POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Studio esplorativo delle soluzioni AI-based per la generazione del codice

Relatori
Prof. Riccardo COPPOLA
Dott. Tommaso FULCINI

Candidato
Claudio SCALA

Luglio 2025

Abstract

Questo studio esamina l'efficacia dei Large Language Models (LLM) nella generazione automatica di codice per progetti software, con particolare attenzione alla correttezza, qualità e manutenibilità del codice prodotto. L'obiettivo principale è valutare la capacità dei LLM di generare codice preciso, semplice, conforme a specifiche metriche qualitative. La ricerca è articolata in due fasi: una prima analisi su un insieme limitato di esercizi di programmazione per testare le capacità dei modelli, le tecniche di prompt engineering e le metriche di valutazione; e una seconda fase estesa, con 150 esercizi, per validare i risultati preliminari. I risultati confermano l'elevato potenziale dei LLM, con un tasso di successo del 90,44% nei test finali. La qualità del codice generato risulta generalmente buona, sebbene gli esercizi più complessi richiedano ottimizzazioni del prompt o un approccio iterativo. L'adozione di tecniche di prompting, come prompt unificati, iterativi, Chain of Thought (CoT) e correttivi, si dimostra cruciale per migliorare le prestazioni dei modelli. La discussione approfondisce i risultati in termini di accuratezza, qualità del codice, impatto delle tecniche di prompting e applicabilità pratica, evidenziando punti di forza e criticità dei vari LLM considerati. In conclusione, vengono proposte direzioni future, come l'estensione ad altri linguaggi di programmazione, l'analisi di problemi più complessi e la creazione di ambienti collaborativi tra LLM per simulare dinamiche di team di sviluppo. Lo studio conclude che i LLM rappresentano una tecnologia promettente per supportare e ottimizzare il processo di sviluppo software, a condizione che l'interazione con i modelli sia ben progettata e guidata da tecniche efficaci di prompt engineering.

Ringraziamenti

Voglio ringraziare tutte le persone che hanno contribuito in modo significativo alla realizzazione di questo lavoro: in particolare i miei relatori, il Prof. Riccardo Coppola e il Dott. Tommaso Fulcini, i quali ringrazio per la disponibilità, la guida e i preziosi consigli durante tutto il percorso di tesi.

Ringrazio i docenti del corso di laurea per avermi fornito degli ottimi strumenti tecnici e critici fondamentali per affrontare il mondo.

Un pensiero speciale va alla mia famiglia, per il sostegno incondizionato, la pazienza e l'incoraggiamento che non mi hanno mai fatto mancare.

Infine, ringrazio i miei amici con cui ho condiviso momenti di studio, confronto e crescita personale.

Indice

Abstract 1

Elenco delle figure 8

Glossario 10

1 Introduzione 11

2 Background 13

2.1 LLM . . . 13

2.2 Gli LLM nella generazione del codice . . . 13

2.3 Correttezza del codice . . . 13

2.3.1 Valutazione della correttezza del codice . . . 13

2.4 Qualità del codice . . . 14

2.4.1 Valutazione della qualità del codice . . . 14

2.5 Prompt Engineering . . . 15

2.5.1 Principi Fondamentali . . . 15

2.5.2 Elementi di un Prompt . . . 16

2.5.3 Tecniche di prompting . . . 16

3 Metodologia di Ricerca 25

3.1 Valutazione preliminare . . . 26

3.1.1 Obiettivi della Valutazione preliminare . . . 26

3.1.2 Selezione degli esercizi . . . 26

3.1.3 Tecniche utilizzate per la correttezza del codice . . . 28

3.1.4 Tecniche utilizzate per la qualità del codice . . . 29

3.1.5 Tecniche utilizzate per la correttezza e la qualità del codice 30
3.1.6 Metriche di valutazione 31
3.2 Valutazione su larga scala dell'efficacia degli LLM 31
3.2.1 Obiettivo 31
3.2.2 Tecnica utilizzata 31
3.3 Considerazioni Finali 32

4 Valutazione preliminare dell'efficacia degli LLM 33
4.1 Esercizio 1: Calcolare il resto con il minor numero possibile di monete 34
4.1.1 GEMINI 2.0 34
4.1.2 GPT 35
4.1.3 ClaudeSonnet 36
4.1.4 Conclusioni sulla correttezza primo esercizio 38
4.1.5 Conclusioni sulla qualità primo esercizio 38
4.1.6 Sintesi dell'esercizio 1 39
4.2 Esercizio 2: Algoritmo di ricerca binaria 39
4.2.1 Analisi correttezza del codice 39
4.2.2 Analisi qualitativa del codice 39
4.2.3 Prompt Ottimizzazione Qualitativa 40
4.2.4 Conclusioni sulla correttezza secondo esercizio 41
4.2.5 Conclusioni sulla qualità secondo esercizio 41
4.2.6 Sintesi dell'esercizio 2 41
4.3 Esercizio 3: Rest Api 42
4.3.1 Gemini 2.0 42
4.3.2 GPT 43
4.3.3 ClaudeSonnet 44
4.3.4 Approccio collaborativo: prompt design con GPT e code generation con Claude 45
4.3.5 Richiesta a GPT di generazione di un prompt per il miglioramento delle metriche con codice già esistente 47
4.3.6 Conclusioni sulla correttezza terzo esercizio 47
4.3.7 Conclusioni sulla qualità terzo esercizio 47
4.3.8 Conclusioni sulla tecnica di generazione prompt da GPT e generazione codice da Claude 48

4.3.9 Sintesi dell'esercizio 3 48
4.4 Esercizio 4: Account bancario 49
4.4.1 Analisi correttezza del codice 49
4.4.2 Analisi qualitativa del codice 49
4.4.3 Conclusioni sulla correttezza quarto esercizio 51
4.4.4 Conclusioni sulla qualità quarto esercizio 51
4.4.5 Sintesi dell'esercizio 4 51
4.5 Esercizio 5: Somma dei numeri in un Item in parallelo - RUST 52
4.5.1 Gemini 2.0 52
4.5.2 GPT 53
4.5.3 Claude 54
4.5.4 Approccio collaborativo: prompt design con Claude e code generation con GPT 55
4.5.5 Richiesta a Claude di generazione di un prompt per il miglioramento delle metriche con codice dal prompt unificato 56
4.5.6 Conclusioni sulla correttezza quinto esercizio 56
4.5.7 Conclusioni sulla qualità quinto esercizio 56
4.5.8 Conclusioni sulla tecnica di generazione prompt da Claude e generazione codice da GPT 57
4.5.9 Sintesi dell'esercizio 5 57
4.6 Esercizio 6: Frequenza lettere in un file - Calcolo parallelo - RUST 58
4.6.1 Gemini 2.0 58
4.6.2 GPT 59
4.6.3 ClaudeSonnet 60
4.6.4 Conclusioni sulla correttezza sesto esercizio 61
4.6.5 Conclusioni sulla qualità sesto esercizio 62
4.6.6 Sintesi dell'esercizio 6 62

5 Valutazione su larga scala dell'efficacia degli LLM 63
5.1 Modello utilizzato 63
5.2 Esercizi selezionati 63
5.2.1 Metodologia delle generazioni di codice 65
5.2.2 Costruzione del Prompt 65
5.3 Risultati e Grafici 67

5.4 Conclusioni sulla valutazione 75
5.4.1 Correttezza del codice 75
5.4.2 Qualità del codice 75

6 Discussioni Generali 77
6.1 Correttezza del Codice - Valutazione preliminare 77
6.2 Qualità del Codice - Valutazione preliminare 78
6.3 Tecniche di Prompting 78
6.4 Risultati del Test Conclusivo 79
6.5 Applicabilità Pratica 79

7 Conclusione 80
7.1 Lavori futuri 80
7.1.1 Eseguire tutte le tecniche per ogni LLM, integrando anche con altri LLM 80
7.1.2 Valutazioni su altri linguaggi di programmazione 81
7.1.3 Valutazioni su esercizi più complessi 81
7.1.4 Progettazione di un ambiente dove gli LLM lavorino insieme 81

Elenco delle figure

2.1 Elementi di un prompt 16
2.2 Generate Knowledge Prompting 19
2.3 Caso di studio proposto da Wei e al. 21
2.4 Differenze tra logic prompting 22

5.1 Complessità ciclomatica Sum (Istogramma e Violin Plot) 67
5.2 Complessità ciclomatica Average (Istogramma e Violin Plot) 68
5.3 Complessità Cognitiva Sum (Istogramma e Violin Plot) 69
5.4 Complessità cognitiva Average (Istogramma) 70
5.5 Difficoltà di Halstead (Istogramma e Violin Plot) 71
5.6 Distribuzione dello Sforzo di Halstead (log10) (Istogramma e Violin Plot) 72
5.7 Indice Manutenibilità (Istogramma e Violin Plot) 73
5.8 Istogramma WMC Ponderato 74

Glossario

addestramento Processo di apprendimento automatico attraverso cui un modello acquisisce conoscenze da dataset di training

complessità Misura delle risorse computazionali richieste da un algoritmo in termini di tempo e spazio

correttezza del codice Capacità del codice di eseguire la funzione richiesta senza errori e di produrre output conformi alle specifiche

leggibilità Chiarezza e comprensibilità del codice, determinata da naming conventions, struttura e documentazione

LLM Large Language Model, modello linguistico di grandi dimensioni preaddestrato su grandi quantità di dati

metriche di valutazione Parametri quantitativi e qualitativi utilizzati per misurare le prestazioni e la qualità del codice generato automaticamente

prompt engineering Tecnica di ottimizzazione delle istruzioni fornite agli LLM per migliorarne le prestazioni e la qualità degli output generati

robustezza Capacità del codice di gestire input non previsti, condizioni eccezionali e mantenere stabilità in situazioni critiche

Capitolo 1

Introduzione

Il mondo dei progetti software sta vivendo una profonda trasformazione grazie all'introduzione dei modelli linguistici di grandi dimensioni (LLM) e il loro impiego nella generazione del codice di programmazione. Questi modelli hanno dimostrato una straordinaria abilità nel comprendere il linguaggio naturale e tradurlo in codice di programmazione, portando sempre più ingegneri del software a utilizzarli come supporto nella scrittura del codice.
La possibilità di agevolare e snellire un'attività tradizionalmente demandata a sviluppatori esperti, rappresenta un'opportunità significativa per aumentare l'efficienza e ridurre i costi di sviluppo software. Tuttavia, nonostante gli enormi progressi, ancora poco è stato verificato su aree legate principalmente alla correttezza, all'efficienza e alla qualità del codice prodotto.
Comprendere come guidare gli LLM per la produzione di un codice sempre più soddisfacente rispetto a dei requisiti specifici, quali correttezza, robustezza, complessità , etc. è fondamentale per un'evoluzione consapevole nel mondo della progettazione software. Va specificato che un codice eseguibile non equivale necessariamente a codice efficiente e robusto, specialmente nel contesto dello sviluppo software applicato al mondo reale. Ad oggi queste metriche qualitative relative alla generazione di codice da parte degli LLM risultano poco esplorate.
Questa tesi si propone di approfondire le capacità degli nella generazione del codice per progetti software, ponendo un importante focus alla qualità del codice generato. L'analisi viene svolta attraverso l'utilizzo di diverse metriche di valutazione e di tecniche prompt engineering.
L'obiettivo principale è quello di misurare non solo la correttezza del codice generato, ovvero la sua capacità di eseguire la funzione richiesta senza errori, ma anche la sua qualità, in termini di leggibilità, manutenibilità e aderenza alle migliori pratiche di programmazione.
La tesi è strutturata con l'obiettivo di fornire al lettore una panoramica completa sia del contesto teorico che dei risultati della ricerca condotta. Dopo un'introduzione generale, il Capitolo II presenta il background teorico necessario per comprendere il lavoro svolto: vengono illustrati i concetti fondamentali relativi ai modelli linguistici di grandi dimensioni (LLM), il loro funzionamento e i meccanismi di addestramento, nonché le principali metriche utilizzate per valutare la qualità del codice generato. Viene inoltre approfondito il ruolo del prompt engineering come elemento chiave per ottimizzare le prestazioni dei modelli. Il Capitolo III descrive nel dettaglio la metodologia adottata nella ricerca. Vengono delineati gli obiettivi dello studio, i modelli LLM selezionati, le tecniche di prompt engineering sperimentate e i criteri utilizzati per la valutazione del codice generato. I Capitoli IV e V sono dedicati alla presentazione dei risultati ottenuti. Il Capitolo 4 si concentra su una valutazione preliminare, svolta su un insieme ristretto di esercizi di programmazione, finalizzata a esplorare diverse strategie di prompting e individuare

quelle più efficaci. Il Capitolo V estende l'analisi a un set più ampio di problemi, con l'obiettivo di ottenere risultati statisticamente significativi e consolidare le evidenze emerse nella fase iniziale. Il Capitolo VI raccoglie e discute criticamente i risultati, analizzando le principali evidenze emerse, i punti di forza e le limitazioni riscontrate nell'impiego degli LLM per la generazione automatica di codice. Infine, nel Capitolo VII vengono presentate le conclusioni, sintetizzando i principali contributi della tesi e riflettendo sulle implicazioni che l'adozione degli LLM potrebbe avere sul futuro del processo di sviluppo software e vengono proposte possibili direzioni per lo sviluppo futuro della ricerca, tra cui l'estensione dello studio ad altri linguaggi di programmazione, l'analisi di problemi più complessi e l'esplorazione di ambienti collaborativi tra modelli.

Capitolo 2

Background

2.1 LLM

Un Large Language Models è un tipo di programma di Intelligenza artificiale, il quale viene addestrato su una grande mole di dati ed è in grado di elaborare e generare diverse tipologie di contenuto[5]. I modelli, durante questo processo, esaminano e elaborano i collegamenti presenti nei dati, identificando tra questi le relazioni più probabili e commprendendo il loro significato attraverso una analisi del contesto.

2.2 Gli LLM nella generazione del codice

I Large Language Models nella la generazione di codice si riferiscono all'utilizzo di questi modelli per produrre codice sorgente a partire da descrizioni e specifiche in linguaggio naturale. Generalmente, queste descrizioni in linguaggio naturale includono enunciati di problemi di programmazione e possono, opzionalmente, fornire anche un contesto di programmazione aggiuntivo. [14].

In questa ricerca si analizzerà gli LLM in base alla loro capacità nella generazione di codice.

2.3 Correttezza del codice

Un codice generato dagli LLM è considerato corretto quando supera con successo tutti i test di controllo e produce risultati conformi con le specifiche richieste.

2.3.1 Valutazione della correttezza del codice

Il codice generato dall'LLM è stato valutato sia attraverso i test di controllo sia in base all'output prodotto date le istruzioni fornite.

2.4 Qualità del codice

La qualità di un codice generato dagli LLM è data dal rispetto e conformità alle migliori pratiche di programmazione in un contesto di sviluppo software.

Queste caretteristiche si possono riassumere nella semplicità del suo codice, nella possibilità di correzione degli errori, nella sua comprensione e nella manutenibilità [7].

2.4.1 Valutazione della qualità del codice

Per la valutazione del codice generato dagli LLM è stato utilizzato uno strumento chiamato Rust-Code Analysis; "una libreria Rust impiegata per analizzare ed estrarre informazioni da codice sorgente scritto in linguaggi di programmazione diversi. Il processo si basa su un generatore di parser e su una libreria di parsing incrementale chiamata Tree Sitter" [17, 22].

Metriche utilizzate da Rust-Code Analysis

1. **CC (Complessità Ciclomatica di McCabe):** questa metrica misura la complessità del codice sulla base del flusso di controllo generato da esso; viene calcolata come il numero di percorsi linearmente indipendenti attraverso un frammento di codice considerando principalmente il numero di vertici, spigoli e componenti connesse in un grafico che rappresenta il flusso di controllo. Il valore è inversamente proporzionale alla complessità del codice analizzato; un valore basso indica una complessità minore. I valori medi si collocano generalmente tra 1 e 2 [16].

2. **Difficoltà di Halstead:** conosciuta anche come "propensione all'errore", quantifica il livello di difficoltà che uno sviluppatore potrebbe incontrare nel produrre codice corretto e privo di bug. Il valore è inversamente proporzionale alla propensione del codice analizzato a generare errori. I valori medi rientrano generalmente tra 10 e 100 anche per frammenti di codice di piccole dimensioni, senza un limite superiore [12].

3. **Sforzo di Halstead:** è una metrica che quantifica l'impegno computazionale e cognitivo necessario per comprendere un programma ed è direttamente proporzionale alla Difficoltà di Halstead e alla dimensione (Nlog2(n)) del programma. I valori sono tipicamente superiori a 10^5 per artefatti di piccole dimensioni. In questo studio viene assunto il log10 di questa metrica per l'analisi [12].

4. **MI (Indice di Manutenibilità):** Una metrica derivata per misurare la facilità di manutenzione di una codebase. Valori più alti indicano output migliori:

   - Bassa manutenibilità: MI < 10
   - Media manutenibilità: 10 <= MI < 20
   - Alta manutenibilità: MI >= 20

5. **WMC (Metodo Ponderato per Classe):** definito come la somma delle complessità dei metodi locali di una classe, questa metrica è utilizzata per misurare la complessità combinata di tali metodi, per ciascuno di essi viene calcolata la complessità ciclomatica (CC) e successivamente il valore della somma totale viene assegnato alla classe. Anche in questo caso si presenta una relazione inversamente proporzionale: valori più bassi indicano una complessità inferiore [3].

6. **Complessità Cognitiva:** Misura quanto è difficile capire intuitivamente un'unità di codice, questo viene misurato esaminando i pesi cognitivi delle strutture di controllo del software. Lo strumento rust-code-analysis implementa la definizione della complessità cognitiva data da SonarSource [2]. Valori più bassi indicano un codice più facile da interpretare. I valori generalmente sono inferiori a 10.

2.5 Prompt Engineering

Il prompt engineering rappresenta un insieme di competenze sempre più importanti per utilizzare efficacemente i modelli di linguaggio ed è lo strumento attraverso il quale i LLM vengono programmati [5, 6, 19]. Questa pratica è fondamentale per ottimizzare l'utilizzo di un LLM: un prompt consiste in un insieme di istruzioni a esso fornite che ne personalizzano e potenziano le capacità attraverso un framework iniziale stabilito di regole, vincoli o linee guida specifiche che possono influenzare le interazioni successive e le risposte generate dal modello. [6]

Un prompt definisce il quadro di riferimento per il dialogo e guida il LLM nell'identificare le informazioni pertinenti, specificando inoltre il formato e il tipo di risposta attesa. [15].

Il prompt engineering

Per mostrare l'efficacia e le potenzialità del prompt engineering, forniamo il seguente prompt:

```
Da ora in poi, vorrei che mi facessi domande per distribuire un'applicazione Python su AWS.
Quando avrai abbastanza informazioni per distribuire l'applicazione,
crea uno script Python per automatizzare la distribuzione.
```

Questo esempio fa sì che l'LLM inizi a porre domande all'utente riguardo la propria applicazione software. Esso guiderà il processo di formulazione delle domande fino a raggiungere un punto in cui ha sufficienti informazioni per generare uno script Python che automatizza la distribuzione. Questo esempio dimostra il potenziale di programmazione dei prompt.[26].

2.5.1 Principi Fondamentali

1. **Specificità:** si riferisce alla precisione e alla chiarezza con cui un prompt è formulato. Un prompt specifico fornisce dettagli chiari e contestuali che guidano il modello verso una risposta mirata e pertinente. Maggiore è la specificità, più mirato sarà l'output ottenuto [6, 15]. Questo aspetto è particolarmente importante quando si cerca un risultato o uno stile di generazione specifico. Non esistono token o parole chiave che portano a risultati migliori. "È più rilevante avere un buon formato e un prompt descrittivo. Al fine di ottenere i risultati desiderati in formati specifici risulta efficace inserire esempi nel prompt." [6]
2. **Approccio Graduale:** implica la suddivisione di compiti complessi in parti più piccole e gestibili. Questo metodo facilita la comprensione e la risoluzione di problemi complessi, permettendo di affrontare ogni parte in modo sistematico [6].
3. **Iterazione e Miglioramento:** questo principio si basa sull'idea di rielaborare e perfezionare gli input attraverso interazioni iterative con il modello. Ogni iterazione fornisce un'opportunità per affinare il prompt e migliorare la qualità dell'output [6, 19].

2.5.2 Elementi di un Prompt

Un prompt può contenere uno dei seguenti elementi:

- **Istruzione** - riguarda un compito o un'istruzione specifica che il modello deve eseguire
- **Contesto** - può coinvolgere informazioni esterne o contesti aggiuntivi che possono indirizzare il modello verso risposte migliori.
- **Dati in Input** - è l'input o la domanda per la quale ci interessa ottenere una risposta.
- **Indicatore dell'Output** - indica il tipo o il formato dell'output.

[9]

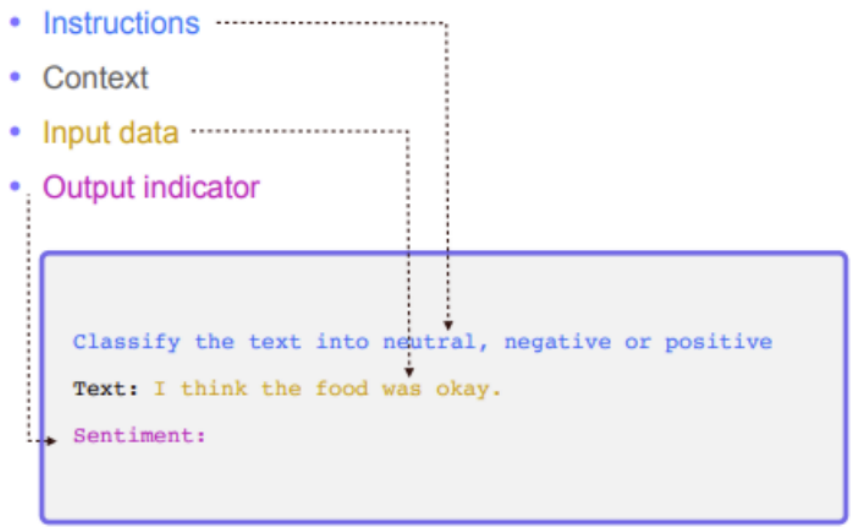


Figura 2.1: Elementi di un prompt

"È possibile progettare prompt efficaci per vari compiti semplici, utilizzando comandi per indicare al modello ciò che si desidera ottenere, quali "Scrivi", "Classifica", "Riassumi", "Traduci", "Ordina", ecc. "[6].

2.5.3 Tecniche di prompting

Di seguito si discute delle principali tecniche utilizzate per il prompting[5].

Priming

"La pratica di fornire un input iniziale al modello prima di generare una risposta. L'input iniziale mira a guidare il modello verso la generazione di una risposta più coerente"[5].

Senza priming:

- "Scrivi una storia su un esploratore che scopre una nuova isola."

Con priming:

- "Immagina un mondo in cui la natura è selvaggia e inesplorata, e gli esploratori utilizzano mappe antiche e strumenti tradizionali per navigare. L'esploratore in questa storia è noto per il suo coraggio e la sua determinazione."

Tabular format prompting

"Questa tecnica indica all'LLM di utilizzare il formato tabellare per una chiara organizzazione e presentazione dei dati. In tal modo risulta più accessibile all'utente analizzare e comprendere l'output." [5].

Esempio:

- P1: scrivi una storia su un esploratore che scopre una nuova isola
- P2: in quali diverse categorie puoi suddividere la tua risposta per renderla più descrittiva?
- P3: ora crea una tabella che includa la tua risposta originale con queste categorie separate in colonne diverse

RGC

Questa tecnica fornisce delle istruzioni in categorie precise per aiutarlo nell'organizzazione e nella generazione dell'output:

"Ruolo, Risultato, Obiettivo, Contesto, Vincolo":

- Ruolo: Il personaggio dell'LM (Modello Linguistico)
- Risultato: L'output desiderato
- Obiettivo: Lo scopo dell'output
- Contesto: Chi, cosa, dove, perché (le circostanze)
- Vincolo: Limitazioni e linee guida

[5].

Esempio:

- Ruolo: Scrivi un racconto di un avventuriero alla ricerca di un'isola perduta.
- Obiettivo: Cattura l'attenzione del lettore in una storia intrigante che racconti il viaggio di questo personaggio.
- Contesto: Una storia ambientanta nell'era dei pirati, dove il protagonista viaggia in mare a bordo della sua nave
- Risultato: Rivela gradualmente comesi sviluppa la trama e aggiungi qualche elemento di sopresa
- Vincolo: La narrazione deve essere coerente, culminando in un lieto fine che lega insieme tutti gli elementi misteriosi.

Few-shot prompting

"Questa tecnica prevede di fornire ai modelli alcuni esempi di input-output per indurre la comprensione di un determinato compito. Un'attenta progettazione del prompt è fondamentale per ottenere prestazioni ottimali e mitigare bias aggiunti indesiderati" [5].

Esempio:

- I numeri in questo gruppo sono tutti divisibili per 3: 3, 6, 7, 11, 14.
  - Risposta: la risposta è falsa
- I numeri in questo gruppo sono tutti divisibili per 3: 21, 18, 24, 6, 60, 30, 27.
  - Risposta: la risposta è vera
- I numeri in questo gruppo sono tutti divisibili per 5: 10, 15, 50, 20, 35, 40, 100.
  - Risposta: la risposta è vera
- I numeri in questo gruppo sono tutti divisibili per 5: 7, 18, 20, 35, 50, 5.
  - Risposta: la risposta è falsa
- I numeri in questo gruppo sono tutti divisibili per 7: 14, 21, 56, 70, 84, 63, 9.
  - Risposta: ?

Generate Knowledge Prompting

" Questa tecnica prevede l'utilizzo di conoscenze aggiuntive, fornite come contesto, per migliorare i risultati su compiti complessi

La conoscenza utilizzata viene generata da un altro modello e utilizzata nel prompt per fare una previsione. Si utilizza quindi la previsione con il livello di confidenza più elevato. " [5].

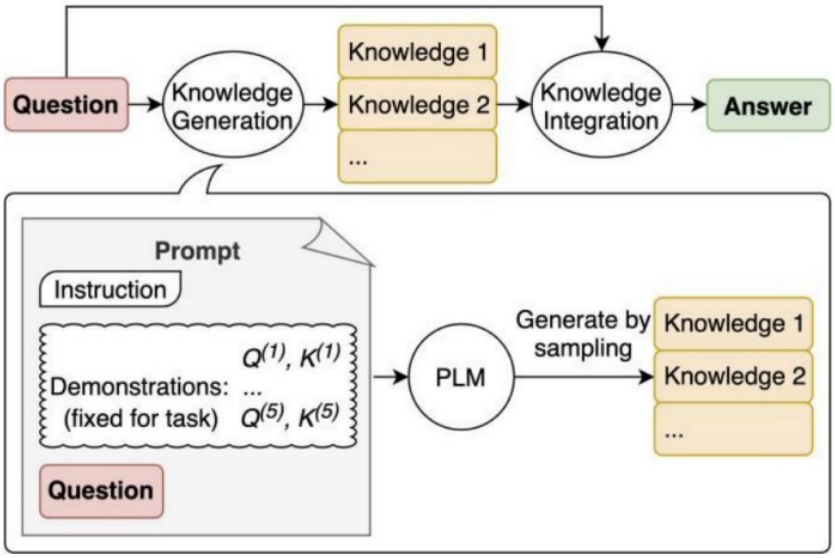


Figura 2.2: Generate Knowledge Prompting

Esempio:

- **Primo Passo**
  - **Input:** La Grecia è più grande del Messico.
  - **Conoscenza:** La Grecia è di circa 131.957 km quadrati, mentre il Messico è di circa 1.964.375 km quadrati, il che rende il Messico più grande della Grecia del 1.389%.
  - **Input:** Un siero ha la stessa consistenza di un'emulsione fluida.
  - **Conoscenza:** Un'emulsione fluida è una formulazione cosmetica caratterizzata da una consistenza leggera e facilmente assorbibile, con una viscosità compresa tra 1.000 e 5.000 centipoise. Le emulsioni fluide sono più leggere delle creme tradizionali (viscosità superiore ai 10.000 centipoise). Questa texture permette una rapida penetrazione cutanea mantenendo proprietà idratanti efficaci.
  - **Input:** Bere molta acqua disidrata il corpo.
    1. **Conoscenza 1:** L'acqua è fondamentale per mantenere l'idratazione dell'organismo. Un'adeguata assunzione idrica supporta la regolazione della temperatura corporea, la funzione renale e il trasporto di nutrienti. Bere molta acqua, entro limiti fisiologici, aiuta a prevenire la disidratazione.

2. **Conoscenza 2:** La disidratazione si verifica quando si perdono più liquidi di quanti ne vengano introdotti. Un consumo estremo può effettivamente comportare degli squilibri elettrolitici, ma ciò non equivale a essere disidratato.

- **Secondo Passo**
  - **Primo esempio (Previsione ad alta confidenza):**
    - **Domanda:** Bere molta acqua disidrata il corpo. Sì o No?
    - **Conoscenza:** L'acqua è fondamentale per mantenere l'idratazione dell'organismo. Un'adeguata assunzione idrica supporta la regolazione della temperatura corporea...
    - **Spiegazione e risposta:** No, assumere tanta acqua non causa disidratazione, anzi, essa contribuisce a rispettare l'equilibrio dei liquidi corporei, soprattutto in condizioni normali.
  - **Secondo esempio (Previsione a bassa confidenza):**
    - **Domanda:** Bere molta acqua disidrata il corpo. Sì o No?
    - **Conoscenza:** La disidratazione avviene quando si perdono più liquidi di quanti ne vengano introdotti. Un consumo estremo può causare squilibri elettrolitici...
    - **Spiegazione e risposta:** Sì, bere molta acqua disidrata il corpo. In alcuni casi, un eccesso d'acqua può provocare uno squilibrio nei livelli di sodio nel sangue, situazione che ha il nome di iponatriemia, la quale genera sintomi simili a quelli della disidratazione

Chain of Thought prompting

"Introdotto per far fronte ai limiti degli LM (Modelli Linguistici) nel ragionamento complesso. L'obiettivo è quello di facilitare processi di ragionamento coerenti e graduali: ovvero un processo iterativo. " [5]

Fase 1: Impostare la scena. Descrivi l'ambiente della storia. Dove e quando si svolge la storia? Includi dettagli sull'ambiente, il periodo di tempo e l'atmosfera generale.

La storia si svolge a Neo-Tokyo 3, una città costruita dopo un evento apocalittico, nel quale gli umani riescono a condurre vite normali, ma sono costretti a proteggersi da delle figure aliene che cercando di distruggere la città. Immaginati un contesto post-moderno, il clima è molto simile a quello dei giorni nostri ma la tecnologia umana ha fatto molti progessi

Fase 2: Presentazione del detective. Quindi, presenta il personaggio principale, il detective. Descrivi il loro aspetto, la personalità e qualsiasi abilità o gadget unici che possiedono. Cosa li distingue?

La detective Jessica Mina è una donna determinata, dallo sguardo profondo e concentrato. Indossa una giacca tecnica rinforzata, adatta al clima instabile delle stazioni orbitali, e un visore multifunzione collegato direttamente alla rete investigativa centrale. Jessica è celebre per la sua mente analitica, la calma glaciale anche sotto pressione, e la sua capacità di ricostruire eventi complessi da pochi indizi sparsi.

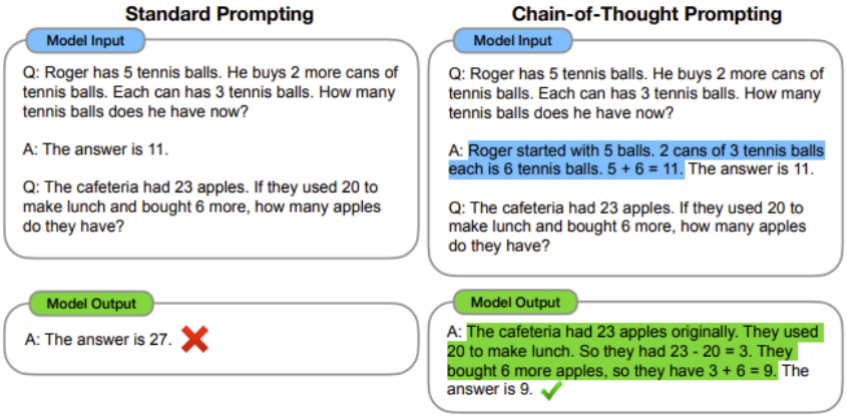


Figura 2.3: Caso di studio proposto da Wei e al.

Self Consinstency with CoT (CoT-SC)

"Per compiti di ragionamento complessi con più percorsi validi. Genera diverse catene di ragionamento e quindi identifica la risposta finale più coerente." [5]

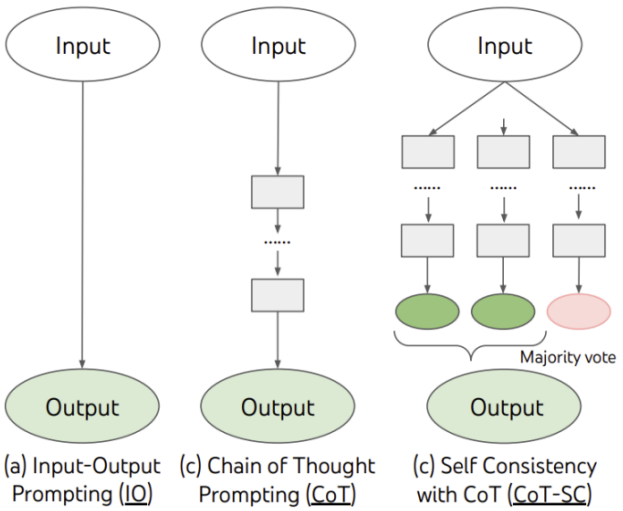


Figura 2.4: Differenze tra logic prompting

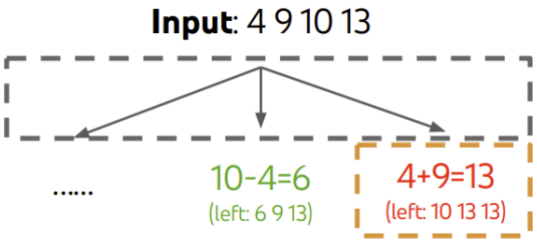
Tree of Thoughts (CoT-SC)

"Estende CoT progettando e suddividendo i ragionamenti organizzandoli in una una struttura ad albero, noti come «pensieri». Su ognuno di questi pensieri intermeti viene applicato un processo di ragionamento e valutazione. " [5]

1. **Thought decomposition:** Mentre CoT campiona i pensieri in modo coerente senza una decomposizione esplicita, ToT sfrutta le proprietà del problema per progettare e decomporre i passaggi intermedi dei pensieri.

	Gioco del 24	**Scrittura Creativa**	**Cruciverba 5x5**
Input	4 numeri (4 9 10 13)	4 frasi casuali	10 indizi (o1. presentato;...)
Output	Un'equazione per raggiungere 24 (13-9)*(10-4)=24	Un testo di 4 paragrafi che termina con le 4 frasi	5x5 lettere: MOSTRATO; WIRRA; DISPON; ...
Pensieri	3 equazioni intermedie (13-9=4 (restano 4,4,10); 10-4=6 (restano 4,4,6); 4*6=24)	Un breve piano di scrittura (1. Introduci un libro che connette...)	Parole da inserire per gli indizi: (o1. mostrato; v5. inchiodato; ...)
N. passaggi ToT	3	1	5-10 (variabile)

2. **Thought generation:** genera k candidati per i successivi passi



3. **State evaluator:** Dato un insieme eterogeno di stati, il valutatore di stato analizza i progressi compiuti verso la soluzione del problema, fungendo da euristica per l'algoritmo di ricerca con lo scopo di determinare quali stati continuare a esplorare e in quale ordine.

   Valutazione: Un prompt di valutazione ragiona sullo stato s per generare un valore scalare v (ad esempio da 1 a 10) o una classificazione (ad esempio certo/probabile/impossibile) che potrebbe essere trasformata in un valore.

   Votazione: gli stati vengono eliminati tramite un confronto deliberato tra diversi stati in S in un prompt di votazione.

4. **Search algorithm:** algoritmi di ricerca ad albero per selezionare i rami. Due diversi approcci principali:

   - **Valutazione:** Un prompt di valutazione ragiona sullo stato s per generare un valore scalare v (ad esempio da 1 a 10) o una classificazione (ad esempio certo/probabile/impossibile) che potrebbe essere trasformata in un valore.
   - **Votazione:** gli stati vengono eliminati tramite un confronto deliberato tra diversi stati in S in un prompt di votazione.

Structured Chain of Thougt (SCoT)

" Una tecnica per la generazione di codice, basata sull'osservazione che gli sviluppatori umani seguono tipicamente, è la programmazione strutturata, che si basa su tre strutture fondamentali (sequenziale, a ramo e a ciclo). " [5]

Esempio:

Istruzioni in linguaggio naturale:

Il tuo compito è completare il seguente codice. Dovresti prima scrivere un processo approssimativo di problem-solving usando tre strutture di programmazione (cioè, sequenziale, branch e loop) e poi produrre il codice finale.

Ecco alcuni esempi dimostrativi:

```
def sum_Of_Primes(n):
    """Calcola la somma dei numeri primi da 1 a n."""

    # Ragioniamo passo passo
    # Input: n, un intero
    # Output: somma, un intero
    # 1. Inizializza una lista "primo" con valori True.
    # 2. Inizializza una variabile "p" a 2.
    # 3. Finché p * p è minore o uguale a n:
    # 4.    Se primo[p] è True:
    # 5.        Imposta tutti i multipli di p a False.
    # 6.    Incrementa la variabile "p" di 1.
    # 7. Calcola la somma dei numeri primi.
    # 8. Restituisci la somma.
    # Scrivi qui il tuo codice
    primo = [True] * (n + 1)
    p = 2
    while p * p <= n:
        pass # (altro codice qui)
```

Requisiti del test

Codice di input

```
def text_lowercase_underscore(testo):
    """Scrivi una funzione per trovare sequenze di lettere minuscole unite da un underscore."""

    # Ragioniamo passo passo
```

Capitolo 3

Metodologia di Ricerca

In questa ricerca è stata condotta un'analisi comparativa del comportamento di alcuni LLM nella generazione di codice di programmazione. Le valutazioni sono state effettuate su tre LLM differenti: GPT-4o, Claude 3.5 Sonnet V2 e Gemini 2.0 Flash. L'obiettivo principale è stato di verificare come questi modelli si comportano in termini di correttezza e qualità del codice, considerando alcuni esercizi di programmazione, diverse tecniche di prompting applicate e diverse istruzioni fornite come input ai modelli.

La scelta dei tre LLM è stata dettata dal loro differente approccio all'elaborazione del linguaggio, dalla loro popolarità e dal loro ampio utilizzo, nonchè dalla loro disponibilità, dal loro accesso, requisito fondamentale nella ricerca, e dalla loro rilevanza nella comunità degli sviluppatori.

GPT-4o è stato scelto per la sua popolarità nel mondo della generazione di codice. "È stato addestrato utilizzando l'apprendimento per rinforzo da feedback umano (RLHF): in particolare sono state utilizzate conversazioni tra istruttori umani che hanno simulato l'interazione tra l'utente e l'assistente artificiale. Un ulteriore progresso nelle risposte di GPT-4o è stato ottenuto dall'utilizzo di Proximal Policy Optimization, basato sul ranking di qualità fornito dagli istruttori, che potrebbe aver potenziato la capacità da parte di GPT-4o di generare del codice corretto e ben strutturato." **(No date) Introducing Chatgpt | openai. Available at: https://openai.com/index/chatgpt/ (Accessed: 07 May 2025).**

Claude 3.5 Sonnet V2 è stato selezionato per la sua specifica capacità di affrontare problemi di coding. Come riportato nel paper **Introducing Claude 3.5 Sonnet (no date) Introducing Claude 3.5 Sonnet Anthropic. Available at: https://www.anthropic.com/news/claude-3-5-sonnet (Accessed: 07 May 2025).** , "Claude 3.5 Sonnet raggiunge un tasso di successo del 64% su una valutazione interna di "agentic coding": il modello può scrivere ed eseguire codice in un ciclo agente, correggendosi iterativamente durante la valutazione. Questa capacità di interagire con il codice, di eseguire test e di autocorreggersi rende Claude 3.5 Sonnet particolarmente rilevante per la ricerca, che si concentra sull'analisi delle performance degli LLM nella generazione di codice funzionale e di qualità. Inoltre, ClaudeSonnet costruisce le sue risposte un carattere alla volta, in sequenza. Non ha possibilità di revisione e rettifica delle risposte una volta generate, a meno che gli utenti non lo richiedano tramite prompt successivo."

Gemini 2.0 Flash è stato incluso in questa ricerca per la sua innovativa funzionalità di esecuzione del codice Python. "A differenza di altri LLM, Gemini non si limita a generare codice, ma può anche eseguirlo in un ambiente sandboxed e apprendere iterativamente dai risultati." Questa capacità, descritta nella documentazione ufficiale **Code execution | Generative AI on Vertex AI | google cloud (no date) Google. Available at: https://cloud.google.com/vertex-ai/generative-ai/docs/multimodal/code-execution (Accessed: 07 May 2025).** , permette a Gemini di affinare le proprie risposte e di produrre output più accurati e funzionali. In particolare, la possibilità di apprendere dall'esecuzione del codice si allinea perfettamente con l'obiettivo della ricerca di valutare l'efficacia degli LLM nella generazione di codice corretto e di qualità.

Di seguito, vengono descritti i dettagli metodologici della valutazione preliminiare e della valutazione conclusiva

3.1 Valutazione preliminare

È stato selezionato un set di esercizi mediamente difficili e, per alcuni di essi, sono state applicate diverse tecniche per testare la generazione del codice dal punto di vista della correttezza e poi della qualità. Ogni tecnica è stata poi confrontata al fine di individuare la più efficace. Non a tutti gli esercizi sono state applicate tutte le tecniche di test in quanto non rivelanti ai fini della valutazione.

3.1.1 Obiettivi della Valutazione preliminare

- **Individuare la tecnica di prompt più efficace per generare codice corretto:** ogni efficacia delle tecniche viene valutata tramite il numero di test superati - le tecniche che superano tutti i test sono le più efficaci.
- **Individuare la tecnica di prompt più efficace per generare codice di qualità:** l'efficacia delle tecniche viene valutata attraverso le metriche di RustCode Analysis e confrontate tra di loro per verificare quale tecnica ha ottenuto i punteggi migliori.
- **Progettare un prompt** da utilizzare per la seconda valutazione in modo da ottenere dei risultati ottimali su un set di esercizi di larga scala.

3.1.2 Selezione degli esercizi

Sono stati selezionati sei esercizi di complessità variabile, 4 in Python e 2 in RUST. Gli esercizi sono riportati di seguito:

1. **Calcolare il resto con il minor numero possibile di monete**
    - **Motivazione:** Questo primo esercizio in Python è stato assunto per il bisogno di risoluzione logica relativo alla generazione del codice, ovvero come si comporta LLM in un contesto dove la parte di programmazione non è quella centrale richiesta dall'esercizio, poichè si mirava di verificare che i risultati delle operazioni (matematiche) fossero corretti.
    - Descrizione: Determinare il minor numero di monete da dare a un cliente in modo che la somma dei loro valori corrisponda all'importo corretto del resto.

2. **Algoritmo di ricerca binaria**
    - **Motivazione:** In questo caso è stato scelto un esercizio "teorico" per verificare come gli LLM si comportano su un esercizio per il quale sono stati allenati. Questo ci può aiutare a far capire come anche il contesto iniziale dell'LLM può impattare i prompt inseriti all'interno dei modelli.
    - **Descrizione:** Implementare un algoritmo di ricerca binaria. Un algoritmo di ricerca binaria trova un elemento in una lista dividendo ripetutamente la lista a metà, mantenendo solo la metà che contiene l'elemento che stiamo cercando. Questo ci permette di restringere rapidamente le possibili posizioni del nostro elemento fino a trovarlo, o fino a eliminare tutte le possibili posizioni.
3. **Rest API**
    - **Motivazione:** Per questo esercizio si è proposto un problema che può essere più comune in programmazione, ovvero la generazione di uno "scheletro" REST API in Python. Questo tipo di esercizio pur essendo frequente presenta una complessità maggiore rispetto ad altri. Pertanto, si suppone che gl LLM debbano comportarsi in maniera soddisfacente a livello di correttezza, ma con alcune difficoltà legate alla natura più articolata del problema.
    - **Descrizione:** implementare un'API RESTful per tracciare i debiti (IOU). Quattro coinquilini hanno l'abitudine di prestarsi denaro frequentemente e hanno difficoltà a ricordare chi deve a chi, e quanto. Implementare una semplice API RESTful che riceva i debiti come richieste POST e possa fornire informazioni di riepilogo specificate tramite richieste GET.
4. **Account bancario**
    - **Motivazione:** Per questo esercizio si è cercato un problema ordinario così da valutare se gli LLM possono eccellere senza necessità di tecniche avanzate di prompting, confermando la loro efficacia su compiti standardizzati e ben definiti.
    - **Descrizione:** Implementare conti bancari che supportino l'apertura/chiusura, i prelievi e i depositi di denaro. Poiché i conti bancari possono essere accessibili in molti modi diversi (internet, telefoni cellulari, addebiti automatici), il tuo software bancario deve consentire che i conti siano accessibili in sicurezza
5. **Somma dei numeri in un Item in parallelo (RUST)**
    - **Motivazione:** Questo esercizio viene richiesto in Linguaggio RUST, per verificare se cambiando linguaggio di programmazione e scegliendone uno più complesso, si ottengono comunque dei risultati simili. Anche per quanto riguarda la letterattura disponibile online, Python è un linguaggio ampiamente esplorato e diffuso, mentre RUST, essendo un linguaggio recente e meno conosciuto, ha sicuramente minor materiale e documentazione online.
    - **Descrizione:** Implementa la seguente funzione, che dovrebbe sommare tutti i numeri nella slice ‘items‘ in parallelo, utilizzando ‘threads‘ thread. Assicurandosi che l'implementazione sia effettivamente parallela e che sia più veloce per input di grandi dimensioni rispetto all'esecuzione su un singolo thread.

6. **Frequenza lettere in un file - Calcolo parallelo (RUST)**

   - **Motivazione:** Anche qui abbiamo richiesto ai vari LLM di risolvere un esercizio di RUST, aggiungendo la complessità del parallelismo dei thread, per verificare effettivamente come si sarebbero comportati sulla correttezza del codice.
   - **Descrizione:** Contare la frequenza delle lettere nei testi utilizzando il calcolo parallelo, ovvero l'operazione centrale del programma consiste nell' eseguire il conteggio in parallelo, invece che farlo in modo sequenziale. Un esempio comune è contare la frequenza delle lettere utilizzando il parallelismo per calcolare la frequenza totale di ciascuna lettera in una lista di testo.

3.1.3 Tecniche utilizzate per la correttezza del codice

Prompt base

Questo è il prompt meno complesso che consiste nel prendere semplicemente il testo dell'esercizio e la struttura dell'esercizio e inserirli come input all'LLM. È stato usato questo prompt per avere una base di risultato/performance dell'LLM per quanto riguarda l'esercizio senza adottare alcuna tecnica: al fine di comprendere come un LLM da solo potesse generare un codice corretto, in base all'esercizio dato.

RGC personalizzato

La tecnica dell'RGC, consiste nel dare un Ruolo, un Risultato, un Obiettivo, un Contesto e un Vincolo all'LLM. Si è voluto verificare in tal modo fosse possibile produrre risultati migliori, senza però fornire ausili effettivi nella generazione del codice, tranne per alcuni casi specificati nel campo "Vincolo" con lo scopo di definire certe limitazioni del programma.

Per esempio, per gli esercizi in RUST, uno dei vincoli constatati:

```
Vincolo: Ricordati le regole del Borrow Checker e sui lifetime
delle reference che potresti andare a generare
```

Chain of Thought prompting

In questa tecnica si sono dati più input all'LLM, in base ai risultati ottenuti dagli output precedenti: come primo prompt non è stata usata alcuna tecnica, ma dopo ogni generazione del codice si sono stati forniti consigli per migliorare la correttezza del codice generato in base agli errori che i test fornivano, oppure, se presenti, il testo degli errori di compilazione.

Structured Chain of Thought con esempi

Si è cercato di far generare un processo approssimativo di problem-solving all'LLM prima di produrre il codice finale, inoltre sono stati aggiunti tutti gli esempi e le criticità riscontrati nelle tecniche precedenti. Per questa tecnica si è adottato un approccio il più completo e dettagliato possibile, per capire se effettivamente fornendo un ampio set di informazioni e di processi sul prompt so possa produrre un codice più corretto.

In alcune circostanze a questa tecnica è stato aggiunto un ulteriore prompt per permettere all'LLM di migliorare il proprio codice, poichè si verificavano errori non rilevanti per i quali è possibile farli correggere all'LLM (per esempio, in RUST, erano presenti degli errori di compilazione che ogni tecnica ha dato e che si potevano superare solo dal secondo prompt, anche dati i consigli iniziali).

Fornire all'LLM il controllo dei test

Questa tecnica è stata usata per effettuare una comparazione tra le tecniche in termini di capacità di successo nei test. Questo approccio rappresenta un benchmark per misurare l'efficacia delle altre tecniche, poiché si parte dall'ipotesi che l'LLM, con accesso a tutte le informazioni necessarie, possa produrre risultati ottimali.

3.1.4 Tecniche utilizzate per la qualità del codice

Prompt Ottimizzazione Qualitativa

Questa tecnica è stata impiegata per un miglioramento qualitativo attraverso:

- Selezione del codice con il miglior punteggio qualitativo tra quelli generati dai vari LLM
- Utilizzo di questo codice come scheletro base per tutti e tre gli LLM
- Implementazione di prompt specifici per il miglioramento qualitativo
- Nuova analisi dei codici generati per verificare l'effettivo miglioramento qualitativo, verificandone anche la correttezza ri-eseguendo tutti i test dell'esperimento precedente

Ad esempio, se il codice di Claude Sonnet risultava qualitativamente superiore, questo veniva utilizzato come base per tutti e tre gli LLM, richiedendo loro specificamente di ottimizzarlo dal punto di vista qualitativo.

Il prompt utilizzato per il miglioramento qualitativo è riportato di seguito ed è stato oggetto di analisi successive grazie ai risultati favorevoli nel corso della ricerca in quanto ha ottenuto degli ottimi risultati, inoltre per le istruzioni che davano delle regole precise riguardanti le nostre metriche ci siamo basati anche su fonti come [4].:

```
Ti verrà fornito un programma in [linguaggio di programmazione].
Non dovrai fare modifiche per quanto riguarda la correttezza del codice,
dovrai modificare il codice solo per quanto riguarda la sua qualità.
Qui di seguito delle linee guida per il risultato aspettato:

- NON modificare il nome delle funzioni, per rispettare l'esecuzione dei test,
    puoi aggiungerne, ma le funzioni principali devono chiamarsi allo stesso modo.
- Utilizza strutture di controllo semplici e riduci al minimo la complessità.
- Evita annidamenti profondi e suddividi il codice in funzioni più piccole se necessario.
- Usa nomi di variabili descrittivi, evita logica complessa e documenta
    ogni passaggio con commenti chiari.
- Evita condizioni annidate e cicli intricati.
- Usa operatori e costrutti di linguaggio chiari e intuitivi.
- Evita codice duplicato e usa costrutti semplici e leggibili.
- Modifica il codice in modo che sia facile da implementare e comprendere,
    riducendo lo sforzo richiesto per la lettura e la modifica.
- Assicurati che il codice sia leggibile, ben documentato e utilizzi
    buone pratiche di programmazione.
- Evita operazioni ridondanti e usa strutture dati appropriate.
- Correggi in modo scalabile, ottimizzando l'uso delle risorse.
- Evita effetti collaterali e separa la logica dal codice di input/output.
- Mantieni i messaggi di errore senza tradurli o cambiarli
```

Il prompt è stato progettato per guidare l'LLM nel miglioramento della qualità senza alterare la sua correttezza o modificare elementi essenziali per l'esecuzione dei test. Le linee guida sono state fornite per garantire che il codice rispetti gli standard delle metriche che abbiamo usato; queste linee guida hanno coperto diversi aspetti:

- Struttura del codice -> Evita annidamenti profondi e suddividi il codice in funzioni più piccole se necessario.
- Nomi descrittivi
- Documentazione -> Assicurati che il codice sia leggibile, ben documentato e utilizzi buone pratiche di programmazione.
- Efficienza e leggibilità -> Evita codice duplicato e usa costrutti semplici e leggibili.
- Scalabilità e manutenzione -> Correggi in modo scalabile, ottimizzando l'uso delle risorse.

Il focus principale del prompt specificava chiaramente l'obiettivo di migliorare il codice dal punto di vista della qualità.

3.1.5 Tecniche utilizzate per la correttezza e la qualità del codice

Progettazione di un prompt per l'ottimizzazione congiunta di correttezza e qualità

- **Obiettivo:** Confrontare l'efficacia di un approccio sequenziale (prima correttezza, poi qualità) con un approccio integrato (richiesta simultanea di correttezza e qualità)
- **Metodologia:**
  - Si è deciso di integrare tutte le tecniche e le ottimizzazioni precedentemente identificate in un unico prompt, utilizzato fin dall'inizio, per valutare quale approccio risulti più efficace
  - Identificare se sia meglio:
    - Trovare un codice corretto per poi migliorarlo qualitativamente
    - Richiedere immediatamente all'LLM di generare un codice che corretto e di qualità
- **Processo di confronto:**
  - Verranno selezionate 2-3 delle migliori ottimizzazioni qualitative ottenute nei precedenti esperimenti
  - Comparate con nuove generazioni di codice ottenute utilizzando un prompt che unisca:
    - I migliori esempi identificati nella valutazione della correttezza del codice
    - Le migliori linee guida emerse nella valutazione della qualità del codice
  - Il prompt viene affinato in base all'esercizio da risolvere
  - Verificare se sia meglio che le linee guida sulla qualità del codice rimangano le stesse di quelle definite nella valutazione precedente

Esplorazione capacità dell'LLM di generare prompt efficaci per un altro LLM

- **Obiettivo:** Valutare se un LLM può essere utilizzato per generare prompt efficaci da fornire a un altro LLM, al fine di ottenere codice corretto e di alta qualità

- **Approccio:**
  - Con questa tecnica si è indagato se sia più efficace utilizzare un LLM per:
    * Risolvere l'esercizio
    * Generare prompt ottimizzati da fornire a un altro LLM
  - L'obiettivo finale è ottenere codice ottimale dal punto di vista di qualità e correttezza

3.1.6 Metriche di valutazione

Le metriche di valutazione si dividono per:

- **Correttezza del codice:** questa si è calcolata tramite il numero di test superati - il massimo punteggio è il superamento di tutti i test dell'esercizio
- **Qualità del codice generato:** questa metrica è stata calcolata tramite RustCode Analysis ed è riportata nel capitolo Background. Un punteggio migliore su queste metriche indicava una miglior qualità del codice.

3.2 Valutazione su larga scala dell'efficacia degli LLM

3.2.1 Obiettivo

Valutare in maniera sistematica la correttezza e la qualità del codice generato su un dataset più ampio e rappresentativo.

In questa valutazione si è preso il prompt congiunto che ha mostrato i migliori risultati per la correttezza e la qualità del codice e lo si è applicato a GPT-4o per verificare la sua efficacia su un set di esercizi di media difficoltà.

3.2.2 Tecnica utilizzata

Il test ha coinvolto quindi 150 esercizi di media complessità, utilizzando un unico prompt per cercare di ottenere il miglior risultato a livello di correttezza e di qualità, senza ricorrere a prompt successivi per migliorare il risultato.

L'intento principale è stato quello di raccogliere le tecniche più efficienti identificate nelle fasi precedenti e verificarne la scalabilità su un campione più importante, valutando se potevano portare a dei risultati pratici in ambito di sviluppo software.

Inoltre abbiamo utilizzato molto le indicazioni presenti nel documento [5], per esempio per l'istruzione "Dovresti prima scrivere un processo approssimativo di problem-solving usando tre strutture di programmazione (cioè, sequenziale, branch e loop) e poi produrre il codice finale."

Come prompt useremo il seguente modello:

```
Ti verrà richiesto di risolvere vari esercizi in Python in questa chat.
Dovrai seguire le istruzioni e generarmi un codice corretto che passi tutti i test.

Per ogni prompt ti verranno date le istruzioni dell'esercizio e la struttura della funzione.
Dovresti prima scrivere un processo approssimativo di problem-solving usando tre strutture
di programmazione (cioè, sequenziale, branch e loop) e poi produrre il codice finale.
```

```
Qui di seguito delle linee guida del risultato aspettato:

- NON modificare il nome delle funzioni, per rispettare l'esecuzione dei test, puoi aggiungerne,
ma le funzioni principali devono chiamarsi allo stesso modo.
- Utilizza strutture di controllo semplici e riduci al minimo la complessità.
- Evita annidamenti profondi e suddividi il codice in funzioni più piccole se necessario.
- Usa nomi di variabili descrittivi, evita logica complessa e
documenta ogni passaggio con commenti chiari.
- Evita condizioni annidate e cicli intricati.
- Usa operatori e costrutti di linguaggio chiari e intuitivi.
- Evita codice duplicato e usa costrutti semplici e leggibili.
- Modifica il codice in modo che sia facile da implementare e
comprendere, riducendo lo sforzo richiesto per la lettura e la modifica.
- Assicurati che il codice sia leggibile, ben documentato e utilizzi
buone pratiche di programmazione.
- Evita operazioni ridondanti e usa strutture dati appropriate.
- Correggi in modo scalabile, ottimizzando l'uso delle risorse.
- Evita effetti collaterali e separa la logica dal codice
di input/output.
- Mantieni i messaggi di errore senza tradurli o cambiarli
```

Il prompt segue molto le linee guida che abbiamo "costruito" nella valutazione di un "Prompt per l'ottimizzazione congiunta di correttezza e qualità" anche se in questo caso, il prompt doveva per forza essere più generico in quanto non era possibile affinarlo per ognuno dei 150 esercizi.

Le metriche di valutazione si dividono per:

- **Correttezza del codice:** questa si è calcolata tramite il numero di test superati - il massimo punteggio è il superamento di tutti i test di tutti gli esercizi.
- **Qualità del codice generato:** questa si è calcolata tramite RustCode Analysis ed è riportata nel capitolo Background. Un punteggio migliore su queste metriche indicava una miglior qualità del codice.

3.3 Considerazioni Finali

La metodolgia utilizzata all'interno di questo studio consente di studiare e verificare in modo sistematico le capacità dei diversi LLM nella generazione di un codice corretto e di qualità, analizzando con particolare attenzione l'impatto delle diverse tecniche di prompting utilizzate. I risultati ottenuti forniranno indicazioni utili con l'obiettivo di ottimizzare e rendere più efficace l'utilizzo degli LLM nei contesti di programmazione e di progettazione software.

Capitolo 4

Valutazione preliminare dell'efficacia degli LLM

Il presente capitolo si propone di analizzare le prestazioni dei modelli scelti attraverso una serie di test mirati nella generazione di codice attraverso diverse tecniche di prompting prendendo sei esercizi di programmazione, al fine di identificare i punti di forza e le criticità.

L'obiettivo principale è quello di identificare e confrontare le strategie più efficaci per ottenere codice sia corretto che di elevata qualità, al fine di stabilire una metodologia ottimale per le valutazioni successive su larga scala.

Attraverso questa valutazione che combina test di correttezza funzionale e analisi qualitative del codice prodotto, si mira a:

- **Identificare la tecnica di prompting più efficace per la generazione di codice funzionalmente corretto**
- **Determinare quale approccio produce codice di qualità superiore secondo metriche standardizzate**
- **Sviluppare una strategia di prompting ottimizzata per applicazioni su set di esercizi di ampia portata**

4.1 Esercizio 1: Calcolare il resto con il minor numero possibile di monete

Questo esercizio richiede l'implementazione di un algoritmo per determinare il minor numero di monete necessarie per un resto specifico.

L'esercizio presenta sfide principalmente di natura algoritmica e logico-matematica. La complessità non risiede tanto nella sintassi del linguaggio quanto nella corretta gestione dei casi limite (valori negativi, importi non componibili) e nell'ottimizzazione.

- Problemi che richiedono ragionamento matematico complesso
- Gestione di casi limite e validazione input
- Implementazione di algoritmi di ottimizzazione

4.1.1 GEMINI 2.0

Analisi correttezza del codice

- **Prima chat: nessuna tecnica adottata**

  La prima generazione ha passato 5 test e ne ha falliti 8. Anche se uno dei test falliti era sulla generazione dell'errore che non seguiva le istruzioni dell'esercizio.

- **Seconda chat: RGC**

  La prima generazione ha passato 6 test e ne ha falliti 7. Semplicemente ha passato in più il test che verifica la stringa di errore sul valore delle monete che non può essere negativo.

- **Terza chat: Chain of Thought prompting**

  Come per la simulazione senza nessuna tecnica la prima generazione ha fallito dei test: anche se uno dei test era sulla generazione dell'errore che non seguiva le istruzioni dell'esercizio. Per il secondo e per il terzo prompt sono stati aggiunti degli input che spiegassero i test falliti all'LLM e per il quarto prompt sono stati forniti esempi simili a quelli usati nei test, cambiando gli importi

  - La prima generazione ha passato 5 test e ne ha falliti 8.
  - La seconda generazione ha passato 6 test e ne ha falliti 7.
  - La terza generazione ha passato 8 test e ne ha falliti 5.
  - Con la quarta generazione, Gemini è riuscito a passare tutti i test

- **Quarta chat: Structured Chain of Thought prompting con gli esempi e i punti di attenzioni di prima**

  La generazione del programma ha passato 6 test e ne ha falliti 7.

- **Quinta chat: RGC rivisitato + dare i test all'LLM**

  Soprendentemente, LLM ha passato solo 8 test, fallendone 5.

  In seguito si è fornito il testo degli errori e dei test falliti - dopo questo ultimo prompt, Gemini è riuscito a passare tutti i test

Analisi qualitativa del codice

L'unica tecnica che ha passato tutti i test è stata quella del CoT, quindi solo su questa sono state analizzate le metriche di qualità

Metriche qualità - Chain of Thought prompting

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	11	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Difficoltà di Halstead	34.675	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	4.345	Log10 > 4	Leggermente superiore alla soglia minima, sforzo moderato.
MI (Indice di Manutenibilità)	75.524 (original)	MI $\geq$ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	11	Non specificata	Complessità elevata per una singola classe, ma manca una media di riferimento.
Complessità Cognitiva	11	Non specificata	Complessità cognitiva moderata, ma manca una media di riferimento.

Metriche qualità - Prompt Ottimizzazione Qualitativa

In questo test è stato adottato lo scheletro del codice di ClaudeSonnet e lo si è dato a Gemini insieme al prompt per ottimizzarne la qualità. Le metriche ottenute sono le seguenti:

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	10 (funzione `find_fewest_coins`)	1-2	Significativamente più alto della media, complessità molto elevata.
Complessità Cognitiva	14 (funzione `find_fewest_coins`)	Non specificata	Complessità elevata, ma manca una media di riferimento.
Difficoltà di Halstead	26.71 (funzione `find_fewest_coins`)	10-100	Rientra nella media, ma verso il limite superiore, indicando una difficoltà elevata.
Sforzo di Halstead (Log10)	4.18 (funzione `find_fewest_coins`)	Log10 > 4	Superiore alla soglia minima, sforzo elevato.
MI (Indice di Manutenibilità)	75.62 (funzione `find_fewest_coins`)	MI $\geq$ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	10 (funzione `find_fewest_coins`)	Non specificata	Complessità molto elevata, ma manca una media di riferimento.

4.1.2 GPT

Analisi correttezza del codice

- **Prima chat: nessuna tecnica adottata**

  La prima generazione ha passato 5 test e ne ha falliti 8.

- **Seconda chat: RGC**

  La prima generazione ha passato 6 test e ne ha falliti 7.

- **Terza chat: Chain of Thought prompting**

  – Come per la simulazione priva di tecnica specifica la prima generazione del programma ha passato 5 test e ne ha falliti 8.

  – La seconda generazione ha passato 8 test e ne ha falliti 5.

  – La terza generazione ha passato tutti i test. Un prompt in meno rispetto a Gemini; non sono stati necessari gli esempi con i ragionamenti simili ai test.

- **Quarta chat: Structured Chain of Thought prompting con gli esempi e i punti di attenzioni di prima**

  La generazione del programma ha passato 6 test e ne ha falliti 7.

- **Quinta chat: RGC rivisitato + dare i test all'LLM**

  Contrariamente a quanto atteso, LLM ha passato solo 8 test, fallendone 5.

  Successivamente si è provato a fornire il testo degli errori e dei test falliti - dopo questo ultimo prompt, GPT è riuscito a passare tutti i test

Analisi qualitativa del codice

Metriche qualità Chain of Thought prompting

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	11	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Difficoltà di Halstead	30	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	4.188	Log10 > 4	Leggermente superiore alla soglia minima, sforzo moderato.
MI (Indice di Manutenibilità)	80.402 (original)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	11	Non specificata	Complessità elevata per una singola classe, ma manca una media di riferimento.
Complessità Cognitiva	18	Non specificata	Complessità cognitiva elevata, ma manca una media di riferimento.

Prompt Ottimizzazione Qualitativa

In questo caso si è preso lo scheletro del codice di ClaudeSonnet e lo si è dato a Gemini insieme al prompt per ottimizzarne la qualità. Le metriche ottenute sono le seguenti:

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	6 (funzione `compute_dp_solution`)	1-2	Significativamente più alto della media, complessità molto elevata.
Complessità Cognitiva	11 (funzione `compute_dp_solution`)	Non specificata	Complessità elevata, ma manca una media di riferimento.
Difficoltà di Halstead	18 (funzione `compute_dp_solution`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	3.69 (funzione `compute_dp_solution`)	Log10 > 4	Appena sotto la soglia minima, sforzo moderato.
MI (Indice di Manutenibilità)	93.67 (funzione `compute_dp_solution`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	6 (funzione `compute_dp_solution`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

4.1.3 ClaudeSonnet

Analisi correttezza del codice

- **Prima chat: nessuna tecnica adottata**

  La prima generazione ha passato tutti i test.

- **Seconda chat: RGC**

  La prima generazione ha passato tutti i test.

- **Terza chat: Chain of Thought prompting**

  Questa chat non è stato necessario testarla, in quanto il programma ha avuto successo a tutti i test precedenti.

- **Quarta chat: Structured Chain of Thought prompting con gli esempi e i punti di attenzioni di prima**

  Nonostante le premesse, il programma generato ha fallito 1 test, ma essendo che era un problema non rilevante, è stato fornito un altro prompt. A seguito di ciò il modello ha passato tutti i test.

- **Quinta chat: RGC rivisitato + dare i test all'LLM**

  Anche in questo caso, il programma generato da Claude ha passato tutti i test.

Analisi qualitativa del codice

Metriche qualità Chain of Thought prompting

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	11	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Difficoltà di Halstead	30	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	4.188	Log10 > 4	Leggermente superiore alla soglia minima, sforzo moderato.
MI (Indice di Manutenibilità)	80.402 (original)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	11	Non specificata	Complessità elevata per una singola classe, ma manca una media di riferimento.
Complessità Cognitiva	18	Non specificata	Complessità cognitiva elevata, ma manca una media di riferimento.

Prompt Ottimizzazione Qualitativa

Le metriche ottenute dopo il prompt per migliorare la qualità sono le seguenti:

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	6 (funzione `compute_dp_solution`)	1-2	Significativamente più alto della media, complessità molto elevata.
Complessità Cognitiva	11 (funzione `compute_dp_solution`)	Non specificata	Complessità elevata, ma manca una media di riferimento.
Difficoltà di Halstead	18 (funzione `compute_dp_solution`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	3.69 (funzione `compute_dp_solution`)	Log10 > 4	Appena sotto la soglia minima, sforzo moderato.
MI (Indice di Manutenibilità)	93.67 (funzione `compute_dp_solution`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	6 (funzione `compute_dp_solution`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

4.1.4 Conclusioni sulla correttezza primo esercizio

Per Gemini il prompt che ha avuto più successo è stato quello del Chain of Thought - dettato dal fatto che i consigli forniti all'LLM riguardavano i test e il contesto era già consolidato al quarto prompt. Tuttavia, immettendo gli stessi esempi e gli stessi consigli già ad inizio prompt questo non ha aiutato particolarmente LLM a generare un codice più corretto.

Dare all'LLM il corpo dei test e anche il risultato dei test falliti ha permesso di generare il codice corretto secondo i nostri criteri di valutazione. Si suppone che l'esercizio, essendo comunque logicamente complesso, abbia messo molto in difficoltà Gemini.

Anche per GPT, il prompt che ha avuto più successo è risultato essere il Chain of Thought - probabilmente perchè i consigli forniti all'LLM riguardavano i test e il contesto era già consolidato al quarto prompt. Tuttavia, immettere gli stessi esempi e gli stessi consigli già a inizio prompt non ha aiutato particolarmente LLM a generare un codice più corretto. Dare all'LLM il contenuto dei test e anche il risultato dei test falliti ha permesso di generare il codice corretto secondo i nostri criteri di valutazione.

ClaudeSonnet invece si è contraddistinto per la sua capacità, poichè è riuscito a generare fin da subito un codice che passasse tutti i test senza l'utilizzo di tecniche avanzate o di ulteriori suggerimenti, superando gli altri due LLM in termini di correttezza.

Gli LLM hanno generato risultati migliori quando sono stati guidati da una sequenza di prompt. I prompt più efficaci sono quelli che all'interno contengono degli esempi dedicati alle nostre istruzioni. La tecnica CoT si è dimostrata particolarmente utile per Gemini e GPT. Fa eccezione ClaudeSonnet, che è riuscito a generare il codice senza nessun ulteriore aiuto. ClaudeSonnet è stato l'LLM che ha ottenuto dal principio un punteggio pieno sulla correttezza del codice.

La natura dell'esercizio, che combina programmazione e logica matematica, potrebbe aver influenzato i risultati, favorendo modelli più robusti nella gestione di problemi logico-matematici, come ClaudeSonnet.

4.1.5 Conclusioni sulla qualità primo esercizio

I risultati delle metriche si assomigliano molto per i 3 LLM. In questa analisi emerge Claude che ottiene risultati migliori, con un valore inferiore nella Difficoltà di Halstead.

Per quanto riguarda i risultati relativi alla tecnica di ottimizzazione Qualitativa sono riportati di seguito:

- Gemini ha migliorato il punteggio rispetto al codice che egli stesso aveva generato, ma rispetto allo scheletro di Claude non sono rilevate differenze sostanziali nelle metriche
- GPT ha ottenuto un ottimo risultato, riducendo la complessità e aumentando di molto la manutenibilità
- Claude ha migliorato il punteggio rispetto al codice che aveva generato in precedenza, ottenendo risultati ottimali, ad eccezione della manutenibilità, in cui GPT risulta essere superiore. Per il resto Claude ha generato il codice migliore da un punto di vista qualitativo.

4.1.6 Sintesi dell'esercizio 1

Questo esercizio ha evidenziato l'importanza del ragionamento algoritmico nella generazione di codice e ha mostrato significative differenze tra i modelli testati. Claude Sonnet emerge come il più competente su problemi logico-matematici, mentre Gemini e GPT beneficiano significativamente di tecniche di prompt engineering iterative.

4.2 Esercizio 2: Algoritmo di ricerca binaria

Questo esercizio richiede l'implementazione di un algoritmo di ricerca binaria per trovare un elemento in una lista ordinata.

La ricerca binaria rappresenta un algoritmo fondamentale dell'informatica, ampiamente documentato nella letteratura accademica e nei materiali didattici. Questo esercizio è stato selezionato specificamente per valutare come gli LLM si comportino con problemi "teorici da libro", dove la soluzione è standardizzata e ben consolidata.

Obiettivi di valutiazione:

- Necessità di implementazione di tecniche avanzate di prompting per l'implemetnazione di un algoritmo ben documentato
- Influenza dei dati di addestramento sulla prestazione dell'LLM
- Qualità del codice su un problema standard di programmazione

4.2.1 Analisi correttezza del codice

Tutti gli LLM considerati hanno superato con successo tutti i test senza il bisogno di applicare alcuna tecnica specifica.

4.2.2 Analisi qualitativa del codice

Di seguito i risultati qualitativi per ogni LLM:

Gemini - Nessuna tecnica adottata

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	5 (funzione `find`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Complessità Cognitiva	5 (funzione `find`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	18.75 (funzione `find`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	3.66 (funzione `find`)	Log10 > 4	Appena sotto la soglia minima, sforzo moderato.
MI (Indice di Manutenibilità)	87.32 (funzione `find`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	5 (funzione `find`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

GPT - Nessuna tecnica adottata

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	6 (funzione `find`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Complessità Cognitiva	6 (funzione `find`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	22 (funzione `find`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	3.78 (funzione `find`)	Log10 > 4	Appena sotto la soglia minima, sforzo moderato.
MI (Indice di Manutenibilità)	88.29 (funzione `find`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	6 (funzione `find`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

ClaudeSonnet - Nessuna tecnica adottata

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	6 (funzione `find`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Complessità Cognitiva	6 (funzione `find`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	22 (funzione `find`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	3.78 (funzione `find`)	Log10 > 4	Appena sotto la soglia minima, sforzo moderato.
MI (Indice di Manutenibilità)	87.65 (funzione `find`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	6 (funzione `find`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

4.2.3 Prompt Ottimizzazione Qualitativa

In questo test è stato assunto lo scheletro del codice di Gemini in quanto questo modello ha ottenuto le metriche migliori.

I risultati ottenuti sono i seguenti:

Gemini

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	5 (funzione `binary_search`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Complessità Cognitiva	5 (funzione `binary_search`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	18.75 (funzione `binary_search`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	3.66 (funzione `binary_search`)	Log10 > 4	Appena sotto la soglia minima, sforzo moderato.
MI (Indice di Manutenibilità)	87.90 (funzione `binary_search`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	5 (funzione `binary_search`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

GPT

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	5 (funzione `binary_search`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Complessità Cognitiva	5 (funzione `binary_search`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	13.2 (funzione `binary_search`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	3.53 (funzione `binary_search`)	Log10 > 4	Appena sotto la soglia minima, sforzo moderato.
MI (Indice di Manutenibilità)	85.90 (funzione `binary_search`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	5 (funzione `binary_search`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

Claude

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	5 (funzione `find_value_binary_search`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Complessità Cognitiva	6 (funzione `find_value_binary_search`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	16.5 (funzione `find_value_binary_search`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	3.66 (funzione `find_value_binary_search`)	Log10 > 4	Appena sotto la soglia minima, sforzo moderato.
MI (Indice di Manutenibilità)	82.59 (funzione `find_value_binary_search`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	5 (funzione `find_value_binary_search`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

4.2.4 Conclusioni sulla correttezza secondo esercizio

Dai risultati ottenuti si evince che:

- Gli LLM tendono ad eccellere su esercizi ben documentati e ampiamente trattati nella letteratura accademica, probabilmente grazie al loro addestramento su dataset che includono temi simili.
- Il contesto iniziale nel quale opera il modello, ovvero la conoscenza su un argomento teorico famoso e consolidato, condiziona in maniera importante la capacità del modello nella generazione di una soluzione corretta e conforme alle istruzioni senza la necessità di inserire ulteriori specifiche.

4.2.5 Conclusioni sulla qualità secondo esercizio

Le metriche di qualità ottenute dai 3 LLM risultano molto simili, tuttavia Gemini ottiene un risultato migliore significativo.

Il prompt per l'ottimizzazione della qualità non apporta miglioramenti rilevanti, essendo un esercizio basilare, non è stato possibile ottenere delle metriche superiori:

- Gemini non apporta nessun miglioramento rispetto allo scheletro inizialmente introdotto, stessi punteggi sulle metriche
- GPT riduce solamente di 5 punti la difficoltà di Halstead, il resto risulta invariato. Tuttavia la struttura del programma è cambiata notevolmente.
- Claude non apporta miglioramenti, ma ottiene risultati analoghi allo scheletro del programma.

4.2.6 Sintesi dell'esercizio 2

L'esercizio della ricerca binaria ha dimostrato l'elevata competenza di tutti gli LLM su problemi algoritmici standardizzati, evidenziando come la conoscenza preesistente nel training data possa rendere superflue tecniche di prompt engineering avanzate per questa categoria di problemi.

4.3 Esercizio 3: Rest Api

Questo esercizio richiede l'implementazione di un'API RESTful per tracciare i debiti (IOU) tra coinquilini. Il sistema deve gestire richieste POST per registrare nuovi debiti e richieste GET per fornire informazioni di riepilogo sui saldi.

L'esercizio presenta una complessità maggiore rispetto ai precedenti, combinando: - Progettazione di architettura software (API RESTful) - Gestione di strutture dati complesse - Gestione di casi limite (saldi negativi, etc) - Validazione di input e gestione errori

Questo problema è stato selezionato per valutare:

- Capacità degli LLM di gestire problemi di programmazione più articolati
- Gestione della complessità logica in scenari realistici
- Necessità di tecniche di prompt engineering per problemi complessi

4.3.1 Gemini 2.0

Analisi correttezza del codice

- **Prima chat: nessuna tecnica adottata**

  La prima generazione ha fallito tutti i test.

- **Seconda chat: RGC**

  La prima generazione ha fallito tutti i test, tranne uno.

- **Terza chat: Chain of Thought Prompting**

  La prima generazione ha fallito tutti i test.

  Il secondo prompt conteneva gli errori principali dei test falliti: la seconda generazione ha fallito tutti i test, tranne uno. Il terzo prompt conteneva le correzioni specifiche da apportare: la terza generazione, anche in questo caso, ha fallito tutti i test, tranne uno. Il quarto prompt conteneva il codice e l'output dei test falliti: la quarta generazione, anche qui, ha fallito tutti i test, tranne uno.

- **Quarta chat: Structured Chain of Thought prompting con gli esempi e i punti di attenzioni di prima**

  La generazione del programma ha passato 1 test su 9.

- **RGC rivisitato + dare i test all'LLM**

  La generazione ha passato 6 test e falliti 3.

Analisi qualitativa del codice

Si ricorda che in questo test il codice generato da Gemini non è stato analizzato, in quanto non è stato possibile generare un codice che passasse tutti i test di correttezza.

Prompt Ottimizzazione Qualitativa

Anche in questo caso Gemini non è stato capace di apportare dei miglioramenti significativi, per cui i risultati non sono stati riportati.

4.3.2 GPT

Analisi correttezza del codice

- **Prima chat: nessuna tecnica adottata**

  La prima generazione ha passato 6 test e falliti 3.

- **Seconda chat: RGC**

  La prima generazione ha passato 6 test e falliti 3.

- **Terza chat: Chain of Thought Prompting**

  Contrariamente alle aspettative la prima generazione ha passato tutti i test senza dover applicare ulteriori tecniche.

- **Quarta chat: Structured Chain of Thought prompting con gli esempi e i punti di attenzioni di prima**

  La generazione ha passato 6 test su 9.

- **RGC rivisitato + dare i test all'LLM**

  La generazione ha passato 6 test e falliti 3.

- **Progettazione di un prompt per l'ottimizzazione congiunta: lato correttezza**

  La generazione ha passato 6 test su 9, analogo comportamento generato dalla chat priva di tecnica specifica. Si è proceduto a verificare se applicando una tecnica iterativa si riuscisse a sviluppare meglio un prompt iniziale già completo e specifico: è stato fornito un ulteriore prompt con dei punti da verificare a GPT, tuttavia anche la seconda generazione ha passato 6 test su 9.

  Per questo esercizio sono state condotte 4-5 chat con GPT, ma in tutte le iterazioni emerge la stessa problematica, con 6 test superati su 9. Nonostante l'utilizzo di diversi prompt, inclusi quelli che fornivano dettagli sugli errori dei test, non sono stati ottenuti miglioramenti significativi.

  L'unico approccio che ha realmente risolto tale problematica e ha permesso all'LLM di superare tutti i test è stato quello di fornire il testo completo dei 3 test che fallivano.

Analisi qualitativa del codice

Metriche qualità Chain of Thought prompting

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	9 (funzione `post`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Difficoltà di Halstead	36.62 (file `test.py`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	4.903 (file `test.py`)	Log10 > 4	Superiore alla soglia minima, sforzo elevato.
MI (Indice di Manutenibilità)	67.19 (funzione `post`)	MI ≥ 20 (alta)	Alta manutenibilità, ma inferiore rispetto ad altre funzioni.
WMC (Metodo Ponderato)	9 (funzione `post`)	Non specificata	Complessità elevata per una singola classe, ma manca una media di riferimento.
Complessità Cognitiva	18 (funzione `post`)	Non specificata	Complessità cognitiva elevata, ma manca una media di riferimento.

Prompt Ottimizzazione Qualitativa con scheletro GPT

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	3 (funzione `_create_iou`)	1-2	Leggermente più alto della media, complessità moderata.
Complessità Cognitiva	2 (funzione `_create_iou`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	9.82 (funzione `_create_iou`)	10-100	Inferiore alla media, difficoltà bassa.
Sforzo di Halstead (Log10)	3.56 (funzione `_create_iou`)	Log10 > 4	Sotto la soglia minima, sforzo moderato.
MI (Indice di Manutenibilità)	90.29 (funzione `_create_iou`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	3 (funzione `_create_iou`)	Non specificata	Complessità moderata, ma manca una media di riferimento.

Progettazione di un prompt per l'ottimizzazione congiunta: lato qualità

L'output che passa tutti i test ha ottenuto le seguenti metriche:

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	3.86 (classe `RestAPI`)	1-2	Significativamente sopra la media, complessità elevata.
Complessità Cognitiva	3 (classe `RestAPI`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	51.3 (classe `RestAPI`)	10-100	Rientra nella media, ma verso il limite superiore, difficoltà elevata.
Sforzo di Halstead (Log10)	5.11 (classe `RestAPI`)	Log10 > 4	Sopra la soglia minima, sforzo significativo richiesto.
MI (Indice di Manutenibilità)	50.12 (classe `RestAPI`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	3.86 (classe `RestAPI`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

4.3.3 ClaudeSonnet

Analisi correttezza del codice

- **Prima chat: nessuna tecnica adottata**

  La generazione ha fallito tutti i test.

- **Seconda chat: RGC**

  La generazione ha passato tutti i test

- **Terza chat: Chain of Thought Prompting**

  La prima generazione ha fallito tutti i test. Successivamente è stato introdotto un secondo prompt contenente i principali errori dei test: la seconda generazione ha passato 7 test e ne ha falliti 2. Il terzo prompt mirava a correggere specifiche logiche che l'LLM falliva per quanto riguarda alcune operazioni sul debito. La terza generazione ha passato tutti i test

- **Quarta chat: Structured Chain of Thought prompting con gli esempi e i punti di attenzioni di prima**

  La generazione ha passato tutti i test

- **RGC rivisitato + dare i test all'LLM**

  La generazione ha passato tutti i test.

Analisi qualitativa del codice

Metriche qualità RGC

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	14 (funzione `post`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Difficoltà di Halstead	41.4 (file `test.py`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	4.888 (file `test.py`)	Log10 > 4	Superiore alla soglia minima, sforzo elevato.
MI (Indice di Manutenibilità)	65.92 (funzione `post`)	MI ≥ 20 (alta)	Alta manutenibilità, ma inferiore rispetto ad altre funzioni.
WMC (Metodo Ponderato)	14 (funzione `post`)	Non specificata	Complessità elevata per una singola classe, ma manca una media di riferimento.
Complessità Cognitiva	17 (funzione `post`)	Non specificata	Complessità cognitiva elevata, ma manca una media di riferimento.

Prompt Ottimizzazione Qualitativa con scheletro GPT

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	3 (funzione `post`)	1-2	Leggermente più alto della media, complessità moderata.
Complessità Cognitiva	2 (funzione `post`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	12.03 (funzione `post`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	3.87 (funzione `post`)	Log10 > 4	Appena sotto la soglia minima, sforzo moderato.
MI (Indice di Manutenibilità)	76.37 (funzione `post`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	3 (funzione `post`)	Non specificata	Complessità moderata, ma manca una media di riferimento.

4.3.4 Approccio collaborativo: prompt design con GPT e code generation con Claude

- **Prima generazione Prompt**

  Primo prompt per l'LLM generato da GPT che verrà utilizzato per Claude:

- **Risultati del primo prompt**

  La prima generazione di Claude ha fallito tutti i test, ma per un errore sui tipi, che non è stato possibile risolvere nonostante tutti i prompt introddotti successivivamente all'errore.

- **Seconda generazione Prompt**

  Per questo motivo si è deciso di far generare un altro prompt da GPT in una nuova chat aggiungendo nelle istruzioni gli errori dei tipi ottenuti prima, sperando che da una prima generazione venissero subito risolti.

- **Risultati del secondo prompt**

  La seconda generazione ha passato tutti i test con le seguenti metriche:

Metrica	**Valore Calcolato**	**Media Tipica**	**Confronto**
CC (Complessità Ciclomatica)	4.71 (classe `RestAPI`)	1-2	Significativamente sopra la media, complessità elevata.
Complessità Cognitiva	4.13 (classe `RestAPI`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	48.75 (classe `RestAPI`)	10-100	Rientra nella media, ma verso il limite superiore, difficoltà elevata.
Sforzo di Halstead (Log10)	5.14 (classe `RestAPI`)	Log10 > 4	Sopra la soglia minima, sforzo significativo richiesto.
MI (Indice di Manutenibilità)	42.44 (classe `RestAPI`)	MI ≥ 20 (alta)	Alta manutenibilità, ma verso il limite inferiore.
WMC (Metodo Ponderato)	4.71 (classe `RestAPI`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

4.3.5 Richiesta a GPT di generazione di un prompt per il miglioramento delle metriche con codice già esistente

In seguito si è verificato se fosse possibile migliorare le metriche con un altro prompt generato da GPT e passato successivamente a Claude per un'ottimizzazione del codice.

Risultati metriche prompt per migliorare la qualità

Il codice generato da Claude passa tutti i test e ha le seguenti metriche:

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	2 (classe `RestAPI`)	1-2	Rientra nella media, complessità accettabile.
Complessità Cognitiva	0.9 (classe `RestAPI`)	Non specificata	Complessità molto bassa, ma manca una media di riferimento.
Difficoltà di Halstead	27.64 (classe `RestAPI`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	4.81 (classe `RestAPI`)	Log10 > 4	Sopra la soglia minima, sforzo significativo richiesto.
MI (Indice di Manutenibilità)	46.41 (classe `RestAPI`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	2 (classe `RestAPI`)	Non specificata	Complessità accettabile, ma manca una media di riferimento.

4.3.6 Conclusioni sulla correttezza terzo esercizio

La performance di Gemini non ha rispettato le aspettative per la maggior parte dei prompt utilizzati: l'unico che ha raggiunto un risultato conforme è stato quello che consisteva nel fornire i test all'LLM. Anche il CoT prompting, con il quarto prompt contenente i test usati, non ha influenzato in nessun modo sulla correttezza del codice. Questi risultati suggeriscono che Gemini abbia avuto una difficoltà strutturale nel gestire la complessità di questo esercizio. GPT ha mostrato risultati uniformi per tutti i prompt utilizzati, senza differenze significative tra le tecniche applicate. L'unico capace di generare una versione del programma che completasse tutti i test è stato il prompt privo di tecnica. Questo comportamento evidenzia la natura probabilistica di GPT, con generazioni che possono variare anche quando viene utilizzato lo stesso prompt. La mancanza di consistenza nei risultati sottolinea la dipendenza di GPT dal contesto e dalla casualità intrinseca del modello. Nella tecnica del prompt congiunto GPT ha presentato molte difficoltà nel raggiungere un codice corretto, che in realtà non è mai stato ottenuto se non fornendo il corpo completo dei test (una condizione poco realistica nella vita reale).

ClaudeSonnet ha ottenuto degli ottimi risultati quando la tecnica di prompting forniva suggerimenti/ragionamenti secondo le specifiche dell'esercizio: inoltre sorprende il risultato della tecnica RGC, dove l'input ha effettivamente migliorato il risultato, probabilmente grazie alla presenza di vincoli molto precisi e rilevanti per il problema. ClaudeSonnet trae vantaggio e sfrutta in modo efficiente prompt ben costruiti e specifici, adattandosi meglio alle richieste del compito rispetto agli altri modelli.

4.3.7 Conclusioni sulla qualità terzo esercizio

GPT ottiene risultati migliori, soprattutto nella complessità e nella difficiltà, nella prima generazione di codice rispetto agli altri LLM.

Mentre i risultati ottenuti dopo il prompt sull'ottimizzazione della qualità, sono:

- GPT migliora significativamente le metriche ma introduce dei fallimenti sui test (sulla gestione di bilanci a zero o negativi). Dopo aver immesso un prompt contenente gli errori sui test per permettere a GPT di apportare delle correzioni sul codice, GPT corregge il codice

e migliora notevolmente le metriche, riducendo la complessità ciclomatica e la difficoltà di Halstead, oltre ad aumentare significativamente la manutenibilità. Le metriche ottenute sono ottimali e superiori a quelle ottenute da ClaudeSonnet.

- Claude migliora le metriche rispetto alla versione iniziale, ma introduce errori sui test che non sono stati successivamente risolti nemmeno tramite un prompt correttivo. È stato necessario ricominciare la generazione di un codice corretto in una nuova chat, il quale ha permesso di ottenere un programma con metriche migliorate, ma inferiori a quelle fornite da GPT.

Sui risultati di qualità del codice attraverso la tecnica del prompt unico: le metriche sono leggermente inferiori rispetto a quelle ottenute mediante la tecnica dell'ottimizzazione; indicando che in questo caso non è stato vantaggioso accorpare tutto in un unico promt, sia per l'impegno nel ricercare esempi e consigli da fornire all'LLM, sia per il mancato raggiungimento dell'obiettivo sperato.

4.3.8 Conclusioni sulla tecnica di generazione prompt da GPT e generazione codice da Claude

Per quanto riguarda la correttezza, affinando maggiormente le richieste di GPT per la generazione del prompt, otteniamo un ottimo risultato riuscendo a superare tutti i test. Si evince che farsi aiutare da un altro LLM per la generazione di un prompt è un metodo efficace in termini di correttezza.

Per quanto riguarda la qualità del codice, anche sulla complessità otteniamo dei risultati ottimali, in confronto a quelli ottenuti nei test precedenti, mentre Difficoltà e Sforzo di Halstead rimangonono medi. Ridurre la complessità ciclomatica e cognitiva sembra aver comportato ad un aumento dello sforzo di Halstead. La manutenibilità ottiene un risultato migliore nelle generazioni dei capitoli precedenti.

4.3.9 Sintesi dell'esercizio 3

L'esercizio REST API ha confermato che problemi di maggiore complessità architetturale beneficiano significativamente di tecniche di prompt engineering avanzate, con Claude che emerge come il modello più efficace per questo tipo di sfide. L'approccio collaborativo rappresenta una direzione promettente per la ricerca futura.

4.4 Esercizio 4: Account bancario

Questo esercizio richiede l'implementazione di un sistema di gestione di conti bancari che supporti operazioni di apertura/chiusura, prelievi e depositi.

L'esercizio rappresenta un problema classico della programmazione orientata agli oggetti e della gestione di stati, caratterizzato da: - Logica di business ben definita e standardizzata - Gestione di eccezioni e validazione input - Controllo dello stato degli oggetti (conto aperto/chiuso) - Implementazione di operazioni atomiche

Questo problema è stato selezionato per valutare:

- Capacità degli LLM su problemi "da libro" ben documentati
- Efficacia su compiti standardizzati senza tecniche avanzate
- Qualità del codice generato su problemi familiari
- Confronto con problemi più complessi degli esercizi precedenti

4.4.1 Analisi correttezza del codice

Tutti gli LLM hanno superato tutti i test senza il bisogno di applicare alcuna tecnica.

4.4.2 Analisi qualitativa del codice

Gemini - Nessuna tecnica adottata

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	5 (funzione `withdraw`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Difficoltà di Halstead	27.36 (file `test.py`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	4.335 (file `test.py`)	Log10 > 4	Superiore alla soglia minima, sforzo elevato.
MI (Indice di Manutenibilità)	61.10 (file `test.py`)	MI ≥ 20 (alta)	Alta manutenibilità, ma inferiore rispetto ad altre funzioni.
WMC (Metodo Ponderato)	5 (funzione `withdraw`)	Non specificata	Complessità elevata per una singola classe, ma manca una media di riferimento.
Complessità Cognitiva	3 (funzione `withdraw`)	Non specificata	Complessità cognitiva moderata, ma manca una media di riferimento.

GPT - Nessuna tecnica adottata

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	5 (funzione `withdraw`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Difficoltà di Halstead	28.32 (file `test.py`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	4.364 (file `test.py`)	Log10 > 4	Superiore alla soglia minima, sforzo elevato.
MI (Indice di Manutenibilità)	69.63 (file `test.py`)	MI ≥ 20 (alta)	Alta manutenibilità, ma inferiore rispetto ad altre funzioni.
WMC (Metodo Ponderato)	5 (funzione `withdraw`)	Non specificata	Complessità elevata per una singola classe, ma manca una media di riferimento.
Complessità Cognitiva	3 (funzione `withdraw`)	Non specificata	Complessità cognitiva moderata, ma manca una media di riferimento.

ClaudeSonnet - Nessuna tecnica adottata

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	5 (funzione `withdraw`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Difficoltà di Halstead	29.32 (file `test.py`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	4.368 (file `test.py`)	Log10 > 4	Superiore alla soglia minima, sforzo elevato.
MI (Indice di Manutenibilità)	65.37 (file `test.py`)	MI ≥ 20 (alta)	Alta manutenibilità, ma inferiore rispetto ad altre funzioni.
WMC (Metodo Ponderato)	5 (funzione `withdraw`)	Non specificata	Complessità elevata per una singola classe, ma manca una media di riferimento.
Complessità Cognitiva	3 (funzione `withdraw`)	Non specificata	Complessità cognitiva moderata, ma manca una media di riferimento.

Prompt Ottimizzazione Qualitativa

Poiché le metriche dei tre LLM risultano praticamente uguali, è stato selezionato il seguente programma di Gemini come scheletro.

Gemini

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	2.1 (classe `BankAccount`)	1-2	Leggermente sopra la media, complessità accettabile.
Complessità Cognitiva	0.67 (classe `BankAccount`)	Non specificata	Complessità molto bassa, ma manca una media di riferimento.
Difficoltà di Halstead	25.48 (classe `BankAccount`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	4.35 (classe `BankAccount`)	Log10 > 4	Sopra la soglia minima, sforzo significativo richiesto.
MI (Indice di Manutenibilità)	57.68 (classe `BankAccount`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	2.1 (classe `BankAccount`)	Non specificata	Complessità accettabile, ma manca una media di riferimento.

GPT

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	1.91 (classe `BankAccount`)	1-2	Rientra nella media, complessità accettabile.
Complessità Cognitiva	0.5 (classe `BankAccount`)	Non specificata	Complessità molto bassa, ma manca una media di riferimento.
Difficoltà di Halstead	25 (classe `BankAccount`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	4.35 (classe `BankAccount`)	Log10 > 4	Sopra la soglia minima, sforzo significativo richiesto.
MI (Indice di Manutenibilità)	57.09 (classe `BankAccount`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	1.91 (classe `BankAccount`)	Non specificata	Complessità accettabile, ma manca una media di riferimento.

Claude

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	2 (classe `BankAccount`)	1-2	Rientra nella media, complessità accettabile.
Complessità Cognitiva	0.5 (classe `BankAccount`)	Non specificata	Complessità molto bassa, ma manca una media di riferimento.
Difficoltà di Halstead	26.19 (classe `BankAccount`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	4.37 (classe `BankAccount`)	Log10 > 4	Sopra la soglia minima, sforzo significativo richiesto.
MI (Indice di Manutenibilità)	54.87 (classe `BankAccount`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	2 (classe `BankAccount`)	Non specificata	Complessità accettabile, ma manca una media di riferimento.

4.4.3 Conclusioni sulla correttezza quarto esercizio

Tutti gli LLM (Gemini, GPT e ClaudeSonnet) hanno superato i test al primo tentativo senza necessità di tecniche particolari di prompting. Questo risultato può essere attribuito a due fattori principali:

- **Completezza del Testo dell'Esercizio:**Il testo fornito risulta molto dettagliato e includeva istruzioni chiare, esempi di eccezioni e requisiti specifici. Questo ha permesso agli LLM di comprendere facilmente il compito e di generare una soluzione corretta senza bisogno di ulteriori interventi o l'utilizzo di tecniche avanzate.
- **Familiarità con il problema:** La gestione di un account bancario è un programma comune e ampiamente trattato nel mondo della programmazione, abbondante anche nella documentazione tecnica: è probabile che gli LLM siano stati addestrati su dataset che includono specifiche simili, rendendoli particolarmente efficaci nella risoluzione di questo tipo di esercizio.

In sintesi, il quarto esercizio ha mostrato che, per problemi ben documentati e comuni, gli LLM possono eccellere senza necessità di tecniche avanzate di prompting, confermando la loro efficacia su compiti standardizzati e ben definiti.

4.4.4 Conclusioni sulla qualità quarto esercizio

Le metriche di qualità ottenute dai tre LLM risultano equivalenti, senza che nessuno prevalesse sull'altro in termini di metriche.

Il prompt per l'ottimizzazione della qualità è stato comunque molto utile poichè Gemini, GPT e ClaudeSonnet hanno tutti migliorato le metriche rispetto al codice iniziale, con riduzioni significative della complessità ciclomatica e cognitiva. I risultati sono molto simili tra i tre modelli, con leggere variazioni nella manutenibilità e nello sforzo di Halstead.

4.4.5 Sintesi dell'esercizio 4

L'esercizio del conto bancario ha confermato l'efficacia degli LLM su problemi standardizzati, dimostrando che la complessità intrinseca del problema determina la necessità di tecniche avanzate di prompt engineering. L'uniformità dei risultati tra i modelli suggerisce una convergenza verso soluzioni consolidate per questa categoria di problemi.

4.5 Esercizio 5: Somma dei numeri in un Item in parallelo - RUST

4.5.1 Gemini 2.0

Analisi correttezza del codice

- **Prima chat: nessuna tecnica adottata**

  La generazione ha manifestato un problema di compilazione.

- **Seconda chat: RGC**

  La generazione ha elaborato un problema di compilazione.

- **Terza chat: Chain of Thought Prompting**

  La prima generazione ha fallito tutti i test.

  Il secondo prompt conteneva principalmente degli errori sul Borrow Checker: con la seconda generazione il programma ha passato 5 test su 6. Il terzo prompt conteneva un aiuto per la creazione del thread: successivamente il programma ha passato tutti i test.

- **Quarta chat: Structured Chain of Thought prompting con gli esempi e i punti di attenzioni di prima**

  La generazione del programma ha passato tutti i test.

- **RGC rivisitato + dare i test all'LLM**

  La prima generazione ha generato un errore di compilazione, quindi sono stati introdotti due prompt uguali per aiutare LLM a superare questo ostacolo, uno successivo all'altro. Dopo questi prompt, quindi alla terza generazione, il codice ha passato tutti i test.

Analisi qualitativa del codice

Metriche Structured Chain of Thought prompting con gli esempi

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	5 (funzione `parallel_slice_sum`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Complessità Cognitiva	4 (funzione `parallel_slice_sum`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	30.16 (funzione `parallel_slice_sum`)	10-100	Rientra nella media, ma verso il limite superiore, indicando una difficoltà elevata.
Sforzo di Halstead (Log10)	4.43 (funzione `parallel_slice_sum`)	Log10 > 4	Superiore alla soglia minima, sforzo elevato.
MI (Indice di Manutenibilità)	80.25 (funzione `parallel_slice_sum`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	5 (funzione `parallel_slice_sum`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

Prompt Ottimizzazione Qualitativa

In questo caso è stato fornito a Gemini lo scheletro del programma di GPT, più il prompt per l'ottimizzazione. Gemini ha introdotto un errore di compilazione il quale è stato risolto tramite un prompt contenente il testo dell'errore, quindi abbiamo potuto analizzare il codice dal punto di vista qualitativo:

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	4 (funzione `parallel_slice_sum`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Complessità Cognitiva	3 (funzione `parallel_slice_sum`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	30.79 (funzione `parallel_slice_sum`)	10-100	Rientra nella media, ma verso il limite superiore, difficoltà elevata.
Sforzo di Halstead (Log10)	4.36 (funzione `parallel_slice_sum`)	Log10 > 4	Sopra la soglia minima, sforzo significativo richiesto.
MI (Indice di Manutenibilità)	74.18 (funzione `parallel_slice_sum`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	4 (funzione `parallel_slice_sum`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

4.5.2 GPT

Analisi correttezza del codice

- **Prima chat: nessuna tecnica adottata**

  La generazione ha elaborato un problema di compilazione.

- **Seconda chat: RGC**

  La generazione ha prodotto un problema di compilazione.

- **Terza chat: Chain of Thought Prompting**

  Questa volta il programma ha passato tutti i test alla seconda generazione, dopo aver avuto un problema di compilazione e aver ricevuto come input un prompt per la sua risoluzione.

- **Quarta chat: Structured Chain of Thought prompting con gli esempi**

  Anche qui, come con il CoT, la prima generazione ha avuto un problema di compilazione. Si è deciso di verificare l'efficacia della tecnica immettendo un prompt per far risolvere a GPT questo ostacolo, successivamente il programma ha superato tutti e sei i test.

- **RGC rivisitato + dare i test all'LLM**

  La generazione ha passato tutti i test

Analisi qualitativa del codice

Metriche Structured Chain of Thought prompting con gli esempi

Metrica	Valore Calcolato	**Media Tipica**	**Confronto**
CC (Complessità Ciclomatica)	4 (funzione `parallel_slice_sum`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Complessità Cognitiva	3 (funzione `parallel_slice_sum`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	28.42 (funzione `parallel_slice_sum`)	10-100	Rientra nella media, ma verso il limite superiore, indicando una difficoltà elevata.
Sforzo di Halstead (Log10)	4.31 (funzione `parallel_slice_sum`)	Log10 > 4	Superiore alla soglia minima, sforzo elevato.
MI (Indice di Manutenibilità)	76.66 (funzione `parallel_slice_sum`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	4 (funzione `parallel_slice_sum`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

Prompt Ottimizzazione Qualitativa con scheletro GPT

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	2 (funzione `parallel_slice_sum`)	1-2	Rientra nella media, complessità accettabile.
Complessità Cognitiva	1 (funzione `parallel_slice_sum`)	Non specificata	Complessità molto bassa, ma manca una media di riferimento.
Difficoltà di Halstead	12.73 (funzione `parallel_slice_sum`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	3.53 (funzione `parallel_slice_sum`)	Log10 > 4	Sotto la soglia minima, sforzo moderato richiesto.
MI (Indice di Manutenibilità)	94.71 (funzione `parallel_slice_sum`)	MI $\geq$ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	2 (funzione `parallel_slice_sum`)	Non specificata	Complessità accettabile, ma manca una media di riferimento.

4.5.3 Claude

Analisi correttezza del codice

- **Prima chat: nessuna tecnica adottata**

  La generazione ha riportato un problema di compilazione.

- **Seconda chat: RGC**

  La generazione ha evidenziato un problema di compilazione.

- **Terza chat: Chain of Thought Prompting**

  La generazione ha elaborato un problema di compilazione. Si è quindi introdotto un secondo prompt contenente il testo dell'errore del borrow checker: la seconda generazione ha passato tutti i test.

- **Quarta chat: Structured Chain of Thought prompting con gli esempi e i punti di attenzioni di prima**

  Anche qui come il CoT, la prima generazione ha avuto un problema di compilazione. Si è comunque deciso di verificare l'efficacia della tecnica immettendo un prompt per far risolvere a Claude questo ostacolo, in seguito il programma ha superato tutti e sei i test.

- **RGC rivisitato fornendo i test all'LLM**

  Come per le altre tecniche, la prima generazione ha avuto un problema di compilazione. Dopo il prompt sull'errore del borrow checker, il programma ha superato tutti e sei i test.

- **Progettazione di un prompt per l'ottimizzazione congiunta: lato correttezza**

  L'output di Claude dopo il prompt unificato a subito passato tutti e sei i test.

Analisi qualitativa del codice

Metriche qualità Structured Chain of Thought prompting con gli esempi

Metrica	Valore Calcolato	**Media Tipica**	**Confronto**
CC (Complessità Ciclomatica)	4 (funzione `parallel_slice_sum`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Complessità Cognitiva	3 (funzione `parallel_slice_sum`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	30.33 (funzione `parallel_slice_sum`)	10-100	Rientra nella media, ma verso il limite superiore, indicando una difficoltà elevata.
Sforzo di Halstead (Log10)	4.40 (funzione `parallel_slice_sum`)	Log10 > 4	Superiore alla soglia minima, sforzo elevato.
MI (Indice di Manutenibilità)	73.72 (funzione `parallel_slice_sum`)	MI $\geq$ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	4 (funzione `parallel_slice_sum`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

Prompt Ottimizzazione Qualitativa con scheletro GPT

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	2 (funzione `parallel_slice_sum`)	1-2	Rientra nella media, complessità accettabile.
Complessità Cognitiva	1 (funzione `parallel_slice_sum`)	Non specificata	Complessità molto bassa, ma manca una media di riferimento.
Difficoltà di Halstead	13.22 (funzione `parallel_slice_sum`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	3.48 (funzione `parallel_slice_sum`)	Log10 > 4	Sotto la soglia minima, sforzo moderato richiesto.
MI (Indice di Manutenibilità)	102.09 (funzione `parallel_slice_sum`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	2 (funzione `parallel_slice_sum`)	Non specificata	Complessità accettabile, ma manca una media di riferimento.

Progettazione di un prompt per l'ottimizzazione congiunta: lato qualità

L'output che passa tutti i test ha ottenuto le seguenti metriche:

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	3 (funzione `parallel_slice_sum`)	1-2	Sopra la media, complessità elevata.
Complessità Cognitiva	2 (funzione `parallel_slice_sum`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	31.94 (funzione `parallel_slice_sum`)	10-100	Rientra nella media, ma verso il limite superiore, difficoltà elevata.
Sforzo di Halstead (Log10)	4.44 (funzione `parallel_slice_sum`)	Log10 > 4	Sopra la soglia minima, sforzo significativo richiesto.
MI (Indice di Manutenibilità)	71.56 (funzione `parallel_slice_sum`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	3 (funzione `parallel_slice_sum`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

4.5.4 Approccio collaborativo: prompt design con Claude e code generation con GPT

Risultati del primo prompt

La prima generazione ha prodotto un errore di compilazione, evidenziando difficoltà nell'interpretazione delle istruzioni o nella gestione del codice richiesto.

Seconda e terza generazione Prompt

La seconda generazione ha prodotto un ulteriore errore di compilazione, simile al precedente. Tuttavia, dopo aver fornito all'LLM i dettagli degli errori di compilazione, la terza generazione ha finalmente superato tutti i test.

Risultati del terzo prompt

La terza generazione ha passato tutti i test con le seguenti metriche:

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	7 (funzione `parallel_slice_sum`)	1-2	Significativamente sopra la media, complessità elevata.
Complessità Cognitiva	8 (funzione `parallel_slice_sum`)	Non specificata	Complessità elevata, ma manca una media di riferimento.
Difficoltà di Halstead	35.75 (funzione `parallel_slice_sum`)	10-100	Rientra nella media, ma verso il limite superiore, difficoltà elevata.
Sforzo di Halstead (Log10)	4.65 (funzione `parallel_slice_sum`)	Log10 > 4	Sopra la soglia minima, sforzo significativo richiesto.
MI (Indice di Manutenibilità)	65.51 (funzione `parallel_slice_sum`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	7 (funzione `parallel_slice_sum`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

4.5.5 Richiesta a Claude di generazione di un prompt per il miglioramento delle metriche con codice dal prompt unificato

È stato verificato se fosse possibile migliorare le metriche con un altro prompt generato da Claude e passarlo a GPT per un'ottimizzazione del codice.

Risultati metriche prompt per migliorare la qualità

Il codice ottimizzato generato presenta un errore di compilazione, quindi è stato ri-immesso un altro prompt con il testo dell'errore, in questo caso GPT ha generato un codice che ha passato tutti i test, con le seguenti metriche:

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	4 (funzione `parallel_slice_sum`)	1-2	Sopra la media, complessità elevata.
Complessità Cognitiva	3 (funzione `parallel_slice_sum`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	25.3 (funzione `parallel_slice_sum`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	4.34 (funzione `parallel_slice_sum`)	Log10 > 4	Sopra la soglia minima, sforzo significativo richiesto.
MI (Indice di Manutenibilità)	75.07 (funzione `parallel_slice_sum`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	4 (funzione `parallel_slice_sum`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

4.5.6 Conclusioni sulla correttezza quinto esercizio

Le chat di Gemini hanno tutte riportato un errore di compilazione al primo prompt, anche nel caso in cui veniva richiesto all'LLM di gestire il Borrow Checker di RUST: l'unica eccezione è stata la tecnica del Structured Chain, con il quale si è ottenuto il miglior risultato già dal primo prompt. Un aspetto interessante è emerso nella quinta chat: il terzo e il quarto prompt, pur essendo identici, hanno generato risultati diversi, evidenziando la natura probabilistica della generazione di codice da parte di Gemini.

Anche per GPT tutte le chat, tranne quella con i test dati come input, hanno generato un errore di compilazione al primo prompt, anche per quelle tecniche in cui si richiedeva all'LLM di gestire il Borrow Checker di RUST. Il comportamento è stato simile a quello di Gemini, ma GPT ha raggiunto un codice corretto utilizzando un prompt in meno nella tecnica Chain of Thought (CoT).

ClaudeSonnet ha mostrato un comportamento simile, anche in questo caso tutte le chat hanno generato un errore di compilazione al primo prompt, anche per quelle tecniche in cui veniva richiesto all'LLM di gestire il Borrow Checker di RUST. Tuttavia, ha raggiunto risultati migliori con un numero inferiore di prompt. Sorprendentemente, la tecnica di fornire i test come input, che in altri contesti si era dimostrata utile, ha avuto scarsi risultati in questo esercizio.

Inoltre Claude risulta efficace e soddisfacente per quanto riguarda il prompt unificato dal punto di vista della correttezza del codice, con tutti e sei i test superati già dopo il primo prompt.

4.5.7 Conclusioni sulla qualità quinto esercizio

In questo caso Claudesonnet e GPT ottengono risultati leggermente superiori a Gemini, tuttavia quest'ultimo è riuscito a generare un programma senza errori con un solo prompt.

I risultati ottenuti dopo il prompt sull'ottimizzazione della qualità, sono:

- Gemini dopo la richiesta di miglioramento del codice, introduce un errore di compilazione iniziale. Successivamente ad un altro prompt contenente il testo dell'errore, Gemini migliora marginalmente le metriche rispetto allo scheletro iniziale
- GPT ottiene un miglioramento significativo, riducendo la complessità ciclomatica e cognitiva, oltre ad aumentare la manutenibilità.
- ClaudeSonnet migliora le metriche in modo simile a GPT, ottenendo risultati eccellenti nella qualità del codice.

Sui risultati di qualità del codice riguardanti la tecnica del prompt unico: un buon risultato sulle metriche, che però non eguagliano quelle ottenute nel capitolo precedente, soprattuttto sulla manutenibilità e sulla difficoltà di Halstead. Rimane incerto se sia stato vantaggioso utilizzare un prompt così complesso: da un lato, il risultato qualitativo è inferiore rispetto alle tecniche iterative, dall'altro è stato raggiunto con un numero inferiore di prompt, riducendo il tempo necessario.

4.5.8 Conclusioni sulla tecnica di generazione prompt da Claude e generazione codice da GPT

Anche in questo caso, la prima generazione ha prodotto errori di compilazione. Dopo iterazioni successive, il codice generato ha superato i test, ma con metriche di complessità relativamente alte. Il prompt correttivo per migliorare la qualità ha portato a una riduzione significativa della complessità ciclomatica e cognitiva, oltre a un aumento dell'indice di manutenibilità.

4.5.9 Sintesi dell'esercizio 5

Questo esercizio in RUST conferma una sfida importante e significativa per gli LLM; la programmazione parallela oltre alla combinazione di un linguaggio più complesso fa emergere delle difficoltà da parte degli LLM nella generazione del codice. Claude emerge come il modello più efficace per questo esercizio, mentre l'efficacia delle tecniche di prompt engineering diventa essenziale per il raggiungimento delle specifiche.

4.6 Esercizio 6: Frequenza lettere in un file - Calcolo parallelo - RUST

Questo esercizio richiede l'implementazione di un sistema per il calcolo parallelo della frequenza delle lettere in testi, utilizzando thread multipli per ottimizzare le prestazioni su dataset di grandi dimensioni.

L'esercizio rappresenta la sfida più complessa della serie, combinando: - Linguaggio Rust avanzato: Gestione complessa di ownership, borrowing e lifetime - Algoritmi di parallelizzazione: Suddivisione del lavoro, sincronizzazione e aggregazione risultati - Gestione delle strutture dati condivise: HashMap thread-safe e gestione della memoria - Ottimizzazione delle prestazioni: Bilanciamento del carico e minimizzazione dell'overhead

Questo problema è stato selezionato per valutare:

- Capacità degli LLM nella gestione di problemi computazionali complessi
- Competenza nella programmazione parallela avanzata con Rust
- Efficacia su problemi che richiedono multiple competenze tecniche simultanee
- Limiti degli LLM su task che combinano sintassi complessa e concetti algoritmici avanzati

4.6.1 Gemini 2.0

Analisi correttezza del codice

- **Prima chat: nessuna tecnica adottata**

  La generazione ha manifestato un problema di compilazione.

- **Seconda chat: RGC**

  La generazione ha elaborato un problema di compilazione.

- **Terza chat: Chain of Thought Prompting**

  La generazione ha riportato un problema di compilazione. Il secondo prompt conteneva principalmente degli errori sul Borrow Checker: con la seconda generazione il programma ha passato tutti i test.

- **Quarta chat: Structured Chain of Thought prompting con gli esempi**

  La generazione ha riportato un problema di compilazione. Si è comunque deciso di verificare l'efficacia della tecnica immettendo un prompt per far risolvere a Gemini questo ostacolo La seconda generazione faceva "panic" sull'ambiente di test, che si è cercato di risolvere con un altro prompt. La terza generazione ha passato tutti i test.

- **RGC rivisitato + dare i test all'LLM**

  In questa casistica si è provato a far passare tutti i test a Gemini, continuando con prompt mirati, tuttavia ci si è dovuti fermare perchè nonostante i prompt, LLM non riusciva a correggere la sua soluzione e generava sempre un codice che falliva sempre nello stesso punto.

Analisi qualitativa del codice

Metriche Terza chat: Chain of Thought Prompting

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	5 (funzione `frequency`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Complessità Cognitiva	5 (funzione `frequency`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	29.5 (funzione `frequency`)	10-100	Rientra nella media, ma verso il limite superiore, difficoltà elevata.
Sforzo di Halstead (Log10)	4.54 (funzione `frequency`)	Log10 > 4	Sopra la soglia minima, sforzo significativo richiesto.
MI (Indice di Manutenibilità)	70.44 (funzione `frequency`)	MI $\geq$ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	5 (funzione `frequency`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

Prompt Ottimizzazione Qualitativa

In questo caso è stato fornito a Gemini lo scheletro del programma di GPT, più il prompt per l'ottimizzazione.

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	4 (funzione `frequency`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Complessità Cognitiva	3 (funzione `frequency`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	19.69 (funzione `frequency`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	4.14 (funzione `frequency`)	Log10 > 4	Sopra la soglia minima, sforzo significativo richiesto.
MI (Indice di Manutenibilità)	77.27 (funzione `frequency`)	MI $\geq$ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	4 (funzione `frequency`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

4.6.2 GPT

Analisi correttezza del codice

- **Prima chat: nessuna tecnica adottata**

  La generazione ha generato un problema di compilazione.

- **Seconda chat: RGC**

  La generazione ha generato un problema di compilazione.

- **Terza chat: Chain of Thought Prompting**

  La prima generazione ha avuto un panic, si è quindi introdotto un prompt che mostrasse l'errore a GPT, il quale poi ha generato un codice che ha passato tutti i test.

- **Quarta chat: Structured Chain of Thought prompting con gli esempi**

  Anche qui come il CoT, la prima generazione ha avuto un problema di compilazione. Si è comunque deciso di verificare l'efficacia della tecnica immettendo più prompt per far risolvere a GPT questo ostacolo, alla terza generazione il programma ha passato tutti i test.

- **RGC rivisitato + dare i test all'LLM**

  La generazione ha passato tutti i test

Analisi qualitativa del codice

Metriche Chain of Thought prompting

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	4 (funzione `frequency`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Complessità Cognitiva	3 (funzione `frequency`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	36.4 (funzione `frequency`)	10-100	Rientra nella media, ma verso il limite superiore, difficoltà elevata.
Sforzo di Halstead (Log10)	4.67 (funzione `frequency`)	Log10 > 4	Sopra la soglia minima, sforzo significativo richiesto.
MI (Indice di Manutenibilità)	68.69 (funzione `frequency`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	4 (funzione `frequency`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

Prompt Ottimizzazione Qualitativa con scheletro GPT

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	4 (funzione `frequency`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Complessità Cognitiva	3 (funzione `frequency`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	20.54 (funzione `frequency`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	4.16 (funzione `frequency`)	Log10 > 4	Sopra la soglia minima, sforzo significativo richiesto.
MI (Indice di Manutenibilità)	77.26 (funzione `frequency`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	4 (funzione `frequency`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

4.6.3 ClaudeSonnet

Analisi correttezza del codice

- **Prima chat: nessuna tecnica adottata**

  La generazione ha passato tutti i test.

- **Seconda chat: RGC**

  La generazione ha passato tutti i test.

- **Terza chat: Chain of Thought Prompting**

  Non è stato possibile testarlo, in quanto già alla prima generazione il programma passava tutti i test.

- **Quarta chat: Structured Chain of Thought prompting con gli esempi e i punti di attenzioni di prima**

  La generazione del programma ha passato tutti i test.

- **RGC rivisitato + dare i test all'LLM**

  Nonostante gli esiti precedenti, la prima generazione ha avuto un problema di compilazione. Dopo il prompt sull'errore del borrow checker, il programma ha superato tutti e sei i test.

- **Progettazione di un prompt per l'ottimizzazione congiunta: lato correttezza**

  L'output di Claude dopo il prompt unificato ha passato tutti i test.

Analisi qualitativa del codice

Metriche qualità Structured Chain of Thought prompting con gli esempi

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	6 (funzione `frequency`)	1-2	Significativamente superiore alla media, indicando una complessità elevata.
Complessità Cognitiva	6 (funzione `frequency`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	32.74 (funzione `frequency`)	10-100	Rientra nella media, ma verso il limite superiore, difficoltà elevata.
Sforzo di Halstead (Log10)	4.60 (funzione `frequency`)	Log10 > 4	Sopra la soglia minima, sforzo significativo richiesto.
MI (Indice di Manutenibilità)	68.50 (funzione `frequency`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	6 (funzione `frequency`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

Prompt Ottimizzazione Qualitativa con scheletro GPT

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	2 (funzione `frequency`)	1-2	Rientra nella media, complessità accettabile.
Complessità Cognitiva	1 (funzione `frequency`)	Non specificata	Complessità molto bassa, ma manca una media di riferimento.
Difficoltà di Halstead	12 (funzione `frequency`)	10-100	Rientra nella media, difficoltà accettabile.
Sforzo di Halstead (Log10)	3.43 (funzione `frequency`)	Log10 > 4	Sotto la soglia minima, sforzo moderato richiesto.
MI (Indice di Manutenibilità)	99.61 (funzione `frequency`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	2 (funzione `frequency`)	Non specificata	Complessità accettabile, ma manca una media di riferimento.

Progettazione di un prompt per l'ottimizzazione congiunta: lato qualità

L'output che passa tutti i test ha ottenuto le seguenti metriche:

Metrica	Valore Calcolato	Media Tipica	Confronto
CC (Complessità Ciclomatica)	5 (funzione `frequency`)	1-2	Significativamente sopra la media, complessità elevata.
Complessità Cognitiva	4 (funzione `frequency`)	Non specificata	Complessità moderata, ma manca una media di riferimento.
Difficoltà di Halstead	29 (funzione `frequency`)	10-100	Rientra nella media, ma verso il limite superiore, difficoltà elevata.
Sforzo di Halstead (Log10)	4.48 (funzione `frequency`)	Log10 > 4	Sopra la soglia minima, sforzo significativo richiesto.
MI (Indice di Manutenibilità)	70.31 (funzione `frequency`)	MI ≥ 20 (alta)	Alta manutenibilità.
WMC (Metodo Ponderato)	5 (funzione `frequency`)	Non specificata	Complessità elevata, ma manca una media di riferimento.

4.6.4 Conclusioni sulla correttezza sesto esercizio

Con l'ultima chat, Gemini non è riuscito a generare un codice che superasse tutti i test, bloccandosi ricorrententemnte sugli stessi problemi e non riuscendo ad avanzare nonostante i successivi prompt: gli errori generati erano simili, suggerendo una difficoltà strutturale del modello nell'adattarsi e correggere errori persistenti. La terza chat ha prodotto risultati molto positivi, con il codice generato che ha superato tutti i test dopo un errore iniziale di compilazione. Questo dimostra che, con il giusto approccio, Gemini può generare codice corretto anche per problemi complessi. La scelta di aumentare il numero di prompt per migliorare il risultato ha permesso di affrontare gli errori di compilazione e di "panic", ma non ha prodotto miglioramenti significativi rispetto alla terza chat.

GPT ha avuto performance simili a Gemini, poichè la terza chat è quella che ottenuto il risultato migliore con meno prompt utilizzati. L'errore di compilazione è stato inevitabile nonostante i prompt iniziali presentassero molti punti di attenzione su questa tipologia di errori. Anche in questo caso per la quarta tecnica si è provato ad aumentare il numero di prompt per verificare se si riuscisse a ottenere un risultato corretto, in quanto gli errori erano principalmente di compilazione o di panic. Tuttavia, i risultati non hanno mostrato miglioramenti significativi rispetto alla terza chat, evidenziando una limitazione nella capacità di GPT di gestire scenari complessi.

ClaudeSonnet ha ottenuto i risultati migliori, generando codice corretto fin dalla prima chat e mantenendo un livello di performance costante in tutte le iterazioni. Questo indica una capacità superiore nella gestione di problemi complessi e nell'implementazione di concetti avanzati come il parallelismo. A differenza di Gemini e GPT, ClaudeSonnet non ha mostrato difficoltà persistenti, dimostrando una maggiore affidabilità e precisione nella risoluzione dell'esercizio.

Considerando il promp unificato, Claude dal punto di vista della correttezza del codice è immediatamente soddisfacente, in quanto è riuscito a superare tutti i test.

4.6.5 Conclusioni sulla qualità sesto esercizio

Trascurando l'indice di Manutenibilità, risalta GPT con una complessità inferiore rispetto alle altre LLM.

I risultati ottenuti dopo il prompt sull'ottimizzazione della qualità, sono:

- Gemini ha migliorato leggermente la manutenibilità rispetto al codice iniziale, ma il resto delle metriche è rimasto invariato.
- GPT ha ottenuto buoni risultati, ma senza miglioramenti significativi rispetto al codice iniziale.
- ClaudeSonnet ha migliorato notevolmente le metriche, ottenendo il risultato migliore in questo esercizio, con una riduzione significativa della complessità e un aumento della manutenibilità.

Sui risultati di qualità del codice relativi alla tecnica del prompt unico: un buon risultato sulle metriche, che però non eguagliano quelle ottenute nell'approccio iterativo, soprattuttto sulla manutenibilità e sulla difficoltà di Halstead.

Rimane incerto se sia stato vantaggioso utilizzare un prompt così complesso: da un lato, il risultato qualitativo è inferiore rispetto al capitolo precedente, dall'altro è stato raggiunto con un numero inferiore di prompt, riducendo il tempo necessario.

4.6.6 Sintesi dell'esercizio 6

Questo esercizio sul calcolo parallelo della frequenza delle lettere si è dimostrato un test limite per tutti i modelli, rivelando differenze significative e prestazionali. L'unico modello che emerge, capace di gestire autonomamente problemi di questa complessità, è Claude, mentre gli altri richiedono supporto intensivo di prompt engineering. Questo esercizio definisce effettivamente i confini attuali delle capacità degli LLM in programmazione avanzata.

Capitolo 5

Valutazione su larga scala dell'efficacia degli LLM

In questo capitolo è stato effettuata la valutazione degli LLM e delle tecniche prima verificate attraverso un numero di 150 esercizi. Come LLM è stato considerato GPT 4o, con l'obiettivo verificare di la capacità di un modello di generare codice corretto e qualitativo su un dataset ampio e variegato.

5.1 Modello utilizzato

Nel corso dell'analisi, ClaudeSonnet si è dimostrato un modello più robusto e scalabile, ottenendo risultati di qualità superiore in diverse metriche, soprattutto in scenari complessi. Tuttavia, GPT 4o ha mostrato prestazioni analoghe, dimostrando una capacità di adattamento e generazione di codice di qualità comparabile.

La scelta di utilizzare GPT 4o come modello principale per questo test è stata dettata dalla sua popolarità e dalla diffusione. Essendo uno degli LLM più utilizzati e conosciuti, GPT offre un'ampia applicabilità pratica e una maggiore accessibilità per gli sviluppatori e le aziende. Inoltre, la sua vasta base di utenti e supporto tecnico lo rendono una scelta ideale per valutazioni su larga scala e per l'implementazione in progetti reali.

5.2 Esercizi selezionati

Per questa analisi è stato scelto un dataset di esercizi che fossero di difficoltà adeguata, evitando problemi troppo elementari che non avrebbero fornito spunti interessanti per la valutazione delle capacità di GPT-4o. Alcuni esercizi del tipo:

```
Write a python function to find the first repeated character in a given string.
```

sono stati esclusi o presi poco in considerazione perchè venivano risolti molto facilmente da GPT 4o e non avrebbero contribuito alla valutazione della qualità del codice.

Quindi è stato adottato un approccio mirato sulla tipologia degli esercizi selezionati, ovvero si sono considerati degli esercizi che:

- Richiedessero una logica articolata e l'uso di strutture di controllo avanzate (ad esempio, condizioni nidificate o cicli intricati).
- Presentassero casi limite o eccezioni da gestire, come input non validi o situazioni impreviste.
- Fossero caratterizzati da requisiti ambigui o dettagliati, che richiedessero una corretta interpretazione delle specifiche.

Per esempio, tra gli esercizi selezionati vi è *Go Counting* (disponibile su `https://exercism.org/tracks/python/exercises/go-counting`).

```
Instructions
Count the scored points on a Go board.

In the game of go points are gained by completely encircling empty intersections with your stones.
The encircled intersections of a player are known as its territory.

Calculate the territory of each player. You may assume that any stones that have been stranded
in enemy territory have already been taken off the board.

Determine the territory which includes a specified coordinate.

Multiple empty intersections may be encircled at once and for encircling only horizontal
and vertical neighbors count. In the following diagram the stones which matter are marked
"O" and the stones that don't are marked "I" (ignored). Empty spaces represent empty intersections.

+----+
|IOOI|
|O  O|
|O OI|
|IOI |
+----+

To be more precise an empty intersection is part of a player's territory
if all of its neighbors are either stones of that player or empty
intersections that are part of that player's territory.

Exception messages
Sometimes it is necessary to raise an exception. When you do this, you should always
include a meaningful error message to indicate what the source of the error is.
This makes your code more readable and helps significantly with debugging.
For situations where you know that the error source will be a certain type,
you can choose to raise one of the built in error types,
but should still include a meaningful message.

This particular exercise requires that you use the raise statement to "throw"
a ValueError when given invalid coordinates. The tests will only pass if you
both raise the exception and include a message with it.

To raise a ValueError with a message, write the message as an argument to the exception type:

# when the coordinates for the piece are invalid
raise ValueError('Invalid coordinate')
```

La selezione degli esercizi ha quindi permesso di focalizzarsi su problemi che rappresentassero una sfida concreta per l'LLM, fornendo al contempo dati significativi per l'analisi delle metriche di qualità del codice.

Inoltre, ogni esercizio aveva un determinato numero di test per valture la correttezza del codice generato dall'LLM, ed ogni generazione di codice veniva analizzata tramite RustCode Analysis e venivano estratte le seguenti metriche:

- Complessità Ciclomatica Sum
- Complessità Ciclomatica Average
- Complessità Cognitiva Sum
- Complessità Cognitiva Average
- Difficoltà di Halstead
- Sforzo di Halstead (log10)
- Indice manutenibilità
- WMC Metodo ponderato

5.2.1 Metodologia delle generazioni di codice

È stata considerata solo la prima generazione del codice a meno che non ci fossero semplici errori di compilazione che potevano essere risolti facilmente dall'LLM, non si è cercato di correggere o migliorare la qualità del codice tramite prompt successivi.

5.2.2 Costruzione del Prompt

Si è cercato di costruire un prompt unificato sia per la correttezza che per la qualità del codice, riprendendo molti dei comportamenti che sono stati valutati nei test precedenti.

Si è attuata la tecnica del prompt unificato e non la tecnica iterativa, in quanto impossibile strutturare un processo dove per ogni codice generato dall'LLM si desse come input un altro prompt mirato al miglioramento della qualità del codice nel contesto dell'esercizio stesso. Si è quindi deciso un unico prompt, con le tecniche riportate nel capitolo "Progettazione di un prompt per l'ottimizzazione congiunta di correttezza e qualità" sulla qualità del codice, aggiungendo la tecnica Structured Chain, in quanto aveva riportato dei buoni risultati per la generazione di un codice corretto. Ciò è stato effettuato tramite il seguente testo.

```
Dovresti prima scrivere un processo approssimativo di problem-solving usando tre strutture
di programmazione (cioè, sequenziale, branch e loop) e poi produrre il codice finale
```

Per valutare correttamente la correttezza del codice tramite i test si è richiesto all'LLM il seguente limite:

```
NON modificare il nome delle funzioni, per rispettare l'esecuzione dei test,
puoi aggiungerne, ma le funzioni principali devono chiamarsi allo stesso modo.
```

In sintesi, il prompt unificato è stato costruito per bilanciare correttezza e qualità, integrando tecniche testate nei capitoli precedenti e linee guida pratiche per migliorare le metriche di qualità del codice. Il prompt selezionato alla fine è stato questo:

```
Ti verrà richiesto di risolvere vari esercizi in Python in questa chat.
Dovrai seguire le istruzioni e generarmi un codice corretto che passi tutti i test.

Per ogni prompt ti verrà dato le istruzioni dell'esercizio e la struttura dell funzione.
Dovresti prima scrivere un processo approssimativo di problem-solving usando tre strutture
di programmazione (cioè, sequenziale, branch e loop) e poi produrre il codice finale.

Qui di seguito delle linee guida del risultato aspettato:

- NON modificare il nome delle funzioni, per rispettare l'esecuzione dei test, puoi aggiungerne,
ma le funzioni principali devono chiamarsi allo stesso modo.
- Utilizza strutture di controllo semplici e riduci al minimo la complessità.
- Evita annidamenti profondi e suddividi il codice in funzioni più piccole se necessario.
- Usa nomi di variabili descrittivi, evita logica complessa e
documenta ogni passaggio con commenti chiari.
- Evita condizioni annidate e cicli intricati.
- Usa operatori e costrutti di linguaggio chiari e intuitivi.
- Evita codice duplicato e usa costrutti semplici e leggibili.
- Modifica il codice in modo che sia facile da implementare e
comprendere, riducendo lo sforzo richiesto per la lettura e la modifica.
- Assicurati che il codice sia leggibile, ben documentato e utilizzi
buone pratiche di programmazione.
- Evita operazioni ridondanti e usa strutture dati appropriate.
- Correggi in modo scalabile, ottimizzando l'uso delle risorse.
- Evita effetti collaterali e separa la logica dal codice
di input/output.
- Mantieni i messaggi di errore senza tradurli o cambiarli
```

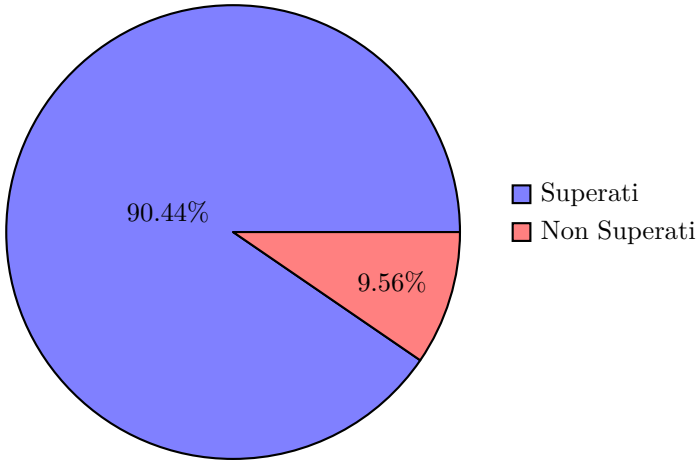
Ogni prompt successivo era:

```
[Istruzioni esercizio]

Contenuto di Main.py

[Struttura esercizio]
```

5.3 Risultati e Grafici

Percentuale Test Superati

N° di test superati: 90,44



Complessità Ciclomatica Sum

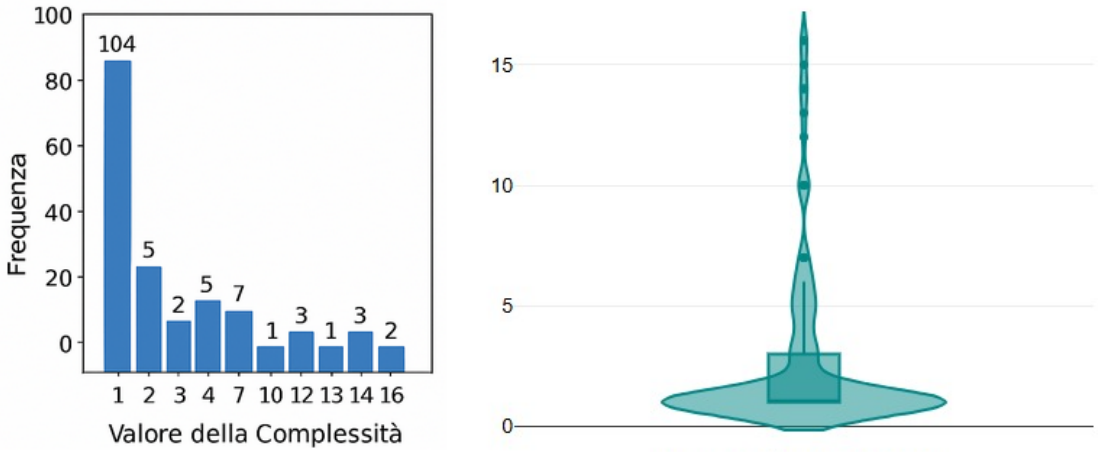


Figura 5.1: Complessità ciclomatica Sum (Istogramma e Violin Plot)

- **Media**: 2,84
- **Mediana**: 1
- **Minimo**: 1
- **Massimo**: 16
- **Osservazione**: La maggior parte dei valori è 1, indicando generalmente una bassa complessità.

Complessità Ciclomatica Average

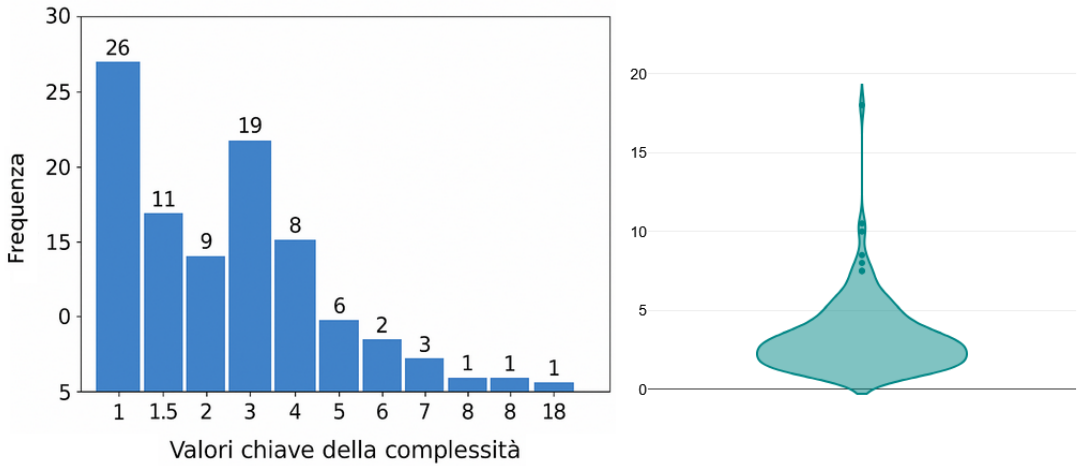


Figura 5.2: Complessità ciclomatica Average (Istogramma e Violin Plot)

- **Media**: 3,31
- **Mediana**: 2,6
- **Minimo**: 1
- **Massimo**: 18
- **Osservazione**: Leggermente superiore ai valori ottimali (1–2).

Complessità Cognitiva Sum

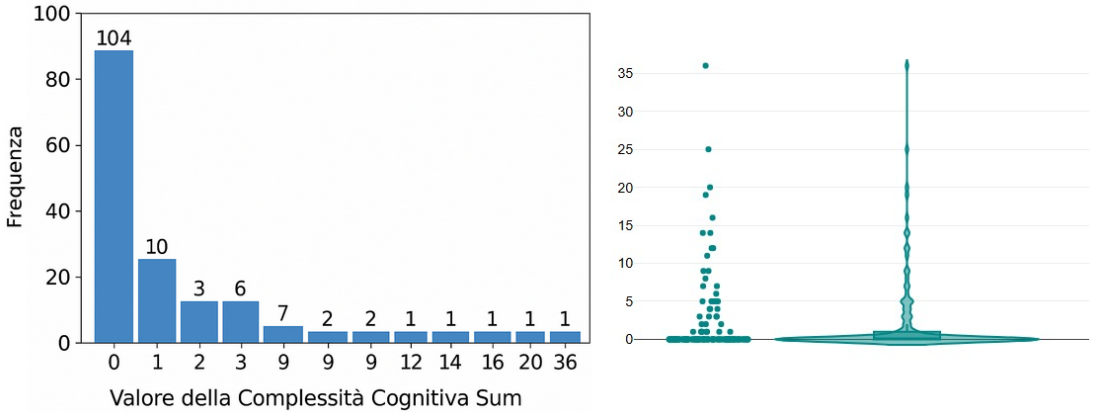


Figura 5.3: Complessità Cognitiva Sum (Istogramma e Violin Plot)

- **Media**: 2,47
- **Mediana**: 0
- **Minimo**: 0
- **Massimo**: 36
- **Osservazione**: La maggioranza dei casi ha valore 0, indicando ottima comprensibilità.

Complessità Cognitiva Average

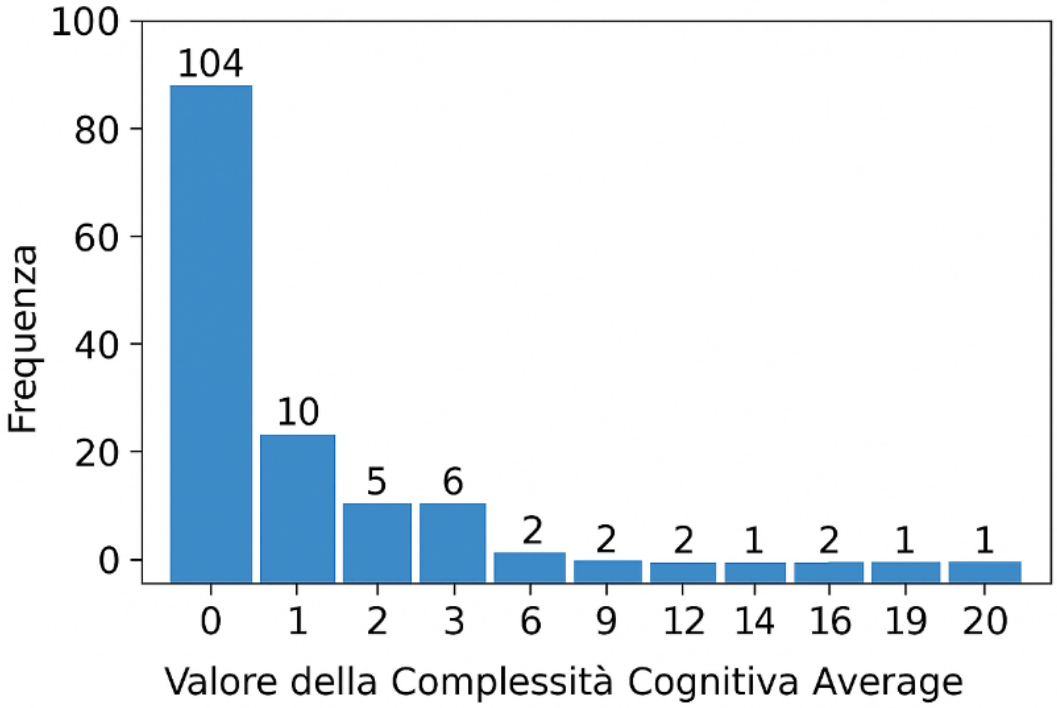


Figura 5.4: Complessità cognitiva Average (Istogramma)

- **Media**: 3,42
- **Mediana**: 3,5
- **Minimo**: 0
- **Massimo**: 36
- **Osservazione**: La maggior parte dei valori è compresa tra 0 e 3, indicando una buona comprensibilità nella media. Tuttavia, il valore massimo di 36 evidenzia che alcuni esercizi hanno generato codice con logica più difficile da seguire.

Difficoltà di Halstead

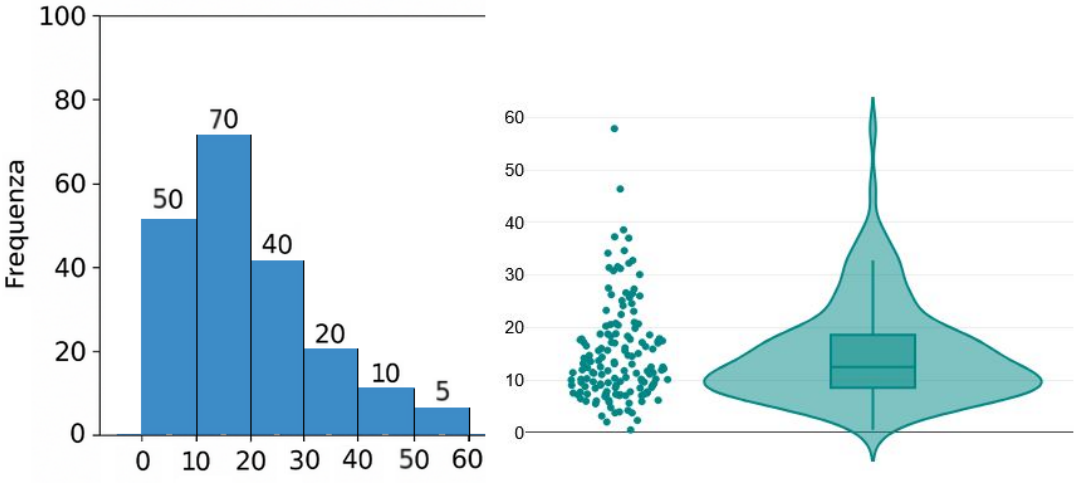


Figura 5.5: Difficoltà di Halstead (Istogramma e Violin Plot)

- **Media**: 14,85
- **Mediana**: 11,96
- **Minimo**: 0,5
- **Massimo**: 57,86
- **Osservazione**: Valori nella norma (10–100).

Sforzo di Halstead (log10)

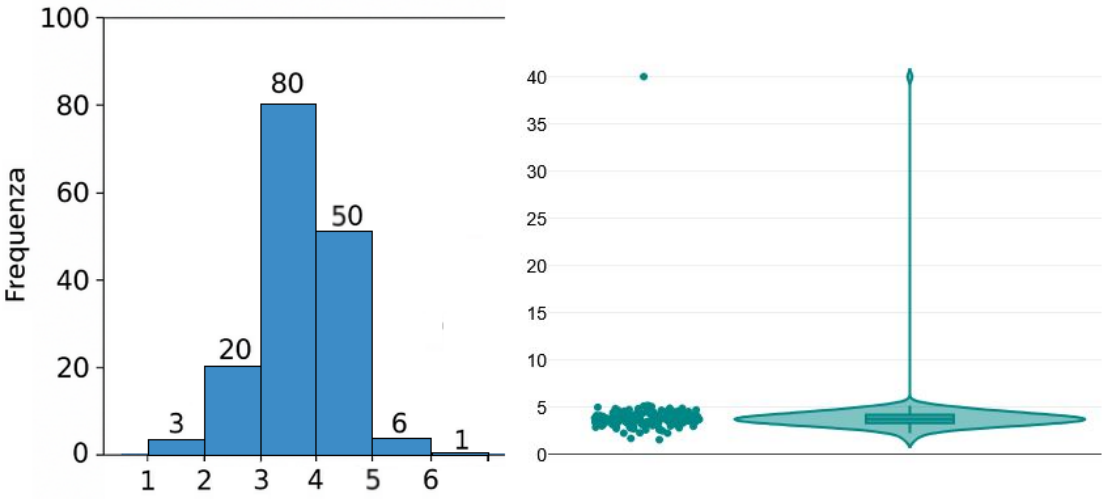


Figura 5.6: Distribuzione dello Sforzo di Halstead (log10) (Istogramma e Violin Plot)

- **Media**: 3,83
- **Mediana**: 3,71
- **Minimo**: 1,56
- **Massimo**: 40,05
- **Osservazione**: Valori generalmente nella norma.

Indice di Manutenibilità

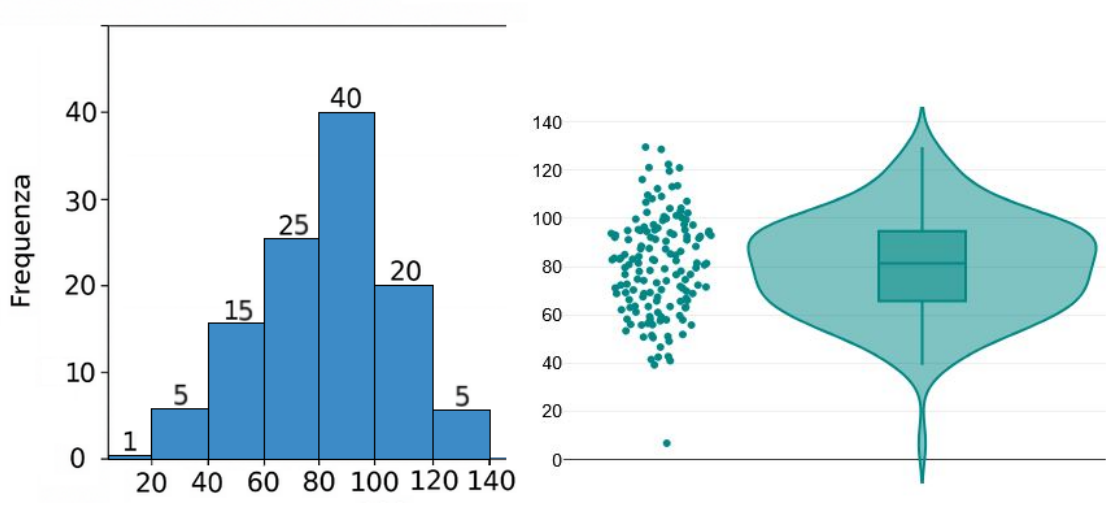


Figura 5.7: Indice Manutenibilità (Istogramma e Violin Plot)

- **Media**: 81,54
- **Mediana**: 83,13
- **Minimo**: 6,77
- **Massimo**: 129,62
- **Osservazione**: Eccellente manutenibilità (>20).

WMC Metodo Ponderato

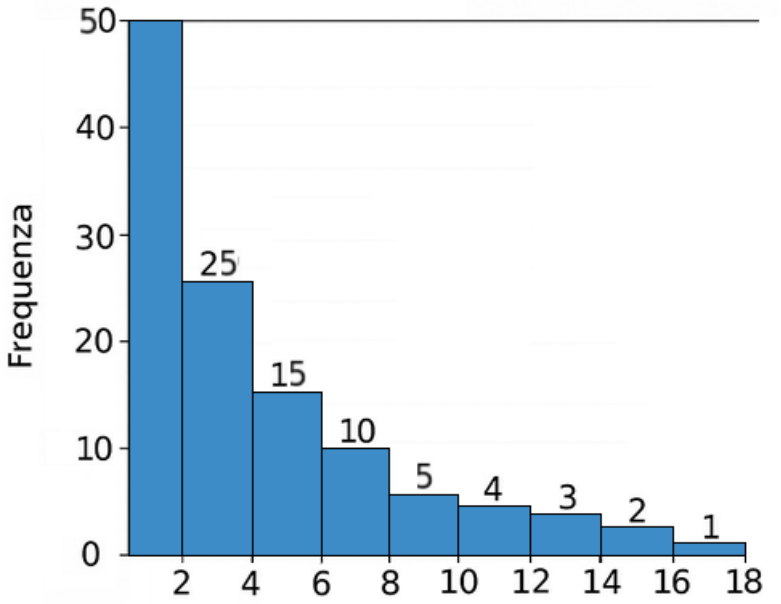


Figura 5.8: Istogramma WMC Ponderato

- **Media**: 3,84
- **Mediana**: 2
- **Minimo**: 0,1
- **Massimo**: 18
- **Osservazione**: Valori generalmente bassi, indicando bassa complessità.

5.4 Conclusioni sulla valutazione

5.4.1 Correttezza del codice

- Percentuale di test superati: avendo superato circa il 90% dei test l'LLM ha dimostrato di poter generare un codice corretto; questo risultato è ancora più significativo perchè molti degli esercizi presi erano di una complessità più alta rispetto al normale e anche si differenziavano molto l'uno dall'altro.

5.4.2 Qualità del codice

Complessità Ciclomatica

- Complessità Ciclomatica Sum: la maggior parte del codice prodotto ha ottenuto un risultato di bassa complessità ciclomatica; con una media di 2.84 e una mediana di 1 si può dire che questo sia un indicatore positivo sulla capacità dell'LLM di generare del codice con bassa complessità.

- Complessità Ciclomatica Average: Il fatto che abbiamo ottenuto una media di 3.31 e una mediana di 2.6, i quali sono valori leggermente superiori ai quelli ottimali (1-2), ci suggerisce che in alcuni casi, LLM fatica ancora a mantenere una bassa complessità - può essere sia stato dettato anche dalla complessità di alcuni di questi esercizi.

Complessità Cognitiva

- Complessità Cognitiva Sum: Otteniamo dei risultati simili alla complessità ciclomatica, anche qui la maggior parte del codice prodotto ha ottenuto un risultato di bassa complessità, cun una media di 2.47 e una mediana di 0 .Il valore massimo di 36, tuttavia, evidenzia che alcuni esercizi hanno generato codice con una logica difficile da seguire.

- Complessità Cognitiva Average: Anche la complessità media riflette una buona comprensibilità complessiva, con una media di 3,42 e una mediana di 3,5. La maggior parte dei valori si concentra su livelli bassi di complessità, ma il valore massimo di 36 conferma che alcuni esercizi hanno prodotto codice più impegnativo da interpretare.

Difficoltà di Halstead

Con una media di 14.85 e una mediana di 11.96, i valori rientrano nella norma (10-100) e indicano che il codice generato è generalmente di qualità e non presenta una propensione eccessiva agli errori. Il valore massimo di 57.86 suggerisce che alcuni esercizi hanno fatto produrre un codice più difficile da sviluppare ma senza errori.

Sforzo di Halstead

Otteniamo dei risultati simili alla difficoltà di Halstead con valori generalmente nella norma, con una media di 3.83 e una mediana di 3.71. Questo va ad indicare che il codice generato richiede uno sforzo moderato per essere compreso e implementato. Tuttavia, il valore massimo di 40.05 evidenzia che alcuni esercizi hanno prodotto codice più impegnativo.

Indice di manuteniblità

Con una media di 81.54 e una mediana di 83.13, l'LLM ha generato codice altamente manutenibile (>20). Questo è un risultato eccellente e riflette l'efficacia delle linee guida del prompt, come "Documenta ogni passaggio con commenti chiari" e "Segui le best practice di programmazione".

WMC (Metodo Ponderato per Classe)

Con una media di 3.84 e una mediana di 2, i valori sono generalmente bassi, indicando una bassa complessità combinata dei metodi locali di una classe. Questo riflette la capacità del prompt di guidare l'LLM verso classi ben progettate e con metodi semplici.

Deduzioni generali

- Efficacia del Prompt: il prompt utilizzato ha ottenuto una generazione del codice che ha portato ad ottimi risultati lato correttezza e lato qualità - le linee guida hanno avuto un impatto positivo nell'ottenere dei risultati soddisfacenti secondo le nostre metriche.
- Esercizi complessi: nonostante le medie e le mediane si attestino sui valori standard, ci sono tuttavia dei valori che vanno ben oltre i valori nelle norme, andando ad indicare che l'LLM fatica comunque per gli esercizi complessi, questi potrebbero beneficiare molto di più applicando un approccio iterativo.
- Applicabilità pratica: definire fin da subito un contesto con delle linee guida precise e mirate per garantire la correttezza e la qualità del codice può rivelarsi molto utile. Tuttavia, questa tecnica può essere sicuramente affinata ulteriormente adattandolo all'esercizio che vogliamo risolvere; inoltre, integrare dei prompt successivi per aiutare LLM a correggere o migliorare eventuali mancanze rappresenterebbe un passo efficace per ottenre risultati ancora più ottimali.

Questi risultati offrono delle linee guida utili per la progettazione di prompt migliori e per l'implementazione pratica degli LLM in progetti reali.

Capitolo 6

Discussioni Generali

In questo capitolo si discutono le osservazioni che sono emerse dalla valutazione sperimentale condotta, al fine di trarre conclusioni generali sull'efficacia degli LLM nella generazione di codice. La discussione è stata articolata secondo i criteri che abbiamo usato durante l'analisi: correttezza, qualità del codice, tecniche di prompting, risultati complessivi e applicabilità pratica.

Le valutazioni effettuate in questo studio ci hanno permesso di comprendere meglio le capacità degli LLM nella generazione di codice, le seguenti conclusioni generali si basano sui risultati ottenuti nei capitoli precedenti e sugli studi condotti.

6.1 Correttezza del Codice - Valutazione preliminare

Nella risoluzione dei sei esercizi gli LLM hanno dimostrato una ottima capacità di generare codice corretto, questo poi confermato anche dal superamento del 90% dei 150 test effettuati durante la valutazione conclusiva. Un risultato importante considerando la varietà degli esercizi: includevano sia problemi teorici che esercizi più complessi e meno comuni.

Per quanto riguarda i risultati dei singoli LLM nella valutazione preliminare della correttezza del codice generato:

- **ClaudeSonnet**: è stato il modello più robusto, performando in quasi tutti gli esercizi e generando codice corretto nella maggior parte dei casi, anche per problemi complessi.
- **GPT**: Ha mostrato una buona capacità di generazione codice corretto per la maggior parte degli esercizi, ma con risultati più variabili. Ha incontrato alcune difficoltà negli esercizi in RUST.
- **Gemini**: Ha avuto performance più deboli rispetto agli altri modelli, evidenziando difficoltà nella gestione di esercizi complessi e una tendenza a "bloccarsi" su errori ricorrenti.

Gli errori più comuni riscontrati sono:

- Python: principalmente errori sulla gestione dei tipi e sulle logiche del programma, si sono presentate funzioni del programma generato che davano i risultati sbagliati secondo le istruzioni che erano state date.

- Rust: il problema principale è stata la gestione del borrow checker, più volte le prime generazioni causavano un errore di compilazione. Un altro errore individuato è stato la gestione dei thread verificata in entrambi gli esercizi di RUST analizzati.

Principalmente, ClaudeSonnet ha dimostrato una grande capacità di generazione, producendo codice corretto anche negli esercizi di rust; mentre GPT, proprio su rust, non è riuscito a performare in maniera ottimale; questo potrebbe essere riconducibile alla scarsa familiarità con la sintassi del linguaggio o alla gestione della memoria esplicita tipica di RUST.

6.2 Qualità del Codice - Valutazione preliminare

La qualità del codice generato è stata valutata attraverso le metriche elencate nel capitolo *"Background"* attraverso lo strumento RustCode Analysis-

- **ClaudeSonnet**: risulta il modello più affidabile per la qualità del codice, ottenendo ottimi risultati nella maggior parte degli esercizi. Tuttavia, **GPT** ha superato Claude in alcune metriche, tra cui la manutenibilità, soprattutto negli esercizi più complessi.
- **Gemini**: Ha mostrato miglioramenti rispetto ai suoi codici iniziali, ma non è riuscito a raggiungere i livelli di qualità ottenuti dagli altri modelli.

6.3 Tecniche di Prompting

La scelta del prompt ha avuto un impatto significativo sulle performance degli LLM. Sono state testate diverse tecniche di prompting, tra cui:

- **Prompt unificato**: Sebbene efficiente per garantire la correttezza del codice, non si è rivelato ottimale per ottenere risultati qualitativi superiori rispetto alla tecnica iterativa.
- **Tecnica iterativa**: L'approccio in due fasi (prima correttezza, poi qualità) ha dato risultati migliori dal punto di vista delle metriche, soprattutto per esercizi complessi.
- **Chain of Thought (CoT)**: Il passaggio di dividere il problema in passaggi logici e sequenziali si è dimostrata una tecnica efficace per migliorare la correttezza del codice, soprattutto per Gemini e GPT.
- **Prompt correttivi**: L'aggiunta di dettagli sugli errori riscontrati ha migliorato significativamente la qualità del codice generato, evidenziando l'importanza di un approccio iterativo.
- **Prompt per il miglioramento del codice**: I prompt utilizzati per migliorare la qualità del codice hanno portato a risultati significativamente superiori rispetto al caso in cui tali istruzioni non fossero state fornite portando a una riduzione significativa della complessità ciclomatica e migliorando la leggibilità del codice.

6.4 Risultati del Test Conclusivo

L'analisi su 150 esercizi ha fornito una visione globale delle capacità degli LLM:

- **Correttezza**: il 90,44% dei test superati testimonia la grande capacità da parte dell'LLM di generare codice corretto, anche per esercizi complessi e variegati.
- **Qualità**: La maggior parte del codice prodotto che è stato generato ha ottenuto dei risultati soddisfacenti, con valori medi e mediani vicini agli standard ottimali. C'è da sottolineare però che l'LLM per alcuni esercizi complessi ha generato un codice con metriche più alte, evidenziando la necessità di un affinamento del prompt o di un approccio iterativo.
- **Applicabilità pratica**: la definizione iniziale di un contesto con linee guida specifiche e mirate si è rivelato un aspetto fondamentale. Tuttavia, l'integrazione iterativa di prompt successivi per correggere eventuali mancanze rappresenta un passo cruciale per ottenere risultati ancora più ottimali.

6.5 Applicabilità Pratica

Gli LLM hanno dimostrato di essere uno strumento valido per la generazione di codice in contesti variegati, ma la loro efficacia dipende dalla qualità del prompt e dall'approccio adottato:

- **Scenari in cui utilizzare un prompt unificato**: Sarebbe sempre da utilizzare all'inizio per introdurre subito un contesto mirato alle nostre specifiche e a quello che cerchiamo sulla generazione del codice.
- **Scenari in cui seguire un approccio iterativo**: per esercizi complessi o con requisiti non precisi, l'approccio iterativo consente di affinare il codice e migliorarnee la qualità attraverso prompt correttivi mirati sugli errori e sulle mancanze delle generazioni precedenti.
- **Limitazioni**: L'efficacia del prompting diminuisce per esercizi con una logica complessa o casi limite non previsti. Inoltre, l'uso di prompt complessi richiede maggiore preparazione e risorse.

La miglior tecnica è quella di introdurre fin da subito un prompt con linee guida mirate alle specifiche, e successivamente far sviluppare un codice migliore all'LLM introducendo dei prompt specifici atti a colmare le mancanze che vogliamo risolvere.

Capitolo 7

Conclusione

Il presente studio contribuisce alla letteratura esistente analizzando il potenziale degli LLM nella generazione di un codice corretto e di qualità. La seguente analisi sistematica èstata basata su criteri quali correttezza, qualità, tecniche di prompting e applicabilità pratica.

L'esito della ricerca dimostra che gli LLM sono in grado di affrontare varietà di esercizi di programmazione e offrono spunti utili per la progettazione di prompt superiori e per l'implementazione pratica degli LLM in progetti reali.

Con un miglioramento ed un affinamento delle tecniche di prompting, combinato ad un approccio iterativo, gli LLM possono diventare ottimi strumenti ancora più efficaci e affidabili per la generazione di un codice di qualità e adatto alle esigenze professionali.

I risultati ottenuti suggeriscono un'estensione di questa ricerca approfondendo aspetti quali: test su altri linguaggi di programmazione, utilizzo di esercizi più complessi, maggiore varietà di modelli e, soprattutto, ambienti collaborativi tra agenti LLM. Tali sviluppi potrebbero portare a strumenti sempre più sofisticati, capaci di supportare (o in alcuni casi sostituire) compiti specifici all'interno di un processo di sviluppo.

La generazione di codice tramite LLM non è più un semplice strumento di supporto, ma si configura sempre di più come una tecnologia concreta, efficace e destinata a trasformare profondamente il modo in cui si scrive software, soprattutto in ambito lavorativo. Tuttavia, affinché questa trasformazione sia efficace, è fondamentale progettare l'interazione tra LLM in modo consapevole, sfruttando a pieno le tecniche di prompting.

7.1 Lavori futuri

Sulla base delle analisi condotte, sono emerse numerose opportunità di approfondimento per estendere e raffinare lo studio. Possibili sviluppi includono:

7.1.1 Eseguire tutte le tecniche per ogni LLM, integrando anche con altri LLM

Un naturale sviluppo consiste nel completare la ricerca analizzando l'intero set delle tecniche su un insieme più ampio di LLM. In questa analisi ci si è concentrati maggiormente su GPT, ma un

confronto sistematico anche con Claude e Gemini permetterebbe una valutazione più completa e comparabile tra efficienza degli LLM e le strategie progettate. Questo approccio favorirebbe anche una maggiore robustezza dei risultati.

7.1.2 Valutazioni su altri linguaggi di programmazione

Principalmente in questo studio si è verificata l'efficacia degli LLM di generare codice tramite Python, con una sperimentazione preliminare su Rust. I test su Rust hanno già dato delle ottime riflessioni sulla reale efficacia degli LLM. Potrebbe essere utile estendere le valutazioni applicando le stesse tecniche ad altri linguaggi diffusi e rappresentativi di contesti diversi, come:

- JavaScript / React: utile per valutare la generazione in ambiti frontend e asincroni;
- SQL: per analizzare la capacità degli LLM in contesti dichiarativi e orientati ai dati;
- Java / C#: per confrontare i risultati in linguaggi più strutturati e verbosi.

Questo permetterebbe di verificare se l'efficacia delle tecniche di prompt è dipendente dal linguaggio target o se è generalizzabile.

7.1.3 Valutazioni su esercizi più complessi

Si è analizzato che gli esercizi più complessi sono quelli che hanno rappresentato una sfida più significativa per gli LLM e hanno messo in evidenza il contributo delle tecniche di prompting. Sarebbe molto interessante verificare queste stesse tecniche su esercizi più complessi poichè consentirebbe di testare l'efficacia delle tecniche in contesti più realistici.

7.1.4 Progettazione di un ambiente dove gli LLM lavorino insieme

Nella valtuazione si è analizzato come un modello genera un prompt per aiutarne un altro a generare un codice: questo concetto può essere ulteriormente sviluppato simulando un ambiente collaborativo tra LLM, ispirato all'organizzazione di un team di sviluppo software. In questo scenario, a ogni modello viene assegnato un ruolo specifico: Ad esempio, uno di questi potrebbe occuparsi della generazione dei prompt, un altro della scrittura del codice (il programmatore) e un altro ancora della revisione e del miglioramento della qualità del codice prodotto.

Questo approccio apre due direzioni di ricerca:

1. Verificare l'efficacia della collaborazione distribuita tra modello, misurando se l'assegnazione di ruoli migliora sistematicamente la qualità finale del codice.
2. Analizzare il punto di saturazione: fino a che livello l'introduzione di nuovi agenti apporta benefici, e quando invece la complessità della comunicazione tra LLM porta a una riduzione dell'efficienza – analogamente a quanto avviene nei team umani troppo estesi o mal coordinati.

Bibliografia

[1] Anthropic. *Introducing Claude 3.5 Sonnet*. Accessed: 07 May 2025. n.d. URL: https://www.anthropic.com/news/claude-3-5-sonnet.

[2] G. Ann Campbell. *Cognitive Complexity: A New Way of Measuring Understandability*. Rapp. tecn. SonarSource S.A., 2017. URL: https://www.sonarsource.com/docs/CognitiveComplexity.pdf.

[3] Shyam R. Chidamber e Chris F. Kemerer. «A Metrics Suite for Object Oriented Design». In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493.

[4] Codegrind. *Audit del codice e revisioni con Git*. Accessed: 15 Mar 2025. n.d. URL: https://www.codegrind.it/documentazione/git/audit-codice-revisioni.

[5] Riccardo Coppola. *Large Language Models*. Presentazione del corso. 2024.

[6] DAIR.AI. *Guida al Prompt Engineering*. URL: https://www.promptingguide.ai/it.

[7] Documentazione interna. *Metriche prese dal progetto Kotlin_JSS*. Fornito dal docente del corso, non disponibile pubblicamente. 2025.

[8] Exercism. *Python Track Exercises*. Accessed: 17 Mar 2025. n.d. URL: https://exercism.org/tracks/python/exercises.

[9] Sofia Ferrante. *Prompt engineering: cos'è e come ottimizzare interazione con AI*. Accessed: 2025-05-24. 2025. URL: https://www.ai4business.it/intelligenza-artificiale/prompt-engineering-cosa-e-come-ottimizzare-interazione-con-ai/.

[10] Google. *Code Execution | Generative AI on Vertex AI*. Accessed: 07 May 2025. n.d. URL: https://cloud.google.com/vertex-ai/generative-ai/docs/multimodal/code-execution.

[11] HackerRank. *Programming Challenges*. Accessed: 17 Mar 2025. n.d. URL: https://www.hackerrank.com/challenges/.

[12] Maurice H. Halstead. «Elements of Software Science». In: *Elsevier* (1977).

[13] Hugging Face e Google Research. *Google Research Datasets*. Accessed: 17 Mar 2025. n.d. URL: https://huggingface.co/google-research-datasets/mbpp.

[14] Juyong Jiang et al. «A Survey on Large Language Models for Code Generation». In: *arXiv preprint arXiv:2406.00515* (2024). Version 2, last revised 10 Nov 2024. URL: https://arxiv.org/abs/2406.00515.

[15] Pengfei Liu et al. «Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing». In: (2021).

[16] Thomas J. McCabe. «A Complexity Measure». In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320.

[17] Mozilla. *rust-code-analysis*. `https://github.com/mozilla/rust-code-analysis`. Accessed: 2025-05-24. 2024.

[18] OpenAI. *Introducing ChatGPT*. Accessed: 07 May 2025. n.d. URL: `https://openai.com/index/chatgpt/`.

[19] promptingguide.ai. *"Techniques." Prompting Guide*. URL: `https://www.promptingguide.ai/techniques`.

[20] Katsiaryna Ruksha. *Prompt Engineering: Classification of Techniques and Prompt Tuning*. URL: `https://medium.com/the-modern-scientist/prompt-engineering-classification-of-techniques-and-prompt-tuning-6d4247b9b64c`.

[21] Wei Tao et al. «MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue ReSolution». In: (2024).

[22] Tree-sitter. *Tree-sitter: a parser generator tool and an incremental parsing library*. `https://tree-sitter.github.io/tree-sitter/`. Accessed: 2025-05-24. 2024.

[23] Nalin Wadhwa et al. «CORE: Resolving Code Quality Issues using LLMs». In: (2024).

[24] Jason Wei et al. «Chain-of-Thought Prompting Elicits Reasoning in Large Language Models». In: (2022).

[25] Jules White et al. «A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT». In: (2023).

[26] Jules White et al. «ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design». In: (2024).

[27] Shunyu Yao et al. «Tree of Thoughts: Deliberate Problem Solving with Large Language Models». In: (2023).