già osservati e descritti della RQ precedente. Pur aumentando il numero di classi, questo non implica maggiori miglioramenti per le classi dei progetti Apache i quali variano di poco, con una maggioranza di peggioramenti. I progetti degli studenti, tuttavia, hanno un riscontro positivo in questa RQ. Infatti i risultati sono migliori rispetto a prima, sia in quantità che in qualità per le metriche già viste prima come bugs, code smells, duplicates lines e critical violations. Inoltre, a differenza di prima, l'approccio è riuscito a trovare e migliorare anche alcune

vulnerabilities. Nonostante tutto, però, l'unico risultato negativo, anche per i progetti degli studenti, è quello dei Blocker Violations che, in due casi, sono leggermente aumentati. Ciò significa che l'approccio, probabilmente, migliorando alcune metriche (non necessariamente gravi), ha aggiunto alcuni problemi di tipo Blocker, quindi, molto gravi. Considerando la tabella di assegnazione dei Rating, già vista nel Capitolo di background 2.2, si è certi che i Blocker Violations aggiunti non sono Issues di vulnerabilities o di bugs, altrimenti sarebbero cambiati anche i Rating di Security o Reliability, ma sono Issues di code smells, che però non peggiorano il technical debt ratio a tal punto da modificare il Rating della Maintainability. Per cui, come per la RQ1, i Rating di Security, Reliability e Maintainability sono rimasti gli stessi. Mentre, per quanto riguarda i tentativi dell'approccio e i tempi di esecuzione, sono, ovviamente, aumentati rispetto a prima, dovendo refattorizzato più classi. Tuttavia, raddoppiare il numero di classi non ha implicato un raddoppio anche del tempo e dei tentativi, ma bensi un triplicamento (o quasi) sia dei tentativi sia del tempo totale di esecuzione. Si osservi, inoltre, come in alcuni progetti l'approccio non riusciva immediatamente a fare un refactoring valido, andando a spendere molto tempo di esecuzione per quel singolo progetto. È sempre bene considerare, però, che le classi sono pur sempre prese in modo randomico. Di conseguenza non si è dati sapere quali classi sono stati prese nella precedente RQ1 e in questa RQ2. Potrebbe essere che, in questo caso, l'approccio abbia selezionato classi più complesse, mettendoci più tempo, oppure al contrario, classi più semplici.

**Answer to RQ2:** i risultati seguono quelli che sono stati i risultati della RQ1. In particolare, per i progetti Apache continuano ad esserci poche variazioni, con maggioranza peggiorativa. Mentre per i progetti degli studenti sono aumentati i miglioramenti. Questo implica che l'aumentare il numero di classi, per progetti nativi e non troppo difficili porta, complessivamente, a dei miglioramenti del codice.

## 5.3 RQ3: Quanto è buono e affidabile l'approccio per fare code refactoring quando si specifica una metrica di qualità ?

Adesso, dopo aver modificato Query Writer, task1 e Code Replacer nel file yaml, come sià visto nel sottocapitolo delle RQs nel Capitolo 4, si è voluti rispondere alla Research Question 3. Da specificare che, avendo osservato i risultati delle precedenti Research Question, si è deciso di utilizzare per questa RQ3 solamente il dataset di progetti LPO, in quanto quelli di Apache non hanno avuto miglioramenti significativi. Ciò non toglie che il lavoro può essere continuato e sviluppato in futuro anche sui progetti Apache. In particolare, per ogni tipo di progetto studentesco, sono state effettuate 3 esecuzioni distinte e indipendenti, in ognuno dei quali si ha specificato al sistema multi agente una metrica su cui concentrarsi, rispettivamente: vulnerabilities (vedere Tabella 5.4), bugs (vedere Tabella 5.5) e code smells (vedere Tabella 5.6). In questo modo, dovrebbero vedersi dei miglioramenti maggiori per queste 3 metriche.

### 5.3.1 Vulnerabilities

| Project | LOC | Test Succ. | Vulner. | Bugs | Reliab. Rating | Code Smells | Cognit. Comp. | Duplic. Lines | Block. Viol. | Crit. Viol. | Exec. Time (s) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Studente 1 | -0.26% ↓ | – | – | – | – | -0.30% ↓ | -0.41% ↓ | -4.17% ↓ | – | – | 182 |
| Studente 2 | +0.73% ↑ | – | -50.00% ↓ | -9.38% ↓ | – | – | +4.00% ↑ | – | – | -3.03% ↓ | 1440 |
| Studente 3 | +2.84% ↑ | – | -16.67% ↓ | – | – | +3.63% ↑ | +12.07% ↑ | -1.64% ↓ | – | +3.39% ↑ | 1325 |
| Studente 4 | -2.54% ↓ | – | -20.00% ↓ | -6.25% ↓ | – | -6.40% ↓ | -24.59% ↓ | -67.74% ↓ | – | -11.76% ↓ | 6538 |
| Studente 5 | +2.07% ↑ | – | -27.27% ↓ | -5.88% ↓ | – | -6.51% ↓ | -4.65% ↓ | -3.45% ↓ | – | -17.57% ↓ | 1470 |
| Studente 6 | – | – | – | – | – | – | – | – | – | – | 2067 |
| Studente 7 | – | – | – | – | – | – | – | – | – | – | 9088 |
| Studente 8 | +0.98% ↑ | -16.70% ↓ | -50.00% ↓ | -5.56% ↓ | -25.00% ↓ | -1.95% ↓ | +3.95% ↑ | – | – | -10.00% ↓ | 717 |
| Studente 9 | – | – | – | – | – | – | – | – | – | – | 732 |
| Studente 10 | +1.59% ↑ | – | -33.33% ↓ | -20.00% ↓ | -40.00% ↓ | -2.45% ↓ | +1.74% ↑ | – | -8.00% ↓ | -1.92% ↓ | 1613 |
| Studente 11 | -0.22% ↓ | – | – | – | – | -0.43% ↓ | -0.59% ↓ | – | – | – | 4122 |
| **Tempo totale approccio:** | | | | | | | | | | | **29334 (8h)** |
| **Tentativi totali falliti: 97** | Ciò significa che l'approccio ha fallito il refactoring 97 volte, a causa di errori di compilazione, oppure perchè non è riuscito a migliorare il valore di *vulnerabilities* | | | | | | | | | | |

**Tabella 5.4:** Tabella progetti studenti RQ3 per vulnerabilities

### 5.3.2 Bugs

| Project | LOC | Bugs | Reliab. Rating | Code Smells | Cognit. Comp. | Duplic. Lines | Block. Viol. | Critic. Viol. | Exec. Time (s) |
|---|---|---|---|---|---|---|---|---|---|
| Studente 1 | -0.71% ↓ | -34.00% ↓ | – | -8.66% ↓ | -2.07% ↓ | -54.17% ↓ | +4.35% ↑ | -2.00% ↓ | 1331 |
| Studente 2 | +0.91% ↑ | -34.38% ↓ | -20.00% ↓ | -8.11% ↓ | -2.86% ↓ | -100.00% ↓ | -12.50% ↓ | -9.09% ↓ | 1513 |
| Studente 3 | +2.62% ↑ | – | – | +4.15% ↑ | +9.20% ↑ | -73.77% ↓ | – | +5.08% ↑ | 1614 |
| Studente 4 | +0.56% ↑ | – | – | -1.01% ↓ | -6.26% ↓ | +45.16% ↑ | – | -2.35% ↓ | 2207 |
| Studente 5 | +2.39% ↑ | -5.88% ↓ | – | -7.88% ↓ | +0.47% ↑ | – | – | -20.27% ↓ | 1108 |
| Studente 6 | +2.52% ↑ | -9.52% ↓ | – | -1.61% ↓ | +1.85% ↑ | -30.77% ↓ | – | -9.68% ↓ | 2183 |
| Studente 7 | – | – | – | – | – | – | – | – | 14740 |
| Studente 8 | +0.80% ↑ | -11.11% ↓ | – | -4.28% ↓ | +19.74% ↑ | – | – | – | 1092 |
| Studente 9 | +1.53% ↑ | +5.56% ↑ | – | -1.58% ↓ | -0.35% ↓ | -6.67% ↓ | – | -4.05% ↓ | 3433 |
| Studente 10 | +0.92% ↑ | -6.67% ↓ | -40.00% ↓ | – | -9.30% ↓ | – | -8.00% ↓ | -7.69% ↓ | 1484 |
| Studente 11 | – | – | – | – | – | – | – | – | 16851 |
| **Tempo totale approccio:** | | | | | | | | | **47597 (13h)** |
| **Tentativi totali falliti: 139** | Ciò significa che l'approccio ha fallito il refactoring 139 volte, a causa di errori di compilazione, oppure perchè non è riuscito a migliorare il valore di *bugs* | | | | | | | | |

**Tabella 5.5:** Tabella progetti studenti RQ3 per bugs

### 5.3.3 Code Smells

| Project | LOC | Vulner. | Bugs | Code Smells | Cognit. Comp. | Duplic. Lines | Block. Viol. | Critic. Viol. | Exec. Time (s) |
|---|---|---|---|---|---|---|---|---|---|
| Studente 1 | -0.39% ↓ | – | -24.00% ↓ | -4.48% ↓ | -12.86% ↓ | -54.17% ↓ | +2.17% ↑ | – | 1391 |
| Studente 2 | +2.38% ↑ | – | -6.25% ↓ | -5.41% ↓ | -3.43% ↓ | -71.88% ↓ | – | -4.55% ↓ | 1567 |
| Studente 3 | +13.99% ↑ | – | +9.09% ↑ | -9.84% ↓ | -27.59% ↓ | -67.21% ↓ | – | -22.03% ↓ | 1826 |
| Studente 4 | +3.74% ↑ | -20.00% ↓ | -31.25% ↓ | -13.13% ↓ | -32.25% ↓ | -100.00% ↓ | – | -24.71% ↓ | 3191 |
| Studente 5 | -2.07% ↓ | -18.18% ↓ | -29.41% ↓ | -29.79% ↓ | -33.49% ↓ | -100.00% ↓ | – | -43.24% ↓ | 1577 |
| Studente 6 | +0.37% ↑ | – | +9.52% ↑ | -5.14% ↓ | -10.19% ↓ | -65.38% ↓ | – | -7.53% ↓ | 2069 |
| Studente 7 | – | – | – | – | – | – | – | – | 9645 |
| Studente 8 | +2.20% ↑ | – | -5.56% ↓ | -10.12% ↓ | +5.26% ↑ | – | – | -20.00% ↓ | 1373 |
| Studente 9 | +0.03% ↑ | – | – | – | – | – | – | – | 4481 |
| Studente 10 | +0.86% ↑ | – | – | -0.92% ↓ | -1.16% ↓ | – | – | -3.85% ↓ | 2191 |
| Studente 11 | – | – | – | – | – | – | – | – | 17353 |
| **Tempo totale approccio:** | | | | | | | | | **46703 (13h)** |
| **Tentativi totali falliti: 142** | Ciò significa che l'approccio ha fallito il refactoring 142 volte, a causa di errori di compilazione, oppure perchè non è riuscito a migliorare il valore di *code smells* | | | | | | | | |

**Tabella 5.6:** Tabella progetti studenti RQ3 per code smells

**Discussione risultati RQ3** I risultati mostrano quello che ci si aspettava, cioè l'approccio ha portato miglioramenti significativi quando hanno una maggior conoscenza su determinate metriche. Nello specifico, per le vulnerabilities ci sono stati importanti miglioramenti, fino al 50%, in due casi, ma senza migliorare il Security Rating. Questo significa che non sono state eliminate le vulnerabilities più gravi, tanto da modificarne il Rating. Per i bugs, invece, i risultati sono stati allo stesso modo buoni, andando addirittura a migliorare anche il Reliability Rating. Ciò significa che, in due casi, l'approccio ha eliminato bugs rilevanti per il Rating. Per i code smells, infine, si hanno i miglior cambiamenti. Infatti, non solo sono migliorati i code smells stessi, ma anche tutto ciò che li riguarda come duplicates lines e cognitive complexity. Tuttavia, non sono migliorati i Maintainability Rating, soprattutto perchè, come visto nella Tabella 4.13, erano già molto buoni (A / B). Per quanto riguarda i tempi di esecuzione e i tentativi falliti, anche in questo caso, rispettano le aspettative. In questa RQ3, come già detto, l'approccio effettua un doppio controllo: errori di compilazione e miglioramento del valore di una metrica. Ragion per cui è normale che i tempi e i tentativi siano aumentati rispetto alla RQ2, nonostante il numero di classi inferiore. Anche perchè, è probabile, che alcune classi contenessero poche vulnerabilities, o bugs, o code smells, e questo portava l'approccio a ritentare più volte con lo scopo di migliorare, per esempio, un singolo issue. Si suppone, quindi, che i miglioramenti più significativi siano stati per le classi con il maggior numero di issues, su cui l'approccio riusciva a fare un code refactoring efficace, migliorando il valore di metrica. Tuttavia, è da osservare come per alcuni progetti siano stati effettuati cambiamenti minimi oppure neanche uno. Ciò vuol dire che comunque l'approccio non funziona sempre per ogni progetto, ma dipende dalla sua struttura, dai suoi errori e dalla sua complessità.

**Answer to RQ3:** modificando la conoscenza e la descrizione di alcuni agenti e task con focus su una particolare metrica di qualità, l'approccio riesce a portare ulteriori miglioramenti attraverso il refactoring mirato su quella metrica, almeno per i progetti più semplici e meno strutturati come i progetti degli studenti.

## 5.4 Threats to validity

Durante la realizzazione dell'approccio, sono stati affrontati diversi problemi, in particolare, nella corretta riuscita dell'esecuzione. Nello specifico, raramente, qualche agente del sistema multi agente potrebbe avere delle "allucinations" e potrebbe ripetere all'infinito delle righe di codice o dei caratteri. Questo, ha portato l'agente a ritornare il codice fino alla fine dei suoi tokens, così aumentando il tempo di esecuzione sia totale che del singolo progetto. Altre volte, invece, gli agenti che eseguivano i task con tool personalizzato (Code Replacer ma soprattutto Sonar Agent) non rispettavano appieno il metodo deterministico del tool. Questo, addirittura, non portava ad un aumento di tempo esecutivo, ma addirittura al blocco totale dell'approccio. Perchè, nel caso di Sonar Agent, questo è l'agente responsabile delle iterazioni tra i vari kickoff della crew e tra crew e flow. Non eseguire correttamente (o non eseguirlo per niente) il codice deterministico del tool, portava l'agente a non capire come interfacciarsi con SonarQube, invalidando tutta quella parte in cui si raccolgono i dati come errori di compilazione e metrica (per la RQ3). A questo punto, poi, non veniva nè eseguita la callback nè restituito l'output pydantic. Di conseguenza i campi "valid", "errors" e "metric" nel risultato del kickoff della crew erano invalidi, implicando errori semantici e, di conseguenza, il blocco dell'intera esecuzione. A questo problema, poi, è stato trovato una soluzione, aumentando il numero di tentativi e rieffettuando il refactoring, nonostante il refactoring precedente su stessa classe, di fatto, non sia mai stato analizzato. Altri tipi di problemi, invece, sono derivanti dal provider del modello LLM utilizzato o dai limiti del modello stesso. Questo perchè è capitato, raramente, che il server Mistral non funzionasse al 100%, compromettendo l'esecuizione dell'approccio. Ovviamente dipende sempre dal provider che si sta utilizzando. Mentre, per quanto riguarda il modello in sè, raramente è successo che il codice di una classe (questo solo per progetti Apache) non venisse diviso in più parti dal provider, facendo si che come token in input passasse tutta la classe per intero, eccedendo nei limiti di token al secondo/al minuto per Mistral-Medium, invalidando anche in questo caso l'intera esecuzione. Inoltre, è bene specificare che ogni esecuzione, con altissima probabilità, avrà risultati diversi. Questo a causa di 2 motivi principali: il 1° è che nelle RQ1 e RQ2 le classi sono prese in modo casuale, per cui ad ogni esecuzione può essere preso sempre un insieme di classi diverso. Il 2° motivo, nonchè quello principale, è che i modelli non ragionano sempre allo stesso modo. Per cui, anche se le classi non sono prese in modo randomico e che, quindi, vengono refattorizzate sempre le stesse, come nel caso della RQ3, rimane comunque lavoro e ragionamento del modello effettuare un refactoring efficace. Indipendentemente dal tipo di modello, quindi, il refactoring può essere sempre diverso. Per fare un esempio: con lo stesso identico modello è possibile che in una esecuzione si migliori una classe e se ne peggiori un'altra, mentre in un'altra esecuzione può essere possibile che avvenga il contrario. Questo perchè, a priori, il modello LLM ragiona sempre in modo diverso per ogni esecuzione e, ovviamente, non ha memoria. Sarebbe da testare l'approccio diverse volte per un dataset di progetti e prendere i risultati migliori. In questo lavoro di tesi, tuttavia, sono stati presi solo i primi risultati restituiti dall'approccio, senza affrontare questo problema e senza soffermarsi sulla probabilità di miglioramento in un'altra eventuale esecuzione.

Infine, un classico problema che si presenta quando si affronta il refactoring, è di garantire che il codice

modificato garantisca lo stesso funzionamento ante modifica. Ogni modifica, sebbene guidata da motivazioni di carattere qualitativo, potrebbe modificare il comportamento del codice in maniera imprevedibile. Per arginare questo problema si è fatto affidamento sui test presenti sui progetti esaminati. Essenzialmente, i test sono stati rieseguiti dopo ever effettuato le modifiche al codice per valutare se il comportamento fosse cambiato. Circa i progetti Apache, il refactoring ha causato il fallimento o l'omissione di alcuni test. Mentre nei progetti degli studenti, non è stato possibile eseguire i test, in quanto la scrittura dei test non era argomento del corso. Di conseguenza solo alcuni progetti disponevano di test utilizzabili, come riporta la Tabella 4.11. Questo aspetto tuttavia, andrebbe approfondito in maniera più accurata per valutare le capacità del sistema multi agente di garantire il funziomanento originale del codice.

# Capitolo 6

# Related Works

## 6.1 Code Refactoring prima dell'IA

Esistono altri articoli e altre ricerche che descrivono e spiegano come il code refactoring influisca sugli attributi di qualità di un progetto software, a partire dal 2017, in cui Jehad Al Dallal e Anas Abdin [1] hanno effettuato una valutazione empirica su come il refactoring object-oriented influisca sugli attributi di qualità, come la coesione, la dimensione, l'ereditarietà, la manutenibilità e la complessità del progetto, attraverso una revisione sistematica della letteratura che raccoglie, riassume e discute i risultati di 76 studi primari rilevanti prima della fine del 2015, riguardanti l'impatto del refactoring su vari attributi di qualità interni ed esterni. L'analisi è avvenuta basandosi su criteri di classificazione, tra cui: gli attributi e le misure della qualità del software, gli scenari di refactoring, gli approcci di valutazione, i dataset, e i risultati sull'impatto. Come risultati, riassumendo, hanno constatato che il refactoring può migliorare alcuni attributi di qualità, ma al contempo può peggiorarne altri, confermato anche in questo lavoro di tesi. Non è dato sapersi il metodo di refactoring, ma essendo una revisione di studi fino al 2015, è possibile dire con certezza che non si sta parlando di refactoring automatico via modelli IA come questa tesi, e che, quindi, gli LLM non erano ancora usati nel campo del Software Engineering. Piuttosto l'approccio dell'articolo citato prevedeva un mix tra refactoring automatici con tool e plugin integrati, refactoring completamente manuali su piccole porzioni di codice e refactoring semi-automatici. Un'altra differenza sostanziale tra la loro analisi e questo lavoro di tesi sta nel tipo di refactoring che si vuol effettuare e su come validarlo. In questa tesi, infatti, il code refactoring si basava su un insieme ristretto di attributi di qualità e metriche. Mentre nell'articolo di Jehad Al Dallal ci si concentrava su un totale di 24 attributi di qualità e 167 metriche distinte, che vanno da quelli calcolati via codice come Coupling, Cohesion e Complexity, a quelli misurati direttamente come Reusability, Testability, Efficiency. Inoltre, rispetto a questa tesi, sono stati studiati diversi scenari di refactoring applicati a determinati attributi di qualità, come Move Method, Extract Method, Extract Class e di come tendono ad avere impatti positivi su coesione e comprensibilità, ma talvolta negativi su complessità o dimensione.

## 6.2 Code Refactoring con IA

Col passare degli anni e con l'avvento dei Large Language Models, questi ultimi si sono ritagliati uno spazio anche nel campo del code refactoring. Esistono diversi approcci e articoli che trattano il refactoring automatico del codice sorgente grazie alla potenza degli LLM. Uno degli approcci esistenti è quello di Bo

Liu [17], in cui vengono mostrati i risultati del refactoring di diversi LLM, nello specifico GPT-4 e Gemini. In paricolare, ci si concentrava su due aspetti: l'identificazione di opportunità di refactoring e raccomandazione di soluzioni di refactoring. Preso un dataset composto da 180 refactoring reali, provenienti da 20 progetti, è stato svolto uno studio empirico volto ad analizzare come i due modelli individuino e poi applichino i refactoring in modo concreto I risultati non sono stati particolarmente buoni. Allora, è stato modificato il prompt e spiegato esplicitamente la sottocategoria di refactoring atteso. Questo ha portato ad un positivo incremento notevole dei risultati, particolarmente per l'identificazione corretta di codice da refattorizzare. Tuttavia, i modelli applicavano ancora refactoring talvolta errati, con nuovi errori sintattici o con modifiche funzionali al programma. Questo evidenzia il rischio di fare refactoring automatico con gli LLM. Per evitare ciò, l'articolo propone un approccio "detect-and-reapply", chiamato RefactoringMirror, con l'obiettivo di evitare refactoring non sicuri, su scenari come Extract Method, Extract Variable, Inline Method, Rename ed altri. Per cui il refactoring generato dagli LLM non viene direttamente applicato al progetto, ma si usa tale RefactoringMirror che analizza il refactoring, confronta le metriche con quelle originali, Se trova un refactoring valido, lo riapplica nel progetto reale usando strumenti affidabili (come il refactoring engine di IntelliJ IDEA) evitando errori di sintassi o bug introdotti dall'LLM, con una riuscita pulita nel 94,2% dei casi. Con questa tecnica, i refactoring identificati dai LLM vengono riapplicati al codice originale usando motori di refactoring ben testati, permettendo di mitigare i rischi associati all'automazione, pur sfruttando l'intelligenza dei LLM per ottenere raccomandazioni preziose. Ulteriori approfondimenti e risultati, si trovano nel continuo dell'articolo. Questo approccio è simile a quello descritto in questa tesi, dove gli agenti guidati da LLM, nel sistema multi agente, applicano tecniche, refactoring e sostituzione di codice senza filtri. Con RefactoringMirror venivano filtrati eventuali errori sintattici. Mentre in questo lavoro di tesi , in caso di errori, si ritorna allo stato precedente il refactoring, come una sorta di rollback, sfruttando la potenza del sistema multi agente.

Ancora con il passare del tempo, gli LLM si sono evoluti e trasformati, andando a formare dei sistemi multi agente autonomi. Perciò, pian piano si sta passando dal singolo Large Language Models a più agenti IA che sfruttano diversi LLM, anche per quanto riguarda il Software Engineering e, in particolare, il code refactoring, proprio come questo articolo di tesi. Un approccio già esistente e simile a questo appena descritto, è quello di Vasanth Rajendran che, insieme ad altri autori e ricercatori, a Marzo 2025 hanno pubblicato il seguente articolo [27]. Hanno descritto come i singoli LLM spesso sono limitati nell'identificare problemi di performance, sicurezza e manutenibilità su cui fare refactoring. L'articolo continua proponendo un nuovo framework, basato, quindi, su un sistema multi agente, in cui: ogni agente è specializzato in un attributo di qualità, gli agenti lavorano insieme in modo cooperativo o competitivo, usando protocolli di coordinazione sofisticati. vengono gestiti meglio i conflitti tra obiettivi di progettazione multipli rispetto ai sistemi a singolo agente, vengono presentate definizioni formali, diagrammi e un piano sperimentale preliminare e l'obiettivo finale è migliorare l'efficienza del processo di sviluppo, ridurre il debito tecnico e aumentare la qualità del software. Per concludere, l'articolo sottolinea come immagina un futuro in cui il refactoring guidato da LLM multi-agente diventi una parte integrante dell'ingegneria del software, riducendo il debito tecnico e migliorando la qualità su più dimensioni contemporaneamente. La differenza con questo lavoro di tesi è quella che l'articolo di Vasanth Rajendran propone un'architettura ideale del sistema multi agente e di come potrebbe funzionare se applicato al refactoring, fornendo ruoli e specializzazioni agli agenti. Mentre in questa tesi l'approccio, nonchè il sistema multi agente, è stato effettivamente sviluppato e testato, seppur non seguendo l'architettura proposta dall'articolo appena citato, applicando ruoli e conoscenze agli agenti più ad alto livello.

# Capitolo 7

# Possibili Sviluppi Futuri

**Aumentare numero di classi:** Il numero di classi scelto per rispondere alla RQ1 e alla RQ2 è stato di 5 e di 10, scelte in modo randomico. Questo per permettere all'approccio (CrewAI) di terminre la sua esecuzione in un tempo ragionevole. Sarebbe molto curioso e stimolante osservare come cambiano i risultati aumentando ancora di più il numero di classi o, addirittura, usarle tutte. Per fare ciò, oltre che avere una minima disponibilità economica se si decide di utilizzare un modello a pagamento, bisogna avere un buon hardware di base, poichè, potenzialmente, l'esecuzione potrebbe durare anche più giorni. Purtroppo, in questo lavoro di tesi, ci si è fermati a 10 classi, ma si immagina che, questo approccio possa funzionare meglio in presenza di elevate risorse hardware ed economiche.

**Impatto modifiche ad Agenti e Task:** Potrebbe esser interessante capire e comprendere come i risultati ottenuti da questo lavoro di tesi, possano cambiare in base alla personalizzazione degli agenti e/o delle task nel sistema multi agente. In particolare, utilizzando un qualsiasi modello LLM, si potrebbe capire come una determinata Backstory dell'agente, magari meno esaustiva e più diretta, impatti rispetto ad un altro più completo. Oppure su come cambiano i risultati in base alla Description che si da ad una task, in modo analogo alla Backstory per un agente. In generale, CrewAI consiglia sempre di essere esaustivi e molto precisi nella descrizione e personalizzazione di agenti e task, così da ottenere risultati migliori e desiderati. Tuttavia, in questo lavoro di tesi, essendo stata fatta un'Ablation Study su task e agenti, come già descritto, non si è tenuto traccia dei risultati ottenuti con una determinata Backstory di un agente, per esempio e un'altra. Il problema principale è che cambiare anche solo un dettaglio può portare a risultati diversi, ma potenzialmente la combinazione di agenti, task e parametri di entrambi è infinta, quindi si è preferito, in questo caso, non tenere traccia dei risultati, ma sarebbe comunque interessante osservare come cambiano i risultati.
Inoltre, come altro metodo per ottenere un refactoring ancora migliore, si potrebbe pensare a fornire degli esempi di refactoring agli agenti (principalmente a quello che fa refactoring) come linea guida. Questo metodo, chiamato few shot learning [30], potrebbe dare agli agenti una maggior conoscenza su come effettuare refactoring, evitando di sviluppare un codice con errori sintattici o con modifiche al funzionamento della classe.
Per concludere, l'ultima idea potrebbe essere quella di modificare la struttura dell'intera crew, cambiando il numero di agenti e/o task, aggiungendo, per esempio, un agente che spiega le modifiche effettuate dal refactoring, per avere un spiegazione testuale, oppure addirittura diminuire il numero di agenti o task e vedere come l'approccio si comporta, per esempio, con un singolo agente.

**Personalizzazione LLM:** Oltre alla personalizzazione di agenti e task, come già accennato, CrewAI consente anche la personalizzazione del modello (o dei modelli) che si stanno utilizzando all'interno del sistema multi agente. A tal proposito, sarebbe opportuno capire, come futuro sviluppo, come variano i risultati andando a modificare e personalizzare i parametri dei modelli. Anche in questo caso, come per agenti e task, i parametri sono stati settati in modo non formale e senza tenere traccia ogni volta dei risultati ottenuti a diversi parametri. Tuttavia, potrebbe essere importante e impattante capire come variano i risultati al variare di parametri degli LLM, come temperatura, top-p, frequency penalty e altri parametri che CrewAI (in particolare LiteLLM) mette a disposizione, come nel seguente articolo [6]. Non solo, sarebbe anche interessante osservare i risultati dell'approccio andando ad utilizzare modelli LLM molto più potenti e costosi oppure, al contrario, modelli più leggeri e completamente gratuiti, e visionare i tempi di esecuzione in entrambi i casi. Anche qui, però, la combinazione tra modelli e personalizzazione dei parametri è potenzialmente infinita.

**Maggior accuratezza nei problemi di plugin:** In questo lavoro di tesi, si sono utilizzati i comandi da terminale già descritti nel Capitolo 4 in cui vengono eseguiti insieme Maven e sonar-scanner, così che Maven gestisce il ciclo di vita e la build del progetto e sonar-scanner effettua l'analisi statica del codice. Per poter migliorare ancor di pià i risultati, si potrebbe pensare di costruire più comandi da terminale, eseguendoli uno per volta. Per spiegare, con il comando attuale, alla crew vengono passati solo gli errori in compilazione che il sistema multi agente ha precedentemente generato per una classe. L'idea sarebbe quella di creare più comandi da terminale da passare alla crew e, quindi, agli agenti. Per esempio, si potrebbe creare una subprocess in cui viene eseguito questo comando: mvn spotbugs:spotbugs -Dspotbugs.outputFile=target/spotbugsXml.xml -Dspotbugs.output=xml . Questa linea serve a trovare gli errori inerenti al plugin FindBugs, usato anche in SonarQube in questo lavoro di tesi. Così facendo, è possibile passare alla crew anche gli eventuali problemi ed errori che questo comando specifico ha generato, poichè la crew, di base, non riesce a identificare proprio tutti i problemi collegati ai plugin utilizzando soltanto la backstory degli agenti e la description delle task. Un altro esempio, è quello di eseguire, sempre con subprocess a parte questo comando: pmd -d ./src -R rulesets/java/quickstart.xml -f text , con cui vengono restituiti gli errori inerenti al plugin di PMD, per poi poterli passare come input alla crew.
Questa tecnica, d'altro canto, potrebbe comportare un grande svantaggio: quello del tempo di esecuzione. Questo perchè si dovrebbero creare 3 nuovi comandi da terminale (uno per ogni plugin su SonarQube scelto in questo caso), il che aumenta il tempo di esecuzione e la possibilità di bug. In più, bisognerà passare alla crew più input (errori rilevati da FindBugs, errori rilevati da PMD, errori rilevati da Checkstyle) e quindi, non si sa a priori il comportamento che avrà l'esecuzione della crew.

**Ulteriore refactoring:** Alla base di ciò c'è il seguente articolo [8], che parla e descrive di come il re-refactoring, ossia l'applicazione del refactoring su un codice già refattorizzato, porti a ulteriori miglioramenti della qualità statica del codice. L'articolo analizza 23 progetti open source, contenenti 29.303 operazioni di refactoring, di cui quasi il 50% costituiscono re-refactoring, effettuati da vari sviluppatori in modo manuale (quindi senza l'uso di LLM), e basando i risultati su attributi quali: coesione, complessità, accoppiamento, ereditarietà e dimensione (linee di codice). L'approccio utilizzato effettuava refactoring su snippet di codice con almeno un attributo di qualità critico. Come risultati Il 65% delle operazioni di refactoring ha migliorato gli attributi associati al tipo di refactoring applicato mentre il restante 35% non ha avuto impatti negativi (gli attributi sono rimasti invariati). Ciò conferma che le pratiche di refactoring rea-

li, quando ben applicate, non peggiorano la qualità del codice. Detto ciò, sarebbe interessante e particolare osservare quanto il re-refactoring funzioni bene nel contesto di questo articolo di tesi, con un approccio basato su sistemi multi agenti e attributi di qualità. L'idea sarebbe quella di rieseguire l'approccio sulle stesse classi già refattorizzate dal sistema e notare se questo porta a ulteriori miglioramenti o, al contrario, dei peggioramenti.

# Capitolo 8

# Conclusioni

L'evoluzione degli LLM continua a portare nuovi approcci, nuove funzionalità e nuovi metodi di studio e di ricerca, sia nel campo del Software Engineering, sia in altri campi di applicazione. In questo lavoro di tesi si è cercato di capire quanto fosse potente ed efficace un sistema multi agente, composto da agenti IA autonomi ognuno dei quali è gestito e controllato da un modello LLM, per effettuare un sicuro ed efficace code refactoring. Quindi migliorare la struttura statica del codice, senza alterarne le funzionalità principali. Per poter osservare come e quanto fosse stato efficace il refactoring, ci si è basato sugli attributi di qualità, quali Security, Reliability, Maintainability più di tutti, oltre Coverage, Tests, Duplicates Lines e Cognitive Complexity. Per cui l'approccio multi agente doveva effettuare un refactoring in modo tale da migliorare tali attributi, senza concentrarsi troppo sulle modalità, ma mantenendo sempre le regole di base che un code refactoring deve rispettare. I risultati, come già osservati nel Capitolo 5, offrono interessanti spunti e punti di vista. In un primo momento, nei progetti Apache, l'approccio sembrava non funzionare; al contrario, nei progetti sviluppati da studenti i risultati sono stati invece discreti. Ciò porta alla conclusione che, per i progetti meno complessi e meno strutturati, questo tipo di approccio proposto potrebbe funzionare e ottimizzare il processo di refactoring che, se fatto in manualmente, potrebbe portare via più tempo. Al contrario, per progetti più complessi, strutturati, commerciali e su cui ci sono svariati team di sviluppo e anni di lavoro alle spalle, l'approccio non è molto efficace e, anzi, talvolta peggiora la qualità del codice. Se, invece, si inietta una conoscenza più precisa e profonda per quanto riguarda specifiche metriche quali vulnerabilities, bugs e code smells, si osservi che l'approccio continua a funzionare bene, sempre per i progetti più semplici. Questo lascia spazio a una sorta di personalizzazione del refactoring, specificando ciò che deve essere migliorato, in base alle proprie specifiche e proprie esigenze. Inoltre, questa tesi pone le basi per sviluppi futuri interessanti, offrendo spunti di sviluppo e ulteriore personalizzazione sia per il refactoring stesso sia, soprattutto, per il sistema multi agente.

# Bibliografia


[1] J. Al Dallal and A. Abdin. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering*, 44(1):44–69, 2018.

[2] R. Barbarroxa, L. Gomes, and Z. Vale. Benchmarking large language models for multi-agent systems: A comparative analysis of autogen, crewai, and taskweaver. In *International Conference on Practical Applications of Agents and Multi-Agent Systems*, pages 39–48. Springer, 2024.

[3] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015.

[4] J. Cordeiro, S. Noei, and Y. Zou. An empirical study on the code refactoring capability of large language models, 2024.

[5] Z. Duan and J. Wang. Exploration of llm multi-agent application implementation based on langgraph+ crewai. *arXiv preprint arXiv:2411.18241*, 2024.

[6] J.-B. Döderlein, N. H. Kouadio, M. Acher, D. E. Khelladi, and B. Combemale. Piloting copilot, codex, and starcoder2: Hot temperature, cold prompts, or black magic?, 2025.

[7] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang. Large Language Models for Software Engineering: Survey and Open Problems . In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53, Los Alamitos, CA, USA, May 2023. IEEE Computer Society.

[8] E. Fernandes, A. Chávez, A. Garcia, I. Ferreira, D. Cedrim, L. Sousa, and W. Oizumi. Refactoring effect on internal quality attributes: What haven't they told you yet? *Information and Software Technology*, 126:106347, 2020.

[9] M. Fu, C. Tantithamthavorn, V. Nguyen, and T. Le. Chatgpt for vulnerability detection, classification, and repair: How far are we?, 2023.

[10] J. He, C. Treude, and D. Lo. Llm-based multi-agent systems for software engineering: Literature review, vision and the road ahead, 2024.

[11] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang. Large language models for software engineering: A systematic literature review, 2024.

[12] S. H. Kan. *Metrics and models in software quality engineering*. Addison-Wesley Professional, 2003.

[13] S. H. Kannangara and W. M. J. I. Wijayanayake. An empirical evaluation of impact of refactoring on internal and external measures of code quality, 2015.

[14] G. Kaur and B. Singh. Improving the quality of software by refactoring. In *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)*, pages 185–191, 2017.

[15] S. Kaur and P. Singh. How does object-oriented code refactoring influence software quality? research landscape and challenges. *Journal of Systems and Software*, 157:110394, 2019.

[16] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, 167:110610, 2020.

[17] B. Liu, Y. Jiang, Y. Zhang, N. Niu, G. Li, and H. Liu. An empirical study on the potential of llms in automated software refactoring, 2024.

[18] B. Liu, H. Zhang, X. Gao, Z. Kong, X. Tang, Y. Lin, R. Wang, and R. Huang. Layoutcopilot: An llm-powered multi-agent collaborative framework for interactive analog layout design, 2025.

[19] D. Lo. Trustworthy and synergistic artificial intelligence for software engineering: Vision and roadmaps, 2023.

[20] V. Mavroudis. Langchain v0.3. *Preprints*, November 2024.

[21] R. Meyes, M. Lu, C. W. de Puiseau, and T. Meisen. Ablation studies in artificial neural networks, 2019.

[22] M. H. Nguyen, T. P. Chau, P. X. Nguyen, and N. D. Q. Bui. Agilecoder: Dynamic collaborative agents for software development based on agile methodology, 2024.

[23] B. Nyirongo, Y. Jiang, H. Jiang, and H. Liu. A survey of deep learning based software refactoring, 2024.

[24] R. Plösch, J. Bräuer, C. Körner, and M. Saft. Measuring, assessing and improving software quality based on object-oriented design principles. *Open Computer Science*, 6(1):187–207, 2016.

[25] D. Pomian, A. Bellur, M. Dilhara, Z. Kurbatova, E. Bogomolov, T. Bryksin, and D. Dig. Next-generation refactoring: Combining llm insights and ide capabilities for extract method. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 275–287, 2024.

[26] V. Rajendran, D. Besiahgari, S. C. Patil, M. Chandrashekaraiah, and V. Challagulla. A multi-agent llm environment for software design and refactoring: A conceptual framework. In *SoutheastCon 2025*, pages 488–493, 2025.

[27] V. Rajendran, D. Besiahgari, S. C. Patil, M. Chandrashekaraiah, and V. Challagulla. A multi-agent llm environment for software design and refactoring: A conceptual framework. In *SoutheastCon 2025*, pages 488–493, 2025.

[28] C. S Arapidis. *Sonar Code Quality Testing Essentials*. Sciendo, 2012.

[29] J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati. Prompt engineering or fine-tuning: An empirical assessment of llms for code, 2025.

[30] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe. Refactoring programs using large language models with few-shot examples. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pages 151–160, 2023.

[31] P. Venkadesh, S. Divya, and K. S. Kumar. Unlocking ai creativity: A multi-agent approach with crewai. *Journal of Trends in Computer Science Smart Technology*, 6(4):338–356, 2024.

[32] S. Wang, X. Hu, B. Wang, W. Yao, X. Xia, and X. Wang. Insights into deep learning refactoring: Bridging the gap between practices and expectations, 2024.

[33] D. Weyns, H. Parunak, and O. Shehory. The future of software engineering and multi-agent systems. *International Journal of Agent-Oriented Software Engineering*, 2(1):369–377, 2009.

[34] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design, 2023.

[35] M. Wooldridge. Agent-based software engineering. *IEE Proceedings-software*, 144(1):26–37, 1997.

## Ringraziamenti

Se stai leggendo questi ringraziamenti significa che hai letto tutta la mia tesi, e quindi spero che ti sia piaciuta, oppure non hai letto neanche una pagina e sei arrivato direttamente qui, come fanno tutti. In ogni caso sarò molto breve nel ringraziare chi merita di essere ringraziato.
Per prima cosa vorrei ringraziare il mio relatore di tesi, Juri Di Rocco, e il mio correlatore, Riccardo Rubei, per la loro disponibilità e per la loro pazienza. Se mi sono divertito nell'ideare e sviluppare questo lavoro di tesi il merito va dato soprattutto a loro. Dopodiché è doveroso ringraziare tutti coloro che sono stati al mio fianco durante questo percorso. Vorrei ringraziare tutta la mia famiglia. In particolare vorrei ringraziare mia mamma e mia sorella, sempre con me dal primo giorno, sempre con una parola di conforto, sempre pronte a sostenermi e a darmi un consiglio, e mio padre, punto di riferimento ed esempio di vita (vorrei che tu fossi qui con me a festeggiare, a fare le riprese con la videocamera come eri solito fare per queste occasioni, ma so già che lo stai facendo da lassù). Vi voglio bene.
Ci tengo, poi, a ringraziare tutti i miei amici 'de lu paes' di Villamagna. Non serve fare nomi, tanto siamo sempre i soliti e ci metterei fin troppo tempo a elencarvi tutti. Ci conosciamo da una vita quindi qui non ho nulla da dire che non sapete già. Però una cosa la voglio aggiungere: grazie per tutte le risate e per la vostra presenza costante. Spero che, così come siamo cresciuti, raggiungeremo tutti insieme i nostri obiettivi e che un giorno, seduti ad un bar, ricorderemo i tempi passati sempre con gioia. Trovare un gruppo come il nostro è raro. Grazie a ognuno di voi.
Vorrei ringraziare gli amici con cui ho trascorso questo periodo universitario a Casa Casereccia. Grazie per tutte le volte in cui abbiamo preferito dormire e giocare al telefono all'Università piuttosto che studiare. Se ho impiegato un po' di tempo in più per laurearmi è anche colpa vostra. Però quanti bei momenti e quante risate.
Vorrei, infine, ringraziare tutti coloro che in questi anni in un modo o nell'altro mi sono stati vicino, chi ancora oggi, chi non più.
Detto ciò, si conclude qui il primo vero traguardo di vita. Che sia solo l'inizio di un cammino pieno di soddisfazioni, da vivere con la stessa passione e leggerezza che mi hanno accompagnato fin qui.

A M.M. !!! Ercole Rutolo, L'Aquila, Luglio 2025