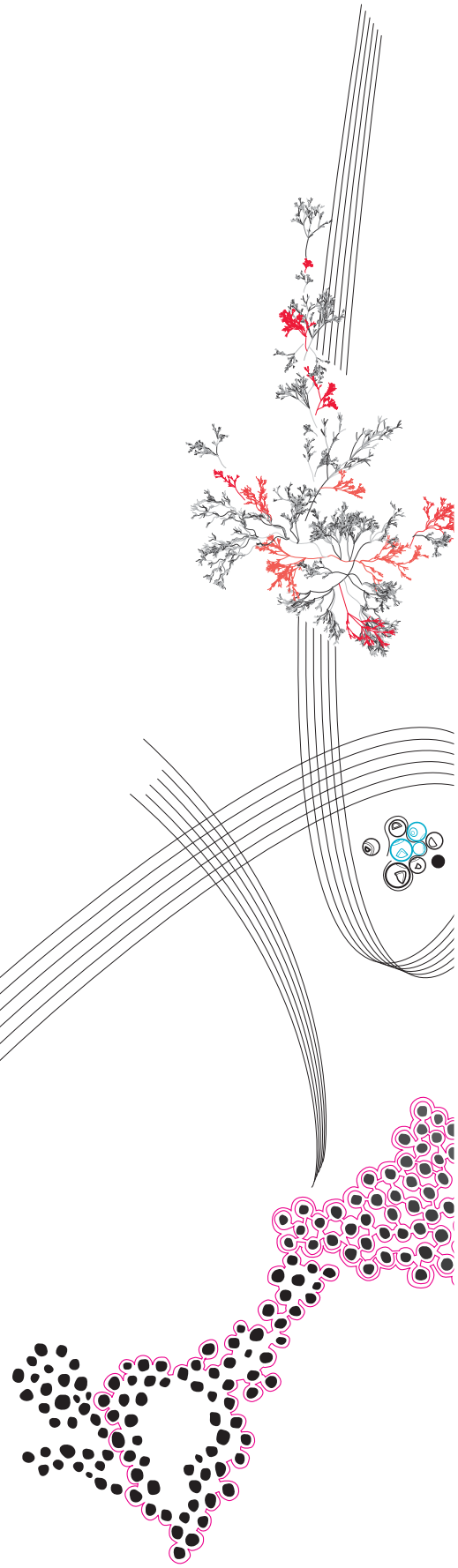

MSc Computer Science
Final Project

Benchmarking the Programming Capabilities of Large Language Models

Faisal Nizamudeen

Supervisor: Vadim Zaytsev

July, 2025

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Acknowledgements **5**

1 Introduction **1**

2 Existing Work **4**
2.1 Transformer Architecture . . . 4
2.2 Models . . . 5
2.2.1 OpenAI . . . 5
2.2.2 Google . . . 5
2.2.3 Anthropic . . . 6
2.3 Programming Capabilities of AI models . . . 7
2.4 Limitations of AI models . . . 8
2.5 Programming in the Small vs Programming in the Large . . . 9
2.6 Conclusion . . . 9

3 Small Problems - Methodology **10**
3.1 Problem Set . . . 10
3.1.1 Easy, Medium and Hard Questions . . . 12
3.1.2 Questions by Topic . . . 15
3.2 Models Tested . . . 15
3.2.1 Overview of Evaluated Models . . . 15
3.2.2 OpenRouter . . . 16
3.3 Prompting and Technical Setup . . . 17
3.3.1 Prompting Strategy . . . 17
3.3.2 Technical Specifications . . . 18
3.4 Evaluation Metrics . . . 19
3.5 Technical Implementation . . . 19
3.5.1 Database . . . 19
3.5.2 Selenium Automations . . . 21
3.5.3 Radon Metrics . . . 22
3.5.4 Pipeline . . . 22
3.6 Conclusion . . . 23

4 Small Problems - Results **24**
4.1 Model Accuracy by Difficulty . . . 24
4.1.1 Easy Questions . . . 24
4.1.2 Medium Questions . . . 25
4.1.3 Hard Questions . . . 26
4.1.4 Overall . . . 26

4.2 Statistical Significance of Model Accuracy . . . 27
4.2.1 McNemar's test . . . 27
4.2.2 Sample Calculation . . . 28
4.2.3 McNemar Values . . . 29
4.3 Evaluation Metrics . . . 30
4.3.1 Wilcoxon Signed-Rank Test . . . 31
4.3.2 Cyclomatic Complexity . . . 31
4.3.3 Maintainability Index . . . 32
4.3.4 Source Lines of Code . . . 33
4.4 Token Usage . . . 33
4.5 Conclusion . . . 34

5 Small Problems - Discussion **35**
5.1 OpenAI o1 . . . 35
5.2 GPT-4o . . . 37
5.3 Gemini 2.0 . . . 38
5.4 Gemini 2.5 . . . 39
5.5 Claude 3.7 . . . 41
5.6 Individual Problems Analysis . . . 43
5.7 Model Architectures . . . 58
5.8 Conclusion . . . 58

6 Large Problems - Methodology **60**
6.1 Problem Identification . . . 60
6.1.1 Landing Page – Frontend . . . 60
6.1.2 Bug Tracker – Backend . . . 61
6.1.3 ToDo List – Full Stack . . . 62
6.1.4 Flappy Bird - Game . . . 63
6.1.5 Pomodoro Tracker – CLI Application . . . 64
6.1.6 Diversity of DataSet . . . 65
6.2 Evaluation Metrics . . . 65
6.3 Cursor IDE & Models Tested . . . 66
6.4 Conclusion . . . 67

7 Large Problems – Results **68**
7.1 CLI Application . . . 68
7.1.1 OpenAI o4 . . . 68
7.1.2 Claude 3.7 Sonnet . . . 69
7.1.3 Gemini 2.5 . . . 70
7.1.4 Model Comparison . . . 71
7.2 Frontend – Flappy Bird Game . . . 71
7.2.1 OpenAI o4 . . . 71
7.2.2 Claude 3.7 Sonnet . . . 73
7.2.3 Gemini 2.5 . . . 75
7.2.4 Model Comparison . . . 77
7.3 React Front-End . . . 77
7.3.1 OpenAI o4 . . . 77
7.3.2 Claude 3.7 Sonnet . . . 78
7.3.3 Gemini 2.5 . . . 80
7.3.4 Model Comparison . . . 82

7.4 Spring Boot Backend . . . 82
7.4.1 OpenAI o4 . . . 82
7.4.2 Claude 3.7 Sonnet . . . 83
7.4.3 Gemini 2.5 . . . 83
7.4.4 Model Comparison . . . 84
7.5 Full-Stack . . . 84
7.5.1 OpenAI o4 . . . 84
7.5.2 Claude 3.7 Sonnet . . . 85
7.5.3 Gemini 2.5 . . . 87
7.5.4 Model Comparison . . . 89
7.6 Conclusion . . . 89

8 Large Problems - Discussions **90**
8.1 Claude 3.7 Sonnet . . . 90
8.2 OpenAI o4 . . . 91
8.3 Gemini 2.5 . . . 92
8.4 Model Architectures . . . 92
8.5 Conclusion . . . 93

9 Future Work **95**
9.1 Threats to Validity . . . 95
9.2 Possible Advancements . . . 96
9.2.1 Increase Problem Set Diversity . . . 96
9.2.2 More Complex Architectures . . . 96
9.2.3 Human Comparison . . . 96
9.2.4 Model Coverage . . . 97
9.2.5 Test Generation . . . 97
9.2.6 Debugging and Error Correction . . . 97
9.2.7 Impact on Education . . . 98
9.3 Conclusion . . . 98

10 Conclusion **99**

Declarations **108**

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Vadim Zaytsev, for his invaluable support and guidance throughout the process of writing this thesis. Our bi-weekly meetings were a consistent source of insight and encouragement. He provided a careful balance between offering direction and allowing me the freedom to explore ideas independently. The opportunity to pursue a research direction that genuinely interested me was something I truly appreciated, and I remain sincerely thankful for that trust.

I am also grateful to my examiner, Nacir Bouali, for his thoughtful and constructive feedback during the final stages of the project. His perspective brought a fresh and holistic view to the work, and his comments helped refine both the content and structure of the final version.

Finally, I want to thank my family. My parents and my brother have been a constant source of support throughout this period. Their encouragement played a key role in helping me stay focused and complete this thesis in a timely and steady manner.

Abstract

This thesis benchmarks the coding abilities of the most widely used large language models (LLMs), particularly the flagship offerings from OpenAI, Google, and Anthropic. The experiments were separated into two sections, small problems and large problems. There were a total of 75 LeetCode problems to determine small problem performance. Models were assessed based on accuracy and code quality, with measures such as maintainability index, source lines of code, and cyclomatic complexity as indicators of the code quality produced. The results found that OpenAI's o1-mini had the best accuracy, while Claude 3.7 Sonnet produced the highest quality code overall. GPT-4o-mini performed significantly worse than the other models. For the large problems, five unique tasks were chosen across various programming languages and project types. The models tested in these experiments were OpenAI o4-mini, Claude 3.7 Sonnet, and Gemini 2.5 Pro. Each solution was assessed based on functional correctness, maintainability, and ease of prompting the model. All of the experiments were done in an agentic manner using the Cursor IDE. In these experiments, Claude 3.7 Sonnet had the best overall scores for all three metrics. OpenAI o4-mini came in as the second-best model, with Gemini 2.5 Pro showcasing the worst average performance across all models tested. While these results are encouraging, there are ample opportunities for growth in future research. Future research areas include testing other models from other providers, sampling a larger variety of problems from other sources, and comparing LLM-generated code to human-generated code. Other aspects of software development, such as test generation and debugging can also be explored. While LLMs are far from perfect, this paper shows that with the right prompting and human guidance, they can already serve as a helpful tool for both small and large programming tasks.

Keywords: Large Language Model, Software Development, GPT, Gemini, Claude, OpenAI, Google, Anthropic, LeetCode

Chapter 1

Introduction

In recent years, we have seen an innovation boom in the field of AI. One of the most valuable innovations that have come about from this is the advancements in LLM models. These models have been trained on large data sets and can simulate human like conversations in a wide range of contexts. They are accessible through many mediums such as Chatbots, APIs and integrated into everyday software like IDEs. These models offer a way for users to ask any question they may have, as long as it is representable in text and receive a response on a nearly limitless number of topics. Recent models have advanced to a state where they support multi-modal input. In these models, input can now be provided in other forms, such as images and audio, and complex file types, such as .zip and .txt.

Three of the most widely used LLM offerings are OpenAI's GPT, Google's Gemini and Anthrophics Claude [59]. OpenAI's ChatGPT was the first major player in this game and was launched in November 2022. It amassed massive popularity, being the first of its kind, reaching over 100 million active users shortly after its public release through its ChatGPT chatbot and becoming the fastest-growing software in history [77]. Its responses, for the most part, were indistinguishable from those of regular human speech.

Google's Bard, later re-branded to Gemini, was released in March of 2023, a mere 3 months after the release of ChatGPT [26]. While it did not possess the early wave of adoption that ChatGPT did, it managed to claim its unique set of use cases. For one, Google focused on allowing multi-modal input to its models from the beginning, allowing Gemini to receive feedback via images and audio [5]. While this feature was eventually added to ChatGPT, it came through much later and is still only available with some of the more advanced models on the site. Gemini also focused on integrating its chatbots with web technologies, which allowed for better data understanding and processing in many cases [94, 19].

Anthropic's Claude was released in March 2023, entering the market shortly after ChatGPT and around the same time as Google's Gemini[37]. While it did not attract the same initial attention, Claude gradually gained recognition for its focus on safety and interpretability. In March 2024, Anthropic introduced the Claude 3 family, comprising Haiku, Sonnet, and Opus, each designed to meet distinct performance needs [38].

This paper focuses on these three models due to their widespread adoption and influence in the AI landscape [59]. Their popularity can be attributed to a combination of factors, including their feature sets, release timing, and strong brand recognition. As of May 2025, ChatGPT maintains its position as the market leader in generative AI chatbots, with its growth stabilising as competitors like Google Gemini enhance their AI assistants to close the gap. Google has set an ambitious goal for Gemini, aiming to reach 500 million users by the end of 2025 [28]. Claude while lagging behind in overall users, is the fastest growing

model, gaining an impressive 14% increase of total users over the first quarter of 2025 [59].

The functionality of these AI models hinges on their ability to understand and interpret human queries in natural language. They employ several NLP (Natural Language Processing) techniques to analyse inputs and generate appropriate responses. All these LLMS employ the use of transformer algorithms for this task [77]. These technologies allow these models to maintain relevant context, providing a more seamless user experience.

One of LLMs' most common use cases is related to programming. These LLMs have numerous functionalities in this domain and can assist with code generation, debugging, and error detection. The most common way users interact with these LLMs in relation to programming is through their dedicated online interface. However, in recent times, many programmers have found different ways. There are CLI tools like Cline, plugins for popular IDEs such as GitHub Copilot and Qodo, and even entire IDEs that are built upon forks of existing software, such as Cursor [12, 13, 21, 10]. While each approach has its pros and cons, the general consensus is that this has greatly eased the use case because the LLMs can directly get context, and programmers do not need to be constantly feeding it through an AI chatbot. It streamlines the user experience and makes it easier to use in this context.

Several of these implementations also have agentic capabilities. This allows the model to not just reply with code fragments but take control of the user's machine to generate or edit entire files and run terminal commands. This gives a very hands-off approach to the programmer. In recent times, this form of programming has been coined as "Vibe Coding". This was a term first mentioned by Andrej Karpathy in February 2025. Vibe coding essentially involves expressing one's intention to an IDE model using natural speech and leaving the rest to the model [62].

Code generation is perhaps the most useful application of AI in programming. LLMs can significantly streamline the process of writing boilerplate code, especially in highly verbose languages or frameworks, saving the programmer time. These LLMs can also generate complex programs when provided with more complete requirements and specifications. The effectiveness of these AI models in code generation has been quantified using various benchmarks. The results show that these models excel in specific programming tasks, particularly in generating solutions to smaller programming problems [69].

Debugging is another useful capability of these tools. It is a critical aspect of software development and something that developers often spend a lot of time on. By leveraging NLP (Nature Language Processing) and its extensive knowledge base containing many error messages, these AI models can often accurately diagnose and provide solutions for code that does not work as expected. This use case of chatbots has dramatically diminished the need for sites like Stack Overflow, as it allows for a more specific and tailored experience. This capability saves developers time and allows them to focus on more complex tasks that arise during programming [84].

Apart from debugging, these tools are also capable of error detection. They can analyse code for specific pitfalls and vulnerabilities and notify the developer before they escalate to major issues. With the increased adoption of coding plugins and AI IDEs, these models can provide continuous feedback to developers seamlessly, ensuring they always adhere to the best practices and coding standards [87].

This paper serves as a starting point for researching the programming capabilities of AI chatbots and comparing the differences between the most popular models. It will focus on code generation and test the ChatGPT and Google Gemini models. Other parts of the programming process, such as error detection and debugging, will also be discussed, but merely in the form of literature reviews. No explicit experiments were conducted to test this functionality.

The goal of this paper is to investigate the following two primary research questions:

- **RQ1**: Which LLM provider delivers the most effective model for solving small scale programming exercises?
- **RQ2:** Which LLM provider delivers the most effective model for solving large scale programming exercises?

It is widely accepted that programming in these two ways can be considered fundamentally different, and they are treated as distinct categories in both research and practice [72].

The methodological framework in this thesis centres on systematically benchmarking the programming performance of Large Language Models (LLMs). This involves the careful selection of tasks, the definition of clear evaluation metrics, and the use of a structured testing process that enables reproducibility and fair comparison. The tasks span a diverse range of computational challenges, including both smaller algorithmic problems and broader software development scenarios. Each task has been designed to ensure that model performance can be meaningfully compared and that findings can be generalised across different LLMs.

The remainder of this paper is structured as follows. Chapter 2 will delve into the existing literature to discuss the architecture of these tools, the design decisions between the different models and their capabilities and limitations. Chapter 3 will discuss the methodology used to benchmark the tools on small problems. The models tested, the problem set that has been used, and the evaluation metrics will all be discussed in this chapter. Chapter 4 will go over the concrete results that were obtained from the experiments and perform statistically significant tests to prove if a model is superior to another on a specific metric. Chapter 5 will discuss the results that were obtained in the previous chapters and provide recommendations for the suitability of the usage of the tools. Similarly, Chapter 6, Chapter 7, and Chapter 8 will go over the methodology, results and discussion for large programming problems. Chapter 9 will acknowledge the limitations of the research carried out and provide possible future work. Lastly, Chapter 10 serves as a conclusion and will summarise the key findings from this paper.

Chapter 2

Existing Work

This chapter will discuss the necessary background related to LLMs. I will begin by outlining the Transformer architecture that underpins all of these models. The evolution and reasoning behind the models developed by OpenAI, Google, and Anthropic will be explored, along with their respective design choices. The capabilities and limitations of these tools will also be examined, as well as the motivation for treating small and large programming problems as fundamentally different tasks.

2.1 Transformer Architecture

At the core of the GPT and Gemini lies the Transformer architecture. The Transformer is a neural network that relies on self-attention mechanisms to handle long-range dependencies between texts efficiently [36, 90].

The architecture typically consists of two main components, an encoder and a decoder. The primary function of the encoder is to process input tokens and transform them into continuous representations. These representations maintain contextual information and the relationship between the input tokens. Each encoder layer comprises two sublayers, the self-attention mechanism and the feedforward neural network. The self-attention mechanism allows the model to accurately weigh out the significance of each token relative to other tokens in the input string. This allows it to accurately capture dependencies regardless of the distance between them in the text. The neural network then processes weighted representations to produce the encoder output [36].

The purpose of the decoder is to generate output sequences from the encoder representations. The decoder consists of three layers. These layers are a masked self-attention mechanism, an encoder-decoder attention mechanism, and a corresponding feedforward neural network. The self-attention mechanism aims to ensure that the prediction for a particular position corresponds only to all outputs till that point. Meanwhile, the encoder-decoder mechanism works by allowing the decoder to focus on the relevant tokens present on the input by reading the representation created by the encoder. This facilitates cohesive and relevant responses to the end user [36].

The models used by LLMs differ slightly from traditional transformer implementation. Instead of an encoder decoder implementation, they employ a decoder-only architecture [57]. The key reason for this is related to these models' ability to perform generative language modelling, which involves predicting subsequent tokens in a sequence based on the previous inputs and tokens. By utilizing only the decoder component, these models have learnt to generate contextual and coherent responses without requiring an explicit encoder to process the input sequence. This design choice dramatically simplifies the model

complexity and leverages the decoder's ability to model dependencies in human text [63].

2.2 Models

2.2.1 OpenAI

Generative Pre-trained Transformer(GPT) models developed by OpenAI have significantly advanced natural language processing (NLP) by enabling these models to understand and generate human-like text [80]. The term GPT refers to the foundational aspects of this model. "Generative" showcases the models' ability to produce text. "Pre-trained" signifies that the model has been trained extensively using training data and fine-tuned for specific tasks. "Transformer" refers to the underlying neural network architecture. [61].

The capabilities of ChatGPT, particularly its GPT-3.5 and GPT-4 models, have been largely explored by the community. GPT-3.5 was the initial version of GPT that was bundled and released in December 2022 [34]. GPT-4 is a newer, more advanced model released in 2023. While it was initially locked behind a premium subscription, free users can now access this newer model for a set number of queries in a specific period, making it more accessible. GPT-4 is also the first GPT model to support multimodal input. GPT 3.5 was an unimodal engine, and prompts could only be supplied via text. GPT-4 exhibits notable improvement when compared to GPT-3 when tested across various professional and academic benchmarks [29, 30].

Starting from September 2025, OpenAI started releasing its 'o' range of models. OpenAI o1-mini was the first to release in September 2024, followed by OpenAI o3-mini in January 2025 and OpenAI o4-mini in April 2025. Less cost-effective OpenAI o1 and OpenAI o3 models were also released and can be considered the larger pro version of the models. According to OpenAI, these models greatly outperform the older GPT-4 models when benchmarked on a variety of tasks. The tasks benchmarked include programming tasks like problems on programming platforms such as CodeForces as well as software engineer tasks like SWE-bench [11, 55]. These models overall boast improved response times, accuracy, and a deeper understanding of the prompts supplied. The o3 model also provides more flexibility to the end user, allowing them to pick between 3 reasoning efforts, high, medium, and low, that impact how the prompts are processed [44, 45, 42].

2.2.2 Google

Gemini, developed by Google's DeepMind division, incorporates a similar decoder-only transformer implementation [23]. Like GPT, it has been pre-trained using extensive amounts of sample data. Its architecture incorporates the Language Model for Dialogue Applications(LaMDA) and the Pathways Language Module (PaLM) [22, 43]. This allows it to leverage extensive datasets and advanced optimisation techniques when generating output for users [3].

In contrast to GPT, Gemini models are designed to be natively multimodal. This allows them to better process and reason over text, images, audio and code within a single, unified architecture. One key innovation in the structure of the GPT model is the presence of cross-modal attention mechanisms. These allow the models to better jointly process data provided in multiple forms and make relationships between them. This provides Gemini with much stronger reasoning abilities and better contextual understanding across diverse modalities [19, 4].

Like GPT, there have been numerous iterations of the Gemini Model. The first iteration of the Gemini model was Bard, which was released in March 2023. It was trained on the

Infiniset dataset that contained approximately 1.56 trillion words [3]. In December 2023, Bard was rebranded as Gemini and became integrated with the Gemini Pro model. The Gemini Pro model greatly improved the multimodal capabilities of the model. It also added enhanced content moderation to reduce incorrect, harmful or biased content. Some new features were also added, such as the ability to generate images using Google Imagen [24].

This was followed by Gemini Ultra 1.0, released in February 2024. It improved upon the reasoning abilities of the older model for better output generation and performed better on a variety of benchmarked tasks, including programming. It also possessed better collaboration capabilities that allow it to engage with the user in a more natural way, tailoring its interaction style to the user's input [20, 41]. Gemini 1.5 was released 3 months later, in May 2024, and was an update that primarily focused on improving the speed and efficiency of the Gemini models. It surpassed Gemini 1.0 in 87% of the benchmarked tests and optimized the model performance for low-latency tasks, enhancing the user experience in real-time applications [40].

The newest models, Gemini 2.0 and Gemini 2.5 were released in December 2024 and March 2025, respectively, and marked a significant step forward in the capabilities of the Gemini model. It featured further improved multimodal capabilities, making it the go-to model for audio and video analysis. It greatly outperformed the older models across various benchmarked tasks. This Gemini model was also capable of interacting with the user beyond the initial chatbot interface. It could now directly interact with a user's computer or the web to complete certain tasks and provide feedback. Google also released AI agents that could better assist users with certain tasks [25, 27, 16].

2.2.3 Anthropic

Anthropic introduced its first AI model, Claude 1, in March 2023. This initial version emphasised safety and interpretability, setting the foundation for subsequent iterations [37]. Claude 2 followed in July 2023, offering enhanced performance and a larger context window, allowing for more extended and coherent responses [6].

In March 2024, Anthropic released the Claude 3 family, comprising three models: Haiku, Sonnet, and Opus. Each model catered to different user needs, balancing speed, cost, and performance. Notably, these models introduced advanced vision capabilities, enabling them to process and reason over other visual formats, including photos, charts, and technical diagrams [38].

June 2024 saw the launch of Claude 3.5 Sonnet, which outperformed its predecessor, Claude 3 Opus, in several benchmarks. This version demonstrated significant improvements in graduate-level reasoning, undergraduate-level knowledge, and coding proficiency. It also introduced features like "Artifacts," allowing users to interact with generated content in a more dynamic way [37].

In February 2025, Anthropic unveiled Claude 3.7 Sonnet, a hybrid reasoning model designed to handle both rapid responses and complex problem-solving tasks. This model introduced an "extended thinking" mode, allowing for deeper reflection before answering, which improved performance in areas like math, physics, and coding. This version of Claude is often regarded as being the best agentic model. Many AI IDEs like Cursor come with Claude 3.7 set as the default selected model and as the recommended option for most programming cases [7].

2.3 Programming Capabilities of AI models

These models exhibit varying degrees of accuracy and quality in code generation, providing a different experience for the end user. Research indicates that ChatGPT has achieved notable success in task-oriented dialogue systems, performing comparably to traditional rule-based systems regarding task success and language understanding [60]. This functionality extends to programming tasks, where GPT-4 has been shown to generate accurate code for various problems. For example, a comparative analysis of various large language models showed that GPT-4 outperformed its predecessor, GPT-3.5, in a wide range of programming exercise samples [69]. This finding demonstrates the advancements made in the latest accessible iteration of the model, which should be able to better understand and respond to programming prompts.

In a quantitative assessment, using the Mostly Basic Python Problems (MBPP), GPT-4 demonstrated superior performance compared to the GPT-3 model and competing models such as Google Gemini and Anthropics Claude. The results indicated that GPT-4 generated more accurate code and improved problem-solving efficiency [69]. This aligns with findings from another study, which state that while GPT-3.5 generated code correctly for approximately 22.2% of problems, GPT-4 achieved a higher success rate of 38.9% showcasing its improved accuracy [75].

The quality of code generated by GPT is another area of research interest. GPT-4 has been known for its ability to tackle more difficult programming exercises, specifically those that are considered hard [75]. The results suggest that the model advancements have equipped GPT-4 with a better understanding of programming logic and syntax, resulting in an overall better output.

Tests have also been done to measure the ability of GPT-4 to provide relevant error messages and debugging assistance. Research has shown that GPT-4 can greatly enhance the clarity of programming error messages, providing important context to the users. This, in turn, helps reduce the frequency of repeated errors among users [76].

The evolution of the GPT models and how they will continue to evolve has also been examined. An analysis of the updates made to the GPT-4 engine from GPT-3.5 revealed that improvements were noticed in the performance of a plethora of everyday tasks. These tasks, including code generation and debugging, have shown consistent improvement [68]. This ongoing refinement indicates a commitment from the side of OpenAI to improve the model's capabilities continuously. In particular, the ability of these models to adapt, learn, and pick up on new programming languages and paradigms has been highlighted, indicating a wide array of use cases across unique programming domains [88].

While the amount of research done on the Gemini model is comparatively less compared to that of GPT, recent studies have begun to evaluate the capabilities and accuracy of the Gemini model, compared explicitly to OpenAI's GPT-4 model. According to some, while Gemini 1.5 Pro demonstrates strong performance across various language abilities, it slightly underperforms GPT-3.5 Turbo in most benchmarked tasks, such as programming [64]. Another study looked at the capabilities of AI tools in solving problems on the most widely used programming platforms, such as HackerRank and LeetCode [66]. It was found that Gemini was slightly outperformed by GPT-4 on all tested platforms. A similar story was reported when the models were compared using the MBPP problem set [69]. In this paper, they found that Google Bard, as it was known at the time, solved fewer exercises correctly when compared to GPT-4 and GPT-3.5.

2.4 Limitations of AI models

Despite these tools' promising capabilities, using generative AI for programming presents several challenges.

One of the primary limitations of these AI models is their propensity for generating inaccurate or non-functional code. A study conducted evaluated the capabilities of the GPT-3.5 model across ten programming languages and 4 software domains. The study showed that while the model was able to correctly generate accurate and syntactically correct code, it often exhibited unexpected behaviour when tackling complex problems. For example, when a problem required specific domain knowledge or intricate logic to be solved, the model returned code that was inefficient and failed to meet the problem constraints. This necessitates significant human intervention to rectify these issues [67]. Another study, which focused on using these models to write NONMEM code, another highly specialised and domain-specific language, found that the code generated by GPT and Gemini often failed to adhere to the best practices of the language and returned inaccurate code. The authors suggested that an expert needs to sample and verify the code before it can be considered for practical use [85].

Research also suggests that while the GPT 3.5 has a reasonable success rate of 71% for LeetCode problems when an incorrect solution is returned, the model showcases difficulty in refining its solution based on feedback from these platforms. The findings show that when the initial attempt to solve a problem is not successful, the model is rarely able to correct its solutions [83].

The issues of inaccurate code are often caused by these models' tendency to "hallucinate," where they generate plausible but ultimately incorrect information. This leads to pitfalls in programming assignments where precision may be crucial [89]. These hallucinations arise from several factors in their design and training process. The most common reason for this is the presence of bias in the training data. If this data contains biases or inaccuracies or is not sufficiently diverse, the model will provide output that reflects these flaws. Furthermore, the probabilistic nature of these models means that they often try to predict the next word or code segment based on its learnt patterns. This can lead to confident but erroneous responses. This issue is especially noticeable in the case of code generation, where the model might fabricate functions or improperly handle imports/libraries [81].

Recent academic studies have also identified significant security vulnerabilities in code generated by these AI models. One study introduced the FormAI-v2 dataset, which consisted of 265,000 C programs generated by AI. The results showed that 63.74% of these programs contained vulnerabilities [86]. Another study analysed Python code using a custom-made tool known as DeVAIC. The tool showed that AI-generated Python code often produces code susceptible to common weaknesses, as classified by the OWASP Top 10 [71, 47].

The code generated by these AI models is also not indistinguishable from that of human-written code. One study compared the code generated by ChatGPT to human-written code using a dataset of 131 prompts. The findings show that the code generated by ChatGPT often lacks comprehensibility and security. Machine learning models were able to distinguish AI-generated code from human code with up to 88% accuracy, suggesting that AI-generated code may have certain identifiable patterns that differ from human coding practices [73].

Until the introduction of advanced IDEs like Cursor, the lack of real-time debugging capabilities in both tools, unless explicitly requested, meant that users had to manually

verify and test the generated code, which could negate some of the efficiency gains these tools aimed to provide. These models have also been criticised for their inadequate error-handling capabilities. For example, while they can generate meaningful code snippets, they often fail to provide insight into the debugging process or to suggest fixes for errors a user may face [74, 78]. This deficiency may lead to frustration among programmers who expect an all-in-one solution that can help more than just writing out the initial starting point of code.

2.5 Programming in the Small vs Programming in the Large

An effort has been made to explicitly assess the capabilities of these AI models on small and large problems separately. This has been done as it is fundamentally accepted that these are two different activities. This idea was first brought up in the 1970s by DeRemer and Kron [72].

They explain that the distinction between programming-in-the-small and programming-in-the-large reflects a difference not only in scale but also in cognitive approach. Programming-in-the-small refers to the activity of writing individual modules that are concise, self-contained, and typically manageable by a single developer. These modules are implemented using conventional programming languages and focus on localised computational tasks. While effective for constructing isolated components, such problems offer limited support for expressing relationships across a wider system.

Programming-in-the-large instead focuses on the relationships between many modules that form a cohesive software system. Here, the difficulty lies not in writing individual components, but in organising how they interact. Developers must decide how to group related functionality, how to manage resource visibility, and how to ensure that changes in one part of the system do not inadvertently break others. Without explicit mechanisms to define these relationships, information about module dependencies becomes scattered. This makes the system harder to maintain, especially in collaborative or long-term development settings.

2.6 Conclusion

Modern large language models such as GPT, Gemini, and Claude are all grounded in the transformer architecture, yet diverge in design choices that influence their performance across tasks. Their capabilities in programming vary not only in terms of code generation quality and debugging support, but also in how they address scale and complexity. While some models handle isolated, well-scoped problems with relative ease, challenges persist when navigating the interconnected structures of larger systems. Limitations such as hallucinations, brittle reasoning under feedback, and susceptibility to insecure code patterns remain evident. The decision to frame two separate research questions for small and large problems explicitly is because the process of solving these problems can be considered fundamentally different.

Chapter 3

Small Problems - Methodology

This chapter outlines the methodology used to evaluate the programming capabilities of large language models (LLMs) when solving relatively small and well-defined algorithmic problems. It begins with a breakdown of the problem set used, followed by an explanation of the models tested, technical setup and prompting methods, and finally, a description of the full solution pipeline and automation system.

The central hypothesis in this chapter is that LLMS should generally perform more accurately and effectively on smaller problems. This is because the problems involve limited complexity and scope. These "small problems" require a focused application of core programming concepts such as loops, conditionals, recursion, and basic data structures. These skills have long been identified as foundational in both introductory programming education and algorithm design [70]. Unlike larger system-level tasks, small problems do not require handling multiple files, managing dependencies, or dealing with persistent state. As a result, they provide a controlled and isolated environment for evaluating LLM capabilities in reasoning, code generation, and syntactic precision [91].

3.1 Problem Set

To evaluate the LLMs, a collection of algorithmic programming problems was selected from LeetCode, a widely used online platform for coding challenges. LeetCode problems span a variety of topics, including algorithms, data structures, and system design. They are commonly used in technical interviews and are designed to simulate real-world programming scenarios in a structured and reproducible way [93].

LeetCode was chosen for this study because it provides a large and diverse set of problems that vary in topic and difficulty [91, 79]. This variety makes it well suited for testing LLMs across multiple dimensions such as computational reasoning, logic synthesis, and data manipulation. Most LeetCode problems are concise in nature, typically solvable within a single function and under 100 lines of code. This makes them particularly suitable for automation and standardized benchmarking [65].

A total of 75 problems were selected: 25 Easy, 25 Medium, and 25 Hard problems. All problems were chosen from those posted within the past two years (2023 to 2024) to minimise the likelihood of training data contamination. This was done to reduce the risk that the models had already seen these problems during their training.

Although LeetCode supports multiple programming languages, Python was chosen for this experiment. Python is the most widely used language on the platform and is known for its readability, simplicity, and concise syntax, making it an excellent choice for these experiments.

Each LeetCode problem contains several core components: a title, a detailed problem description, defined input and output constraints, and a starting code snippet that outlines the function to be implemented. Each problem is also tagged with one or more relevant topics, like "Arrays", "Graphs" or "Dynamic Programming". These components were used to construct the prompts provided to the models and to support later analysis of performance by topic.

For each evaluation, the name, description, constraints, and starting code were included in the prompt provided to the model. The topic tags were not included in the prompts but were stored separately for later analysis. This allows us to assess the types of problems where models struggle the most, broken down by topic.

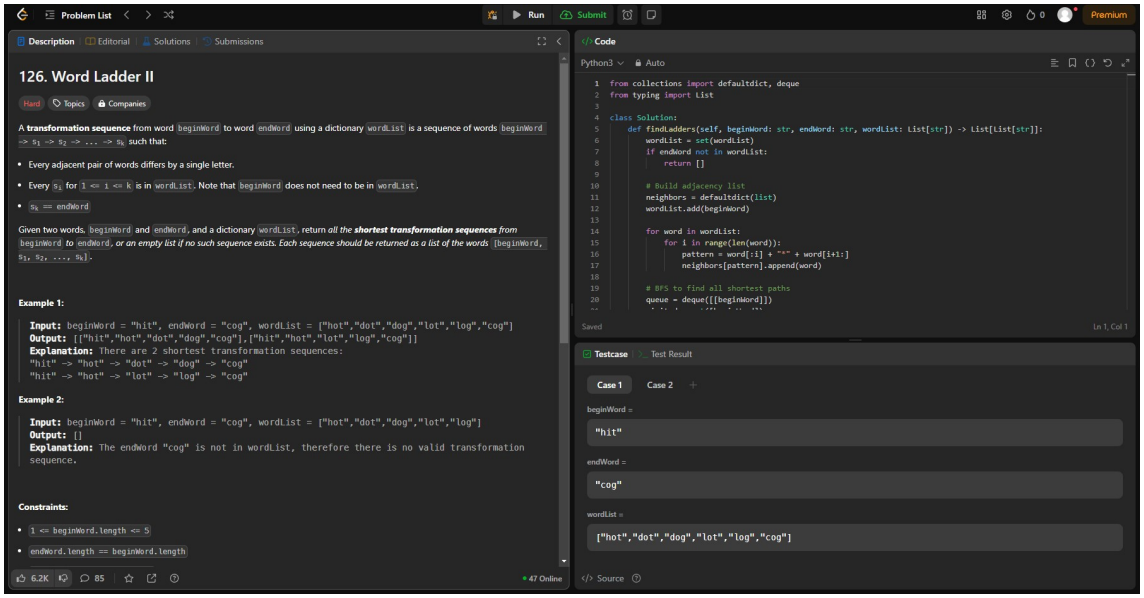

FIGURE 3.1: Sample LeetCode Problem

3.1.1 Easy, Medium and Hard Questions

Title	Topics
Maximum Subarray With Equal Products	Array, Math, Sliding Window, Enumeration, Number Theory
Substring Matching Pattern	String, String Matching
Minimum Operations to Make Columns Strictly Increasing	Array, Greedy, Matrix
Maximum Difference Between Even and Odd Frequency I	Hash Table, String, Counting
Count Subarrays of Length Three With a Condition	Array
Count Partitions with Even Sum Difference	Array, Math, Prefix Sum
Maximum Difference Between Adjacent Elements in a Circular Array	Array
Smallest Number With All Set Bits	Math, Bit Manipulation
Check If Digits Are Equal in String After Operations I	Math, String, Simulation, Combinatorics, Number Theory
Fruits Into Baskets II	Array, Binary Search, Segment Tree, Simulation
Transformed Array	Array, Simulation
Find Valid Pair of Adjacent Digits in String	Hash Table, String, Counting
Minimum Positive Sum Subarray	Array, Sliding Window, Prefix Sum
Minimum Number of Operations to Make Elements in Array Distinct	Array, Hash Table
Maximum Containers on a Ship	Math
Sum of Good Numbers	Array
Sum of Variable Length Subarrays	Array, Prefix Sum
Zigzag Grid Traversal With Skip	Array, Matrix, Simulation
Find Special Substring of Length K	String
Transform Array by Parity	Array, Sorting, Counting
Maximum Unique Subarray Sum After Deletion	Array, Hash Table, Greedy
Button with Longest Push Time	Array
Unique 3-Digit Even Numbers	Array, Hash Table, Recursion, Enumeration
Find the Largest Almost Missing Integer	Array, Hash Table
Minimum Operations to Make Array Values Equal to K	Array, Hash Table

TABLE 3.1: List of EASY LeetCode questions with associated topics.

Title	Topics
Assign Elements to Groups with Constraints	Array, Hash Table
Longest Palindromic Subsequence After at Most K Operations	String, Dynamic Programming
Sort Matrix by Diagonals	Array, Sorting, Matrix
Find Minimum Cost to Remove Array Elements	Array, Dynamic Programming
Paint House IV	Array, Dynamic Programming
Design Spreadsheet	Array, Hash Table, String, Design, Matrix
Maximum and Minimum Sums of at Most Size K Subsequences	Array, Math, Dynamic Programming, Sorting, Combinatorics
Maximum Frequency After Subarray Operation	Array, Hash Table, Dynamic Programming, Greedy, Enumeration, Prefix Sum
Closest Equal Element Queries	Array, Hash Table, Binary Search
Count Mentions Per User	Array, Math, Sorting, Simulation
Minimize the Maximum Edge Weight of Graph	Binary Search, Depth-First Search, Breadth-First Search, Graph, Shortest Path
Choose K Elements With Maximum Sum	Array, Sorting, Heap (Priority Queue)
Minimum Cost to Make Arrays Identical	Array, Greedy, Sorting
Maximum Sum With at Most K Elements	Array, Greedy, Sorting, Heap (Priority Queue), Matrix
Sum of K Subarrays With Length at Least M	Array, Dynamic Programming, Prefix Sum
Properties Graph	Array, Hash Table, Depth-First Search, Breadth-First Search, Union Find, Graph
Find the Number of Copy Arrays	Array, Math
Reschedule Meetings for Maximum Free Time II	Array, Greedy, Enumeration
Fruits Into Baskets III	Array, Binary Search, Segment Tree, Ordered Set
Separate Squares I	Array, Binary Search
Number of Ways to Arrive at Destination	Dynamic Programming, Graph, Topological Sort, Shortest Path
Find the Minimum Amount of Time to Brew Potions	Array, Simulation, Prefix Sum
Zero Array Transformation IV	Array, Dynamic Programming
Eat Pizzas!	Array, Greedy, Sorting
Maximum Manhattan Distance After K Changes	Hash Table, Math, String, Counting

TABLE 3.2: List of MEDIUM LeetCode questions with associated topics.

Title	Topics
Shortest Matching Substring	Two Pointers, String, Binary Search, String Matching
Maximum and Minimum Sums of at Most Size K Subarrays	Array, Math, Stack, Monotonic Stack
Count Beautiful Numbers	Dynamic Programming
Longest Special Path II	Array, Hash Table, Tree, Depth-First Search, Prefix Sum
Lexicographically Smallest Generated String	String, Greedy, String Matching
Check If Digits Are Equal in String After Operations II	Math, String, Combinatorics, Number Theory
Separate Squares II	Array, Binary Search, Segment Tree, Line Sweep
Minimum Increments for Target Multiples in an Array	Array, Math, Dynamic Programming, Bit Manipulation, Number Theory, Bitmask
Frequencies of Shortest Supersequences	Array, String, Bit Manipulation, Graph, Topological Sort, Enumeration
Count Non-Decreasing Subarrays After K Operations	Array, Stack, Segment Tree, Queue, Sliding Window, Monotonic Stack, Monotonic Queue
Longest Common Prefix of K Strings After Removal	Array, String, Trie
Count Substrings Divisible By Last Digit	String, Dynamic Programming
Permutations IV	Array, Math, Combinatorics, Enumeration
Maximum Score of Non-overlapping Intervals	Array, Binary Search, Dynamic Programming, Sorting
Maximize Subarrays After Removing One Conflicting Pair	Array, Segment Tree, Enumeration, Prefix Sum
Maximize the Minimum Game Score	Array, Binary Search, Greedy
Manhattan Distances of All Arrangements of Pieces	Math, Combinatorics
Count the Number of Arrays with K Matching Adjacent Elements	Math, Combinatorics
Maximum Difference Between Even and Odd Frequency II	String, Sliding Window, Enumeration, Prefix Sum
Minimum Operations to Make Array Elements Zero	Array, Math, Bit Manipulation
Maximize the Distance Between Points on a Square	Array, Binary Search, Greedy
Length of Longest V-Shaped Diagonal Segment	Array, Dynamic Programming, Memoization, Matrix
Minimum Cost Good Caption	String, Dynamic Programming
Longest Special Path	Array, Hash Table, Tree, Depth-First Search, Prefix Sum
Maximize Subarray Sum After Removing All Occurrences of One Element	Array, Dynamic Programming, Segment Tree

TABLE 3.3: List of HARD LeetCode questions with associated topics.

3.1.2 Questions by Topic

Some topics are far more prevalent in LeetCode than others. For example, almost every problem includes the "Array" tag. The table below shows the distribution of problem topics in the selected dataset.

Topics	Questions
Array	55
Math, String	16
Dynamic Programming, Hash Table	15
Binary Search, Greedy, Prefix Sum	10
Sorting	9
Enumeration	8
Combinatorics, Matrix, Segment Tree, Simulation	6
Bit Manipulation, Counting, Depth-First Search, Graph, Number Theory, Sliding Window	4
String Matching	3
Breadth-First Search, Heap (Priority Queue), Monotonic Stack, Shortest Path, Stack, Topological Sort, Tree	2
Bitmask, Design, Line Sweep, Memoization, Monotonic Queue, Ordered Set, Queue, Recursion, Trie, Two Pointers, Union Find	1

TABLE 3.4: Grouped LeetCode topics by number of associated questions.

3.2 Models Tested

This section provides an overview of the large language models (LLMs) evaluated in this study. Each model is briefly described, with a focus on its generation, key features, and relevant specifications such as release date. All models were accessed through **OpenRouter**, a unified API gateway that allows seamless integration with multiple LLMs from providers such as OpenAI, Google, and Anthropic. This setup ensured consistent benchmarking and simplified the experimentation process.

When testing models from OpenAI and Google, I used the mini and Flash versions of their latest releases. These versions are designed to be faster and more cost-efficient, and they are also the ones people are more likely to actually use. For example, o1-mini was used instead of the full o1 model, and Gemini 2.5 Flash was used in place of Gemini 2.5 Pro. In the following chapters, the mini and Flash suffixes may sometimes be left out for brevity, but it can be assumed that these are the versions being referred to.

3.2.1 Overview of Evaluated Models

GPT-4o-mini (OpenAI)

GPT-4o (Omni) was released in May 2024 and is the successor to OpenAI's GPT-3 and GPT-3.5 models. It is currently the default model used in ChatGPT's web interface and is also available on the free plan. GPT-4o offers significant improvements over the previous generation, with better performance across a wide range of benchmarks, improved tokenization for non-English languages, and enhanced multimodal capabilities [33].

o1-mini (OpenAI)

Released in September 2024, o1-mini is part of a newer generation of models from OpenAI. It is designed to be a cost-efficient model with a focus on reasoning tasks, especially in STEM fields such as coding and mathematics. It outperforms GPT-4o in many coding scenarios and is optimized for applications that require fast and accurate problem-solving without relying on extensive world knowledge [44].

Gemini 2.0 Flash (Google)

Gemini 2.0 Flash was launched in December 2024 as the successor to Gemini 1.5 Flash [25]. It is a highly capable multimodal model known for its speed and efficiency, offering strong performance in text and code generation. It is the current version used when accessing Gemini through the web interface and is also available to free-tier users. Gemini 2.0 Flash is particularly effective for tasks that require both speed and quality, including software development [17].

Gemini 2.5 Flash (Google)

Previewed in April 2025, Gemini 2.5 Flash builds on the capabilities of 2.0 Flash and introduces hybrid reasoning features that let developers balance performance, quality, and cost [18]. While it retains the speed of the previous version, it adds more advanced reasoning capabilities. It is important to note that this version was still in preview at the time of testing, and its final performance may differ once officially released [54].

Claude 3.7 Sonnet – Thinking (Anthropic)

Released in February 2025, Claude 3.7 Sonnet is Anthropic's most advanced model to date. It introduces hybrid reasoning modes that allow users to choose between fast responses and slower, more thoughtful outputs. The model excels in code generation, data analysis, planning, and general content creation. Claude 3.7 Sonnet is also integrated into development tools like Cursor, where it serves as the default assistant [6].

Model Name	Provider	Release Date
GPT-4o	OpenAI	May 2024
o1-mini	OpenAI	Sep 2024
Gemini 2.0 Flash	Google	Dec 2024
Gemini 2.5 Flash (preview)	Google	Apr 2025
Claude 3.7 Sonnet	Anthropic	Feb 2025

TABLE 3.5: Comparison of evaluated LLMs by provider and release date.

3.2.2 OpenRouter

OpenRouter was used in this project as the main interface for generating model outputs [46]. It provides a simple and unified API that supports multiple large language models from different providers, including OpenAI, Google, and Anthropic. By using a single API key, I was able to access and interact with all the models needed for the evaluation without having to manage separate credentials or endpoints. This streamlined the

development process and allowed for consistent handling of requests across different models. OpenRouter also offered access to all the major models I aimed to test, making it a practical and efficient choice for this research.

Through OpenRouter, you are able to adjust many of the model parameters. The most relevant parameter in our case is related to temperature. The possible values range from provider to provider. For example, in OpenAI it goes from 0 to 2, whereas in Claude it ranges from 0 to 5. However, the general principle is the same, increasing this value increases randomness. It is set to 1 by default on all models mentioned above. This is a key feature of LLMs, allowing them to generate more diverse and sometimes better results, but it does introduce a level of non-determinism. For the purposes of this experiment, I did not adjust any of these settings and left everything at its default value. I did so to mimic the most likely use case of regular individuals, as most people do not change these parameters when using the models.

3.3 Prompting and Technical Setup

This section describes the prompt design strategy and technical setup used during the evaluation of the models. The goal was to standardise input across all models, ensuring fairness and consistency when solving the selected LeetCode problems.

3.3.1 Prompting Strategy

Each model was prompted using a fixed template that included pretext and posttext instructions surrounding the problem description. The prompt was designed to simulate how a human would interpret and solve the problem using the LeetCode platform, while also constraining the model to return usable Python code with minimal formatting issues.

The pretext instructed the model to behave as a LeetCode expert and set the context for the task. The posttext provided formatting constraints, such as preserving the function signature and returning only valid Python code without any natural language commentary or extraneous output. This was important to allow for direct copy-pasting of the generated solution into LeetCode for verification.

Below is an example of a complete prompt sent to each model:

```
Consider yourself a LeetCode problem expert. I am going to give you a problem, and you
    have to solve it.
Here is the problem statement:

You are given an array of positive integers nums.
An array arr is called product equivalent if prod(arr) == lcm(arr) * gcd(arr), where:
prod(arr) is the product of all elements of arr.
gcd(arr) is the GCD of all elements of arr.
lcm(arr) is the LCM of all elements of arr.
Return the length of the longest product equivalent subarray of nums.

Example 1:
Input: nums = [1,2,1,2,1,1,1]
Output: 5
Explanation:
The longest product equivalent subarray is [1, 2, 1, 1, 1], where prod([1, 2, 1, 1, 1]) =
    2,
gcd([1, 2, 1, 1, 1]) = 1, and lcm([1, 2, 1, 1, 1]) = 2.

Example 2:
Input: nums = [2,3,4,5,6]
```

```
Output: 3
Explanation:
The longest product equivalent subarray is [3, 4, 5].

Example 3:
Input: nums = [1,2,3,1,4,5,1]
Output: 5

Constraints:
2 <= nums.length <= 100
1 <= nums[i] <= 10

Right above is the starting code for the problem. Keep the structure of the code the same.
Don't remove the class definition or change/remove the function signature/name,
otherwise LeetCode may throw errors.
Simply write the code inside the function and add any imports you need above the class
    definition.
We are working with Python 3.
Please return only the Python code so that I can directly copy it to LeetCode to verify
    the solution.
```

LISTING 3.1: Prompt Example

This prompt was carefully chosen to elicit a response that would be the most accurate and in a suitable format to copy directly to LeetCode to verify. The statement **"Consider yourself a LeetCode problem expert. I will give you a problem, and you must solve it"** was chosen to guide the AI to take an expert-level problem-solving approach. Research has been done that suggests that providing a clear role when prompting AI models can enhance their problem-solving capabilities [92, 82]. These studies show that structured prompts significantly improve the logical reasoning and problem-solving accuracy of these models. The choice of words to instruct the AI tso act as a **"LeetCode problem expert"** ensures that the response it provides is aligned with the problem-solving mindset required to solve these kinds of problems.

The phrase **"I will give you a problem, and you must solve it."** reinforces a direct, goal-orientated approach and minimises the need for explanations in the final solution. The final statement, **"The starting code for the problem is right above. You must complete the function and return the answer. Please return only the Python code so that I can directly copy it to LeetCode to verify the solution."** was added to ensure only Python code is returned so that it could be easily copied to LeetCode for verification.

This prompt structure was reused across all 75 problems, with the problem description dynamically inserted between the pretext and posttext. Consistent formatting helped reduce response variability and made the evaluation setup robust and reproducible.

3.3.2 Technical Specifications

All experiments were conducted on a custom-built desktop computer. The compute specifications had minimal direct impact on model performance, since all models were accessed through OpenRouter's cloud APIS.

- **Operating System:** Windows 11
- **Processor:** Intel Core i5-13400 (13th Gen)
- **Memory:** 32 GB DDR4 RAM

3.4 Evaluation Metrics

To evaluate the capabilities of each language model, I needed to define clear and measurable criteria. While accuracy was the most obvious and important metric, I also wanted to explore aspects of code quality that could provide deeper insights into how well the models perform beyond just passing test cases.

After reviewing available options, I decided to focus on the following four metrics:

- **Accuracy / Correctness:** This is a binary metric that indicates whether the solution passed all the default test cases on LeetCode. A value of `true` means the solution was accepted; `false` means it failed.
- **Maintainability Index (MI):** A value from 0 to 100 that estimates how easy the code is to understand and maintain. Higher scores suggest cleaner and more maintainable code.
- **Source Lines of Code (SLOC):** This counts the number of actual lines of code in the solution, ignoring comments and blank lines. It helps indicate the verbosity or conciseness of the generated code.
- **Cyclomatic Complexity (CC):** A complexity metric that reflects the number of independent paths through the code. Higher values suggest more complex control flow, which can make the code harder to test and maintain.

By combining these four metrics, I was able to construct a more well-rounded and reliable picture of model performance. Accuracy captures whether the model can solve the problem correctly, but it says little about how the solution is structured. Maintainability Index offers insight into how readable and manageable the code would be for future developers, while Source Lines of Code and Cyclomatic Complexity highlight the conciseness and logical intricacy of the implementation. Taken together, these measures ensure that the evaluation does not reward correctness alone but also accounts for the quality and practicality of the generated code. This multi-dimensional approach aligns more closely with real-world coding standards, where efficient and understandable solutions are valued just as highly as functional correctness [1, 35].

3.5 Technical Implementation

This section explains how the data was collected and managed during the study. The entire pipeline was implemented in Python. Python was chosen because of its rich ecosystem of libraries. Tools like `radon` were used for calculating code metrics, and `selenium` was used to automate the verification process on LeetCode. Python also provides good support for working with APIs, databases, and general automation tasks.

All the collected data was stored in a PostgreSQL database, hosted online using Fliess.io [14]. This setup allowed access from any device, which was important since I often worked from multiple machines. Hosting the database online made the workflow more flexible compared to using a local setup.

3.5.1 Database

The database consisted of two main tables: one for the problems and another for the generated solutions. Each table was designed to store all relevant information needed for querying, analyzing, and evaluating model performance.

Problems Table (`leetcode_questions`)

This table stores all the metadata related to each LeetCode problem used in the study.

- **question_id (TEXT)**: A unique ID for the problem. It is in the form *difficulty-no.* (Eg. easy-12, hard-9 etc.)
- **url (TEXT)**: The direct link to the problem on LeetCode.
- **title (TEXT)**: The problem title.
- **difficulty (TEXT)**: One of `EASY`, `MEDIUM`, or `HARD`.
- **description (TEXT)**: The full problem description shown to the models.
- **python_starting (TEXT)**: The starting Python code template provided by LeetCode.
- **topics (TEXT[])**: A list of topic tags for the problem, such as `Array`, `Graph`, or `Dynamic Programming`.

Python Solutions Table (`leetcode_python_solutions`)

This table stores all the model-generated solutions for each problem, along with the evaluation results and code metrics.

- **uuid (UUID)**: A randomly generated unique identifier for each solution.
- **question_id (TEXT)**: Foreign key referencing the associated problem.
- **model (TEXT)**: The name of the LLM that generated the solution.
- **generated_solution (TEXT)**: The full Python code produced by the model.
- **correct (BOOLEAN)**: Whether the solution passed all LeetCode test cases.
- **maintainability_index (NUMERIC)**: A score between 0 and 100 indicating how easy the code is to maintain.
- **cyclomatic_complexity (NUMERIC)**: A measure of the code's complexity based on control flow.
- **source_lines_of_code (NUMERIC)**: The number of non-empty, non-comment lines of code in the solution.

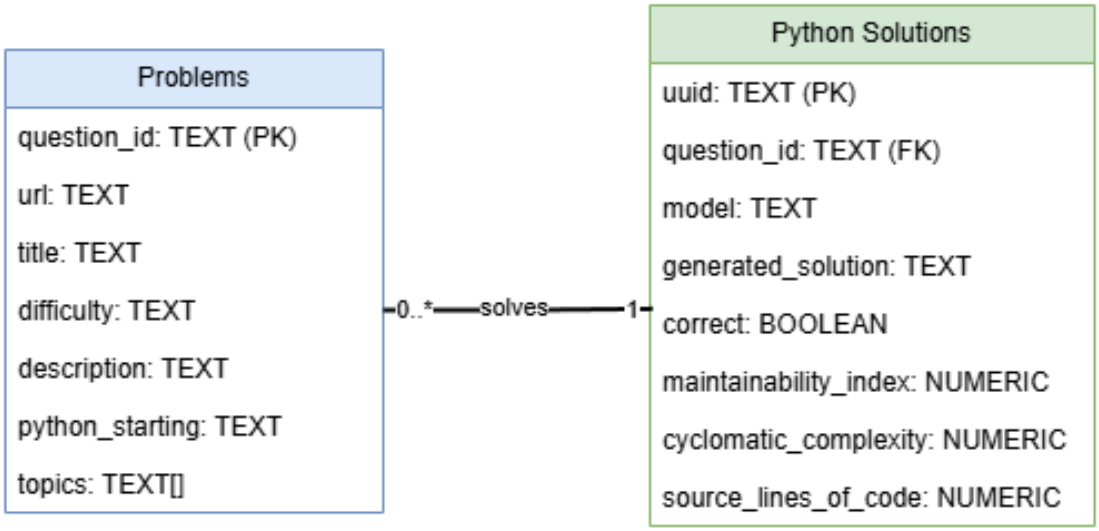

FIGURE 3.2: Database schema used to store problem metadata and model-generated solutions.

3.5.2 Selenium Automations

Early in the project, I realized that one of the biggest time bottlenecks was interacting manually with the LeetCode website [51]. These interactions became especially time-consuming when working with multiple models or a large set of problems. The two main repetitive tasks were:

- Collecting data from the LeetCode website, such as the problem description and starting code. This was originally done by visiting each page and copying the required content into a text file.
- Verifying model-generated solutions by navigating to the problem page, pasting the code, clicking the "Run" button, and waiting for the results to appear.

To automate these steps, I used Selenium, a Python framework that allows for browser automation. Selenium is often used for testing web applications, but it is also effective for scraping content and simulating user interactions with a website.

I built two separate scripts using Selenium:

- The first script scrapes the full problem data, including the title, description, constraints, starting code and topics and then returns a tuple of all this data.
- The second script opens the problem page, pastes the model-generated solution into the code editor, runs the test cases, and returns true or false depending on whether the test cases passed.

These tools allowed me to gather data much faster and made it easy to scale the project by adding more models or problems in the future.

3.5.3 Radon Metrics

To calculate the code quality metrics used in this study, I needed a reliable external library. For this purpose, I used the `radon` library in Python [49]. Radon is a popular tool for analyzing code complexity and maintainability.

It was chosen because it provides built-in support for extracting the three metrics required in this research:

- Maintainability Index (MI)
- Source Lines of Code (SLOC)
- Cyclomatic Complexity (CC)

Each generated solution was passed through Radon using a Python script, and the resulting values were stored in the database along with the solution.

3.5.4 Pipeline

The full solution pipeline connects all components into a single, automated system. It handles everything from selecting problems to evaluating solutions and storing results. Figure 3.3 shows the architecture used in this study.

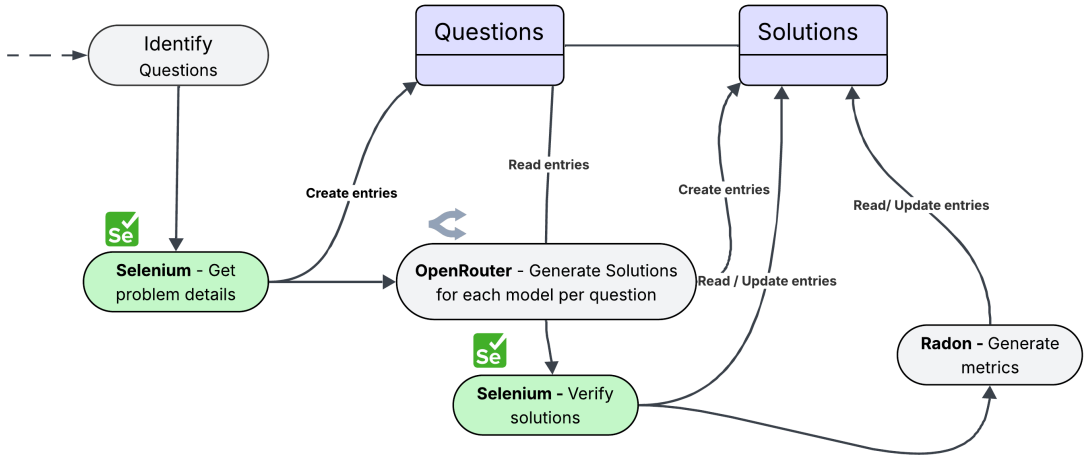

FIGURE 3.3: Solution generation and evaluation pipeline.

The pipeline consists of the following steps:

1. **Identify Questions:** A list of LeetCode questions is manually selected based on difficulty and relevance to the study.
2. **Scrape Problem Details:** A Selenium script is used to visit each LeetCode problem page and extract the description, constraints, and starter code. This information is saved to the `Questions` table in the database.
3. **Generate Solutions via OpenRouter:** For each problem in the database, solutions are generated by calling different models through the OpenRouter API. Each solution is recorded in the `Solutions` table.

4. **Verify Solutions:** A second Selenium script opens the problem page on LeetCode, pastes the model's solution, runs the tests, and checks whether the output is correct. The result is saved back to the `Solutions` table.

5. **Analyze Code with Radon:** Once the solutions are verified, each one is analyzed using the Radon library. This step extracts maintainability, complexity, and size metrics, which are also stored in the database.

Each component of the pipeline reads from and writes to the same database, ensuring that results remain synchronised. This design also makes it easy to scale the system by adding more problems or models in future experiments.

3.6 Conclusion

A curated set of 75 small algorithmic problems from LeetCode was used to test the code generation abilities of modern language models. These problems were carefully selected to cover a wide range of topics and difficulty levels, and all were recent enough to reduce the risk of being seen during training.

Leading models from OpenAI, Google, and Anthropic were prompted using a standardised template to ensure fairness. A fully automated pipeline handled everything from data collection and solution generation to correctness verification and code quality analysis. Tools like Selenium and Radon helped streamline this process, while a shared PostgreSQL database ensured that all results were organised and easy to query. This setup provides a strong foundation for evaluating model performance across multiple dimensions in the next chapter.

Chapter 4

Small Problems - Results

This chapter presents a detailed analysis of how each large language model (LLM) performed on the small-scale programming benchmark. It begins with a breakdown of accuracy across the three difficulty levels, Easy, Medium, and Hard. These results reveal trends in performance, highlight specific model strengths and weaknesses, and identify consistent outliers. To determine whether these differences are meaningful, I apply statistical significance testing using McNemar's test. In addition to accuracy, the chapter examines the quality of code produced by each model using established software engineering metrics, cyclomatic complexity, maintainability index, and source lines of code. These measurements offer further insight into the structure, readability, and efficiency of the generated solutions, providing a more comprehensive view of model performance beyond just the accuracy.

4.1 Model Accuracy by Difficulty

4.1.1 Easy Questions

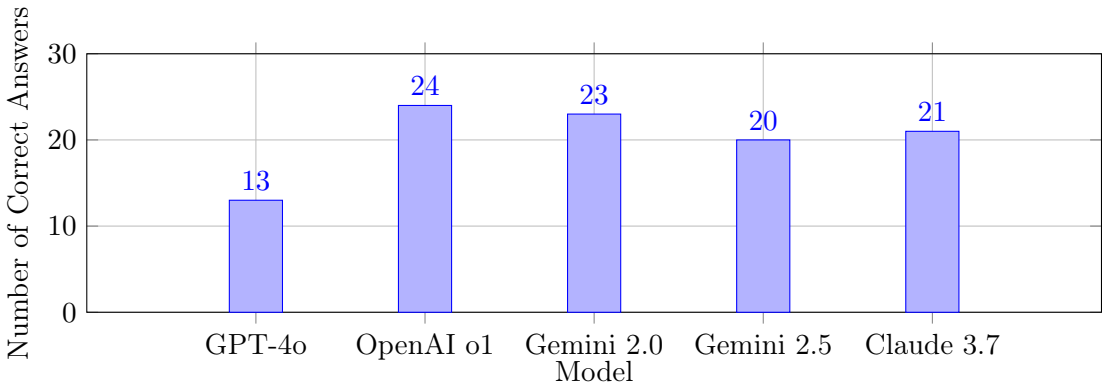

FIGURE 4.1: Easy difficulty correct answers per model.

The model accuracy on the Easy questions was relatively high. All models scored at least 20 correct answers, except GPT-4o, which got just over half correct and stands out as an outlier. OpenAI o1 performed the best, missing only one question and finishing with 24 out of 25. Gemini 2.0 and Claude 3.7 also did well, with 23 and 21 correct answers respectively. Gemini 2.5 followed closely with 20 correct. GPT-4o, in contrast, answered only 13 correctly, trailing significantly behind the rest.

Most models were clustered around the 20 to 24 range, which suggests that the Easy questions were well within their capabilities. GPT-4o's performance, however, shows a significant deviation. This could indicate that either its calibration was off for these questions or it lacked robustness in handling simpler prompts during evaluation. The gap of 11 questions between GPT-4o and the best model, OpenAI o1, is particularly large given the simplicity of this question set.

4.1.2 Medium Questions

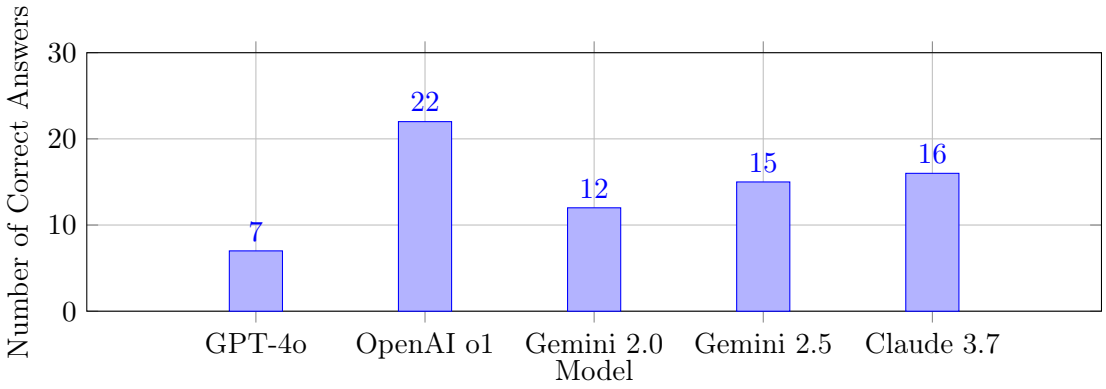

FIGURE 4.2: Medium difficulty correct answers per model.

Accuracy dropped notably for all models on the Medium questions. OpenAI o1 still led with 22 correct answers, showing strong consistency across difficulty levels. The next-best performer was Claude 3.7 with 16, followed by Gemini 2.5 with 15. Gemini 2.0 experienced the largest decline, going from 23 correct in Easy to just 12 here. GPT-4o scored the lowest again, with only 7 correct.

The ranking between models shifted slightly from the Easy set. Claude 3.7 and Gemini 2.5 outperformed Gemini 2.0, reversing their positions from the previous round. The margin between OpenAI o1 and the next best model grew to 6 correct answers, a wider gap than observed in the Easy set. This shows a significant advantage in mid-level reasoning tasks.

The sharp drop in accuracy for most models suggests that the Medium questions introduced a noticeable increase in complexity. Gemini 2.0 showed a steep decline, which points to a lack of generalization beyond simpler tasks. GPT-4o's further drop from 13 to 7 also highlights its difficulties with more nuanced prompts.

4.1.3 Hard Questions

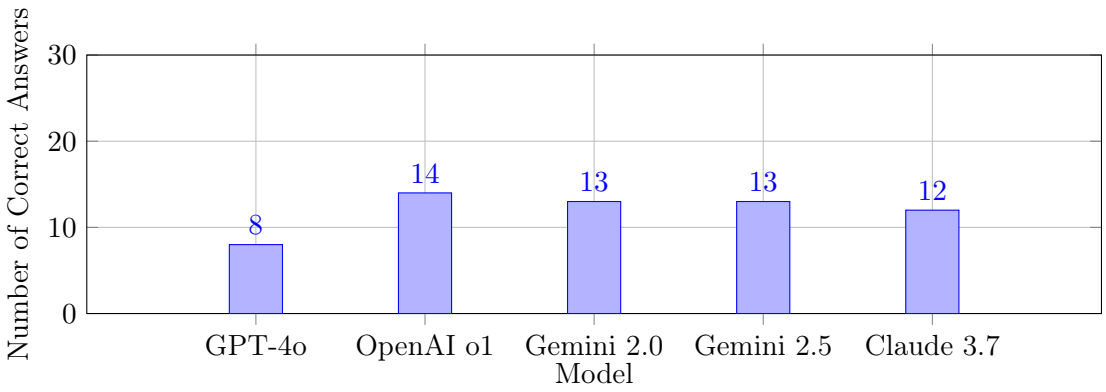

FIGURE 4.3: Hard difficulty correct answers per model.

Performance on the Hard questions was more tightly grouped than in the previous two categories. OpenAI o1 again performed best with 14 correct answers, though this was a significant drop from its 22 on Medium. Gemini 2.0, Gemini 2.5, and Claude 3.7 were closely matched with 13, 13, and 12 correct answers respectively. GPT-4o slightly improved to 8 correct, compared to 7 on Medium.

For most models, the decrease in performance from Medium to Hard was small or nonexistent. Gemini 2.0 and GPT-4o each performed slightly better, scoring one more correct answer than before. This suggests that for those models, the Medium and Hard sets were similar in difficulty. OpenAI o1 showed the largest drop, which may indicate that while it generalizes well across most difficulty levels, its advantage narrows at the hardest questions.

Claude 3.7's performance remained fairly stable, with only a four-point difference from its Easy score. Gemini 2.5's consistent mid-range scores across all levels suggest steady, if not exceptional, performance. The Hard question set overall appears to compress model accuracy into a narrower band. This highlights where the strengths of more capable models begin to converge.

4.1.4 Overall

Model	Easy	Medium	Hard	Total
Gemini 2.0	23	12	13	48
Gemini 2.5	20	15	13	48
GPT-4o	13	7	8	28
OpenAI o1	24	22	14	60
Claude 3.7 Sonnet	21	16	12	49

TABLE 4.1: Correct answers by model and difficulty level, including totals.

OpenAI o1 was the most accurate overall, with 60 correct answers across the 75-question benchmark. It consistently outperformed the other models on every difficulty level. Its advantage was most noticeable on Medium questions, where it exceeded the second-best model by 6 answers. On Easy and Hard questions, the differences between models were smaller, but o1 still maintained a lead.

Gemini 2.0 and Gemini 2.5 both finished with a total of 48 correct answers. While Gemini 2.5 was expected to improve upon 2.0, it did not show a meaningful advantage in this evaluation. It is worth noting that the Gemini 2.5 model used here was still in preview at the time of testing and may not reflect its final capabilities.

Claude 3.7 performed well overall, finishing with 49 correct answers. This was slightly ahead of the Gemini models. Its stability across difficulty levels suggests a strong baseline, though it did not demonstrate the same peak performance as OpenAI o1.

GPT-4o had the weakest overall performance, with a total of just 28 correct answers. It was outperformed by every other model across all difficulty levels. The gap between GPT-4o and OpenAI o1 was 32 correct answers, which highlights a large disparity in performance. It also trailed the Claude and Gemini models by about 20 correct answers, raising concerns about its effectiveness on this benchmark.

4.2 Statistical Significance of Model Accuracy

4.2.1 McNemar's test

To demonstrate that the differences in accuracy between models are not just due to chance, I applied McNemar's test. This test allows me to verify whether the observed performance differences between two models are statistically significant, rather than relying solely on the accuracy totals.

McNemar's test is a non-parametric statistical test used to compare the performance of two classifiers on the same set of data. It is particularly useful when the outcome for each instance is binary, such as "correct" or "incorrect". This makes it well-suited for my evaluation setup, where each model either solves a problem correctly or not.

The test is applied to paired nominal data, meaning both models are tested on the same examples, and their individual outcomes are recorded. Since all models in this experiment were evaluated on the same benchmark of 75 problems, and since there are only two possible outcomes (correct or incorrect), McNemar's test is an appropriate choice for this analysis.

Another important consideration is the sample size. McNemar's test becomes increasingly reliable with larger samples, and the 75 problems used here provide a sufficient number of data points to apply the test with confidence.

The test statistic is computed using the following formula:

$$\chi^2 = \frac{(b-c)^2}{b+c}$$

Where:

- b: the number of instances where Model 1 is correct and Model 2 is incorrect,
- c: the number of instances where Model 1 is incorrect and Model 2 is correct,
- χ^2: the chi-squared test statistic with 1 degree of freedom.

The values of b and c are taken directly from the disagreement between the two models' predictions. The test statistic follows a chi-squared distribution and allows me to compute a corresponding p-value.

The null hypothesis H_0 of McNemar's test assumes that both models perform equally well, meaning the number of disagreements in both directions is the same. If the resulting p-value is less than 0.05, I can reject the null hypothesis H_0, concluding that the performance difference between the two models is statistically significant.

4.2.2 Sample Calculation

To illustrate how McNemar's test was applied in this study, I will walk through one full comparison between two models: OpenAI o1 and Claude 3.7.

From the evaluation results, the performance breakdown between these two models on the 75-question benchmark is as follows:

- Both models correct: 43 questions
- OpenAI o1 correct, Claude 3.7 incorrect: $b = 17$
- OpenAI o1 incorrect, Claude 3.7 correct: $c = 6$
- Both models incorrect: 9 questions

McNemar's test focuses on the values of b and c, which represent the disagreements between the two models. The test statistic is computed using the following formula:

$$\chi^2 = \frac{(b-c)^2}{b+c}$$

Plugging in the values $b = 17$ and $c = 6$, the calculation becomes:

$$\chi^2 = \frac{(17-6)^2}{17+6} = \frac{121}{23} \approx 5.26$$

This test statistic follows a chi-squared distribution with 1 degree of freedom. For this comparison, the p-value was calculated as:

$$p = 0.0218$$

The null hypothesis H_0 in McNemar's test assumes that there is no difference between the two models' performance. That is, both models are equally likely to be correct where the other is not. In mathematical terms:

$$H_0 : b = c$$

To reject the null hypothesis, I require a p-value less than the commonly used significance threshold of 0.05. In this case:

$$p = 0.0218 < 0.05$$

This means I can reject the null hypothesis H_0 and conclude that the performance difference between OpenAI o1 and Claude 3.7 is statistically significant.

Since there are 10 model combinations in total, it would be impractical for me to manually show the full McNemar's test calculation for each one. Instead, I used a Python library to perform the test across all pairs automatically. The complete results of these comparisons are presented in the next subsection.

4.2.3 McNemar Values

Model 1	Model 2	Both ✓	1 ✓ 2 ✗	1 ✗ 2 ✓	Both ✗	P-Value	Better Model	Significant
OpenAI o1	Claude 3.7	43	17	6	9	0.0218	OpenAI o1	✓
OpenAI o1	Gemini 2.0	40	20	8	7	0.0233	OpenAI o1	✓
OpenAI o1	GPT-4o	21	39	7	8	2.38e-06	OpenAI o1	✓
OpenAI o1	Gemini 2.5	40	20	8	7	0.0233	OpenAI o1	✓
Claude 3.7	Gemini 2.0	38	11	10	16	0.8273	Claude 3.7	✗
Claude 3.7	GPT-4o	24	25	4	22	9.64e-05	Claude 3.7	✓
Claude 3.7	Gemini 2.5	38	11	10	16	0.8273	Claude 3.7	✗
Gemini 2.0	GPT-4o	23	25	5	22	0.000261	Gemini 2.0	✓
Gemini 2.0	Gemini 2.5	39	9	9	18	1.0000	Neither	✗
GPT-4o	Gemini 2.5	24	4	24	23	0.000157	Gemini 2.5	✓

FIGURE 4.4: Pairwise McNemar's Test between models. Tick (✓) and cross (✗) indicate whether a model answered a question correctly or incorrectly. Statistical significance is based on $p < 0.05$.

The table above presents the results of the McNemar's test for all ten model pair combinations. These results allow me to draw several conclusions about the relative accuracy of the models.

The OpenAI o1 model shows statistically significant improvement over every other model it was compared against. In all four of its pairwise comparisons against Claude 3.7, Gemini 2.0, Gemini 2.5, and GPT-4o, the p-values are below the threshold of 0.05. This means I can reject the null hypothesis in each case, and conclude that o1 consistently and significantly outperforms the other models. Based on this, it can be considered the most accurate model in the evaluation.

GPT-4o, on the other hand, performs significantly worse than all other models. Each of its comparisons results in a p-value smaller than 0.05, indicating strong statistical evidence that the model underperforms relative to its peers. This places GPT-4o clearly at the bottom of the accuracy ranking among the five evaluated models.

When comparing Claude 3.7, Gemini 2.0, and Gemini 2.5, none of the p-values are below the significance threshold. This suggests that although there are numerical differences between their scores, those differences are not statistically meaningful. Their performances are therefore considered comparable based on the data available.

The results can be grouped into three distinct accuracy tiers:

- **Top tier:** OpenAI o1, statistically better than all other models.
- **Middle tier:** Claude 3.7, Gemini 2.0, and Gemini 2.5, with no significant difference among them.
- **Bottom tier:** GPT-4o, significantly worse than all others.

As such, we can say that OpenAI o1 is the best model in terms of correctness and accuracy.

4.3 Evaluation Metrics

In this section, I evaluate the models using three key code-level metrics: Maintainability Index (MI), Cyclomatic Complexity (CC), and Source Lines of Code (SLOC). These metrics provide insight into the quality, complexity, and readability of the code produced by each model.

Model	Maintainability Index	Cyclomatic Complexity	Source Lines of Code
OpenAI o1	72.64	8.87	28.45
GPT-4o	77.77	6.35	18.60
Gemini 2.0	65.99	6.46	23.35
Gemini 2.5	65.68	8.07	27.08
Claude 3.7	82.32	8.27	21.69

Table 4.2: Average code-level metrics per model across all submitted solutions.

The table above presents the average value of each metric across all evaluated submissions per model. There is considerable variation across models, particularly in maintainability and complexity, which suggests that some models are more likely to produce readable and maintainable code than others. In the following subsections, I break down each metric individually and analyze how the models compare. Based on this analysis, I group the models into performance tiers.

As with accuracy, I wanted to assess whether the observed differences in these metrics are statistically significant. While average values are informative, they are not sufficient for determining significance on their own. To make that determination, it is necessary to analyze the paired differences between each model's outputs across all problems.

4.3.1 Wilcoxon Signed-Rank Test

To assess statistical significance between model pairs for each metric, I used the Wilcoxon signed-rank test. This test is well-suited to my use case because it is designed for comparing two related samples. In this case, I am comparing the metric values generated by two models for the same set of code outputs across identical problem instances.

I initially considered using a paired t-test, which is also used to compare paired samples. However, one of the core assumptions of the t-test is that the differences between the paired observations are normally distributed. To check this, I created distribution plots for the metric values across all model pairs. These plots showed that the data was skewed and not normally distributed. Because of this, I chose to use the Wilcoxon signed-rank test instead.

The Wilcoxon test is a non-parametric alternative that does not require the normality assumption. It works by comparing the relative ranks of the differences between pairs, and it evaluates whether the median difference between the two sets is significantly different from zero. This makes it an appropriate choice for my analysis, where I want to know whether one model consistently produces higher or lower values for a given metric compared to another.

To perform the test, I used the Python `scipy` library, which had an appropriate function for this. The function computes the test statistic and the corresponding p-value for each pair of models.

I applied the test to every pair of models across all three metrics. The results are presented in the following tables, along with an indication of whether the differences are statistically significant using a threshold of 0.05.

4.3.2 Cyclomatic Complexity

Model 1	Model 2	Avg CC 1	Avg CC 2	P-Value	Significant
OpenAI o1	Claude 3.7	8.87	8.27	0.4577	✗
OpenAI o1	Gemini 2.0	8.87	6.46	0.0008	✓
OpenAI o1	GPT-4o	8.87	6.35	1.66e-05	✓
OpenAI o1	Gemini 2.5	8.87	8.07	0.4084	✗
Claude 3.7	Gemini 2.0	8.27	6.46	0.0004	✓
Claude 3.7	GPT-4o	8.27	6.35	0.0003	✓
Claude 3.7	Gemini 2.5	8.27	8.07	0.4839	✗
Gemini 2.0	GPT-4o	6.46	6.35	0.9753	✗
Gemini 2.0	Gemini 2.5	6.46	8.07	0.0332	✓
GPT-4o	Gemini 2.5	6.35	8.07	0.0616	✗

TABLE 4.3: Pairwise comparisons of average Cyclomatic Complexity between models. Statistical significance is determined by $p < 0.05$.

The results show a clear division in cyclomatic complexity across the models. OpenAI o1, Gemini 2.5, and Claude 3.7 make up the upper tier, consistently generating code with

higher complexity. OpenAI o1 leads with an average score of 8.87, followed closely by Claude 3.7 at 8.27 and Gemini 2.5 at 8.07. This suggests that these models tend to produce solutions with more intricate control flow and decision-making.

Gemini 2.0 and GPT-4o fall into a second group with noticeably lower complexity scores, at 6.46 and 6.35 respectively. The gap between these two tiers is statistically significant, confirming that the top three models produce consistently more complex code than the bottom two.

Within the higher tier, the differences between models are not significant. This indicates that while their outputs are more complex overall, they perform similarly in this respect when compared to one another.

I believe this pattern reflects a trade-off in model behavior. The more complex solutions may be better equipped to handle nuanced logic, which could support higher accuracy in difficult tasks. At the same time, increased complexity can affect readability and maintainability, making it important to consider the intended use of the generated code.

4.3.3 Maintainability Index

Model 1	Model 2	Avg MI 1	Avg MI 2	P-Value	Significant
OpenAI o1	Claude 3.7	72.64	82.32	6.52e-07	✓
OpenAI o1	Gemini 2.0	72.64	65.99	9.96e-04	✓
OpenAI o1	GPT-4o	72.64	77.77	6.28e-03	✓
OpenAI o1	Gemini 2.5	72.64	65.68	5.36e-03	✓
Claude 3.7	Gemini 2.0	82.32	65.99	1.78e-11	✓
Claude 3.7	GPT-4o	82.32	77.77	1.05e-02	✓
Claude 3.7	Gemini 2.5	82.32	65.68	4.04e-11	✓
Gemini 2.0	GPT-4o	65.99	77.77	7.81e-07	✓
Gemini 2.0	Gemini 2.5	65.99	65.68	0.7629	✗
GPT-4o	Gemini 2.5	77.77	65.68	2.54e-07	✓

TABLE 4.4: Pairwise comparisons of average Maintainability Index (MI) between models. Statistical significance is determined by $p < 0.05$.

Claude 3.7 achieves the highest maintainability index across all models, with an average score of 82.32. This result places it at the top when it comes to producing code that is readable and easy to work with. GPT-4o also performs well, scoring 77.77. These two models form the upper tier in terms of maintainability.

OpenAI o1 falls in the middle with a score of 72.64. While it does not match the top two models, it still performs better than both Gemini 2.0 and Gemini 2.5, which score 65.99 and 65.68, respectively. These two consistently produce the least maintainable code and form the lower tier in this metric.

Statistical comparisons confirm these groupings. Claude 3.7 significantly outperforms all other models, and GPT-4o shows a clear advantage over both Gemini models. The difference between Gemini 2.0 and Gemini 2.5 is not significant, indicating that their performance in this area is effectively the same.

I believe these results highlight Claude 3.7's strength in generating clean, maintainable code. GPT-4o also delivers strong results in this regard, while the Gemini models fall short on this metric.

4.3.4 Source Lines of Code

Model 1	Model 2	Avg LOC 1	Avg LOC 2	P-Value	Significant
OpenAI o1	Claude 3.7	28.45	21.69	0.0104	✓
OpenAI o1	Gemini 2.0	28.45	23.35	0.0199	✓
OpenAI o1	GPT-4o	28.45	18.60	2.59e-05	✓
OpenAI o1	Gemini 2.5	28.45	27.08	0.6160	✗
Claude 3.7	Gemini 2.0	21.69	23.35	0.8831	✗
Claude 3.7	GPT-4o	21.69	18.60	0.0053	✓
Claude 3.7	Gemini 2.5	21.69	27.08	0.0393	✓
Gemini 2.0	GPT-4o	23.35	18.60	0.0549	✗
Gemini 2.0	Gemini 2.5	23.35	27.08	0.1075	✗
GPT-4o	Gemini 2.5	18.60	27.08	0.0002	✓

TABLE 4.5: Pairwise comparisons of average Source Lines of Code (SLOC) between models. Statistical significance is determined by $p < 0.05$.

The results for Source Lines of Code (SLOC) show that OpenAI o1 produces the longest outputs, with an average of 28.45 lines per solution. Gemini 2.5 follows closely with 27.08 lines, while Gemini 2.0 averages 23.35. Claude 3.7 and GPT-4o generate the shortest code, at 21.69 and 18.60 lines respectively.

Statistical comparisons confirm that OpenAI o1 produces significantly longer code than Claude 3.7, GPT-4o, and Gemini 2.0. Its output length is not significantly different from Gemini 2.5, placing the two models in a similar category in terms of verbosity.

GPT-4o stands out as the most concise model. It produces significantly shorter code than Claude 3.7 and Gemini 2.5, indicating a clear preference for brevity.

Gemini 2.0 falls in between and does not differ significantly from Claude 3.7 or Gemini 2.5. This places it in a more neutral zone with regard to output length, showing less consistency than the other models.

4.4 Token Usage

Model	Avg Input Tokens	Avg Output Tokens	Avg Generation Time (ms)
OpenAI o1	555	4938	23898
GPT-4o	520	216	3752
Gemini 2.0	519	301	959
Gemini 2.5	565	2340	14420
Claude 3.7	549	770	9569

TABLE 4.6: Average token usage and generation time for each model.

The table above shows the average token usage and generation times for each model across the benchmark, gathered from OpenRouter. The average number of input tokens remains fairly similar across all models, indicating that the encoding of the prompts was relatively consistent across all models. The range is from 519 to 565, with Gemini 2.0 having the smallest and Gemini 2.5 having the largest token usage.

The output tokens seem to show a much more varied story. The leading models for this metric are OpenAI o1 and Gemini 2.5, with 4938 and 2340, respectively. This has also resulted in them having the longest generation times, with OpenAI o1 requiring, on average, 24,000 ms for a solution and Gemini 2.5 requiring 14,420 ms. This long output token usage makes more sense when examining the source code lines, where we can see that o1 and Gemini 2.5 generated the longest solutions on average. In the case of the o1 model, the additional output token usage and longer generation time seem to be worthwhile, given its good accuracy. However, for Gemini, it seems that this did not come at the cost of improved accuracy.

Claude 3.7 was in the middle of the pack for both output tokens, having an average output token size of 770, significantly less than the top two, and a generation time of 9569 ms. The biggest surprise with regard to generation time is GPT-4o. It had the shortest output token length, which we can also confirm from seeing its short SLOC metric. However, it still had a relatively large generation time of 3752 ms. This is much longer than the comparable Gemini 2.0 model, which had 301 output tokens but only 959 ms. Given this, along with the long generation time of the o1 model, we can say that the generation time of the models from OpenAI seems to be much longer than that of the other providers. However, this could be due to many factors separate from the models themselves, such as the API or resources that OpenRouter itself uses to access the model.

4.5 Conclusion

The results show a clear hierarchy in model performance. OpenAI o1 emerged as the top performer, consistently achieving the highest accuracy across all difficulty levels and outperforming every other model in statistical comparisons. It also produced the most complex and verbose code, suggesting a trade-off between correctness and maintainability.

Claude 3.7 followed closely in accuracy and led in code quality, generating the most maintainable and readable solutions. While it did not reach o1's level of correctness, its outputs were clean and consistent across metrics. The two Gemini models performed similarly, with Gemini 2.5 showing no clear advantage over its predecessor. Both models produced less maintainable code and showed mid-range accuracy, offering no strong edge in either correctness or quality. GPT-4o had the weakest performance overall. While its code was the shortest and simplest, it struggled to solve problems correctly and consistently fell behind on every metric.

In terms of token usage and generation time, OpenAI o1 and Gemini 2.5 Flash produced the longest outputs, which correlated with their extended generation times. Claude 3.7 struck a balance between output size and latency, maintaining competitive generation times without excessive verbosity. Interestingly, despite generating the shortest outputs, GPT-4o still had relatively long generation times compared to Gemini 2.0, pointing to possible external factors such as API latency or server-side throttling.

Chapter 5

Small Problems - Discussion

In this chapter, we will discuss the results obtained in the previous chapter. Rather than focusing only on raw numbers, we will try to extract meaningful insights from the accuracy scores and code metrics. The aim is to understand not just how the models performed, but why they performed the way they did, and what that tells us about their strengths and weaknesses.

We will also have a look at the specific topics that the models struggled with. This is a little bit complicated, as we saw in Table 3.4, the topics are distributed very unevenly. Some topics only have one or two questions, while others, like Array make up more than two-thirds of the dataset. As such, for the weakest topics, I will explicitly mention those that have between 4 and 20 questions sampled. From these, I will calculate which topics had the highest proportion of correct answers and select them. It is also important to note that this is relative to the performance of the model on other topics. For example, OpenAI o1 has much better accuracy overall than GPT-4o, so one of its weaker topics might be one of GPT-4o's strongest ones. This means that each model will have its own weakest topics, and they are not necessarily comparable between models. Also note, that the array topic is not mentioned in the graph as it is part of more than half of the problem set.

5.1 OpenAI o1

Stats:

- Accuracy: 60/75
- Average Cyclomatic Complexity: 8.87
- Average Source Lines of Code (SLOC): 28.45
- Average Maintainability Index (MI): 72.64
- Weakest Topics: Prefix Sum, Dynamic Programming, Math, Sliding Window, Segment Tree

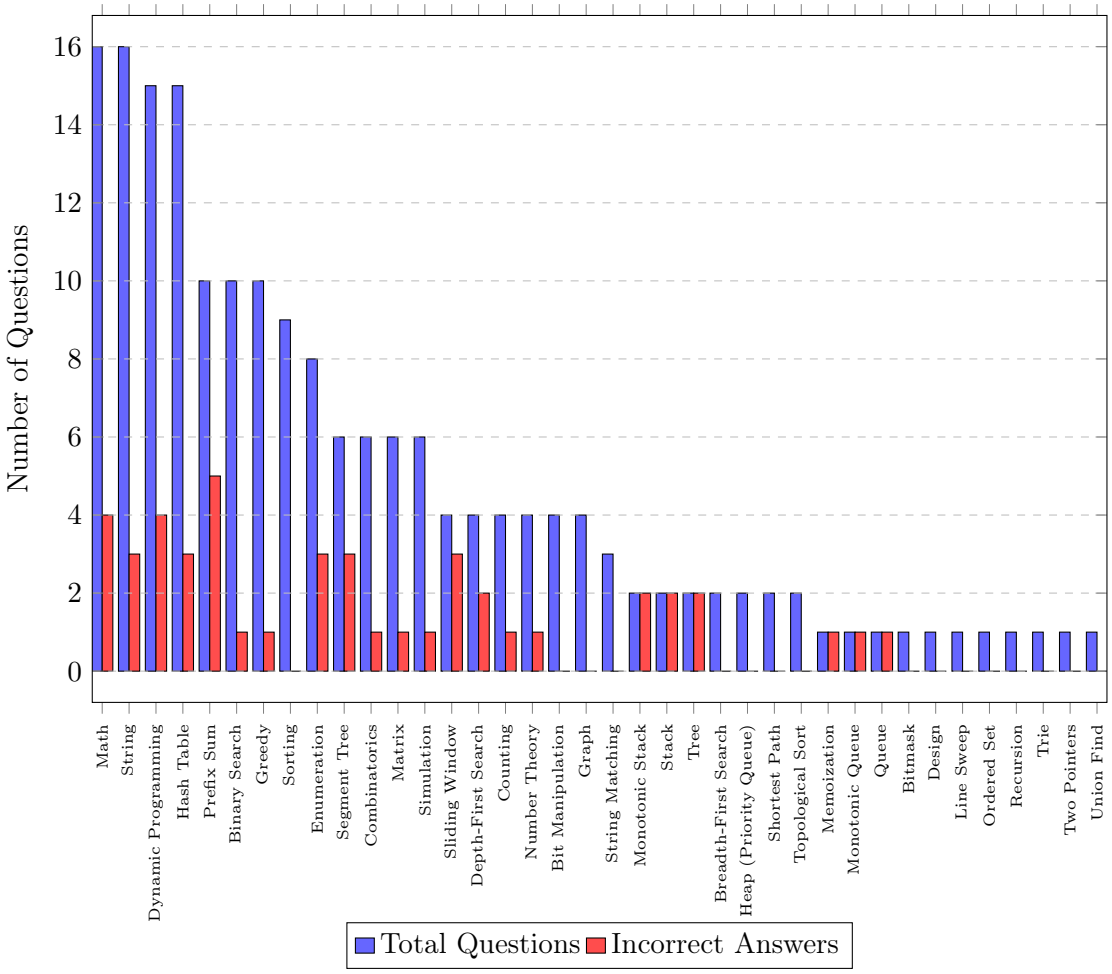

FIGURE 5.1: OpenAI o1-mini: Topic-Level Question Breakdown

OpenAI o1 is the top-performing model in this evaluation. It correctly solves 60 out of 75 problems, earning the highest accuracy score overall. It also places first across all three difficulty levels: Easy, Medium, and Hard. No other model achieves this. Based on statistical tests, this lead is consistent and significant in all direct comparisons.

At the code level, o1 leans toward more complex solutions. Its cyclomatic complexity score averages 8.87, the highest among all models. This suggests that o1 often relies on deeper or more intricate logic to solve problems. While such complexity might raise concerns in general software development, in this context it appears to support better accuracy.

Its outputs are also the most verbose. With an average of 28.45 source lines of code per solution, o1 produces longer and more detailed responses than any other model. Its code length is about 50 percent greater than GPT-4o's and also exceeds that of Claude 3.7 and Gemini 2.0.

The maintainability index is 72.64, placing it in the middle range. It scores lower than GPT-4o and Claude 3.7 but remains ahead of both Gemini versions. Despite the added complexity and length, the code remains reasonably readable and usable.

I believe OpenAI o1 is the best option for scenarios where correctness is the top priority. Its strong performance makes it especially valuable in high-stakes environments where small mistakes can have a major impact. I recommend using it in tasks that demand high accuracy, with the understanding that its code may need to be simplified or cleaned up

before deployment.

5.2 GPT-4o

Stats:

- Accuracy: 28/75
- Average Cyclomatic Complexity: 6.35
- Average Source Lines of Code (SLOC): 18.60
- Average Maintainability Index (MI): 77.77
- Weakest Topics: Binary Search, Enumeration, Sorting, Bit Manipulation, Depth-First Search, Counting, Hash Table

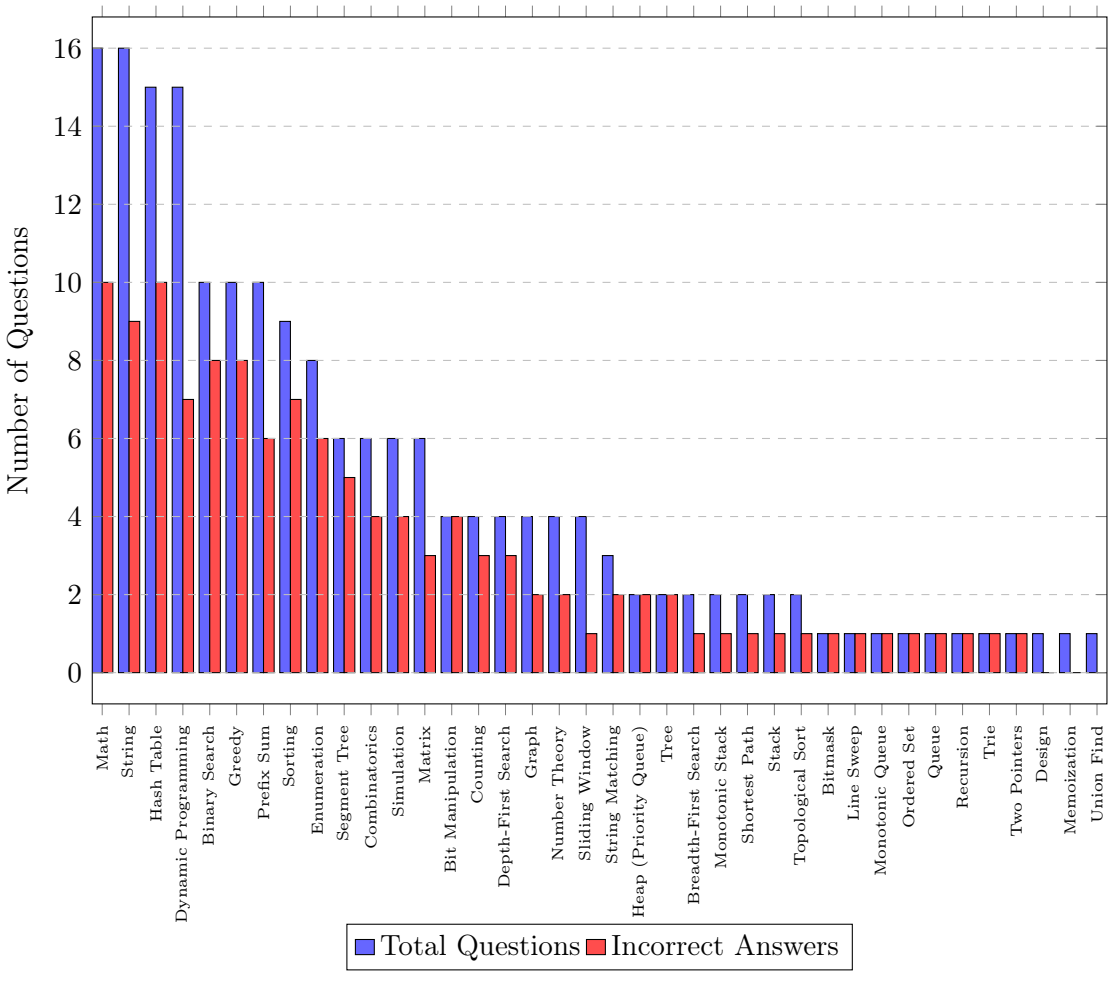

FIGURE 5.2: OpenAI o4-mini: Topic-Level Question Breakdown

GPT-4o ranks the lowest in terms of accuracy. It solves 28 out of 75 problems correctly, placing it at the bottom of the evaluation. Statistical comparisons show that it underperforms across the board and does not outperform any other model in direct head-to-head results.

Despite the weak accuracy, GPT-4o has strengths in code quality. It produces the simplest and most concise outputs, with an average cyclomatic complexity of 6.35 and just 18.60 lines of code per solution. Among all models, its outputs are the shortest and the least complex. It also achieves a strong maintainability index of 77.77, suggesting that its code is relatively easy to understand and work with.

These advantages are noteworthy but do not outweigh the low correctness. The primary goal of code generation models is to produce accurate and functional solutions, and GPT-4o often fails to meet that standard. Well-structured code is not helpful if it does not solve the problem.

I do not recommend GPT-4o for tasks where accuracy is a key requirement. It might be useful during the early stages of prototyping, especially when speed or simplicity is important. Still, I believe any output from this model would need to be reviewed carefully, and frequent corrections would likely be necessary before it could be used with confidence.

5.3 Gemini 2.0

Stats:

- Accuracy: 47/75
- Average Cyclomatic Complexity: 6.46
- Average Source Lines of Code (SLOC): 23.35
- Average Maintainability Index (MI): 65.99
- Weakest Topics: Binary Search, Sorting, Bit Manipulation, Depth-First Search, Dynamic Programming, Math

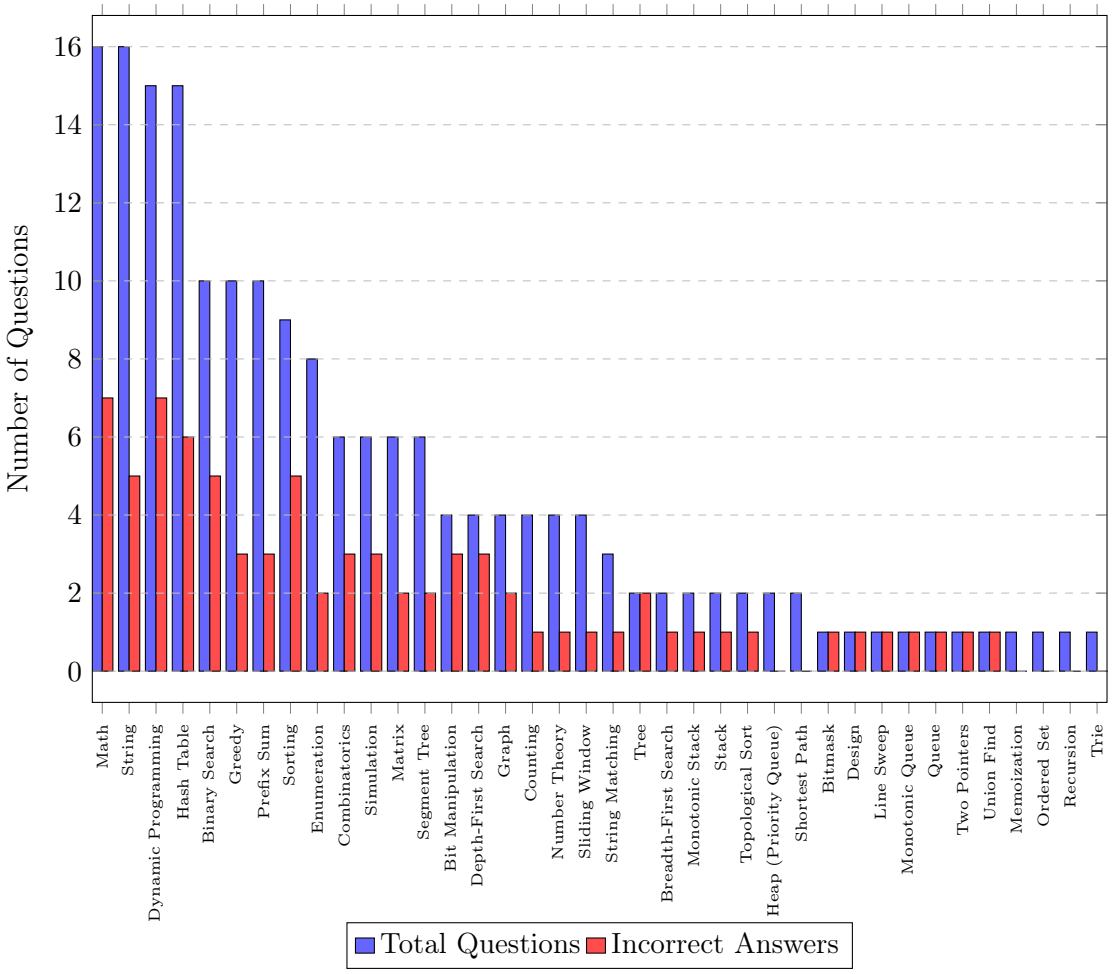

FIGURE 5.3: Gemini 2.0 Flash: Topic-Level Question Breakdown

Gemini 2.0 performs solidly but does not lead in any area. It answers 47 out of 75 problems correctly, placing it just behind Claude 3.7 and one point below Gemini 2.5. The difference between the two Gemini models is not statistically significant, which suggests they perform at roughly the same level in terms of accuracy.

The code generated by Gemini 2.0 tends to be short and relatively simple. Its average cyclomatic complexity is 6.46, and the average number of source lines of code is 23.35. These figures point to a model that favours brevity and straightforward solutions.

That said, its simplicity does not seem to improve code maintainability. The model scores a maintainability index of 65.99, which is only slightly higher than Gemini 2.5 and the second-lowest overall. Statistical comparisons show that it produces less maintainable code than Claude 3.7, GPT-4o, and OpenAI o1.

I believe Gemini 2.0 is a capable but unremarkable model. Its performance is generally average, and it does not offer a clear advantage over stronger alternatives. I would recommend considering other models for tasks that require high accuracy or well-structured code.

5.4 Gemini 2.5

Stats:

- Accuracy: 48/75
- Average Cyclomatic Complexity: 8.07
- Average Source Lines of Code (SLOC): 27.08
- Average Maintainability Index (MI): 65.68
- Weakest Topics: Hash Table, String, Greedy, Bit Manipulation, Depth-First Search, Matching

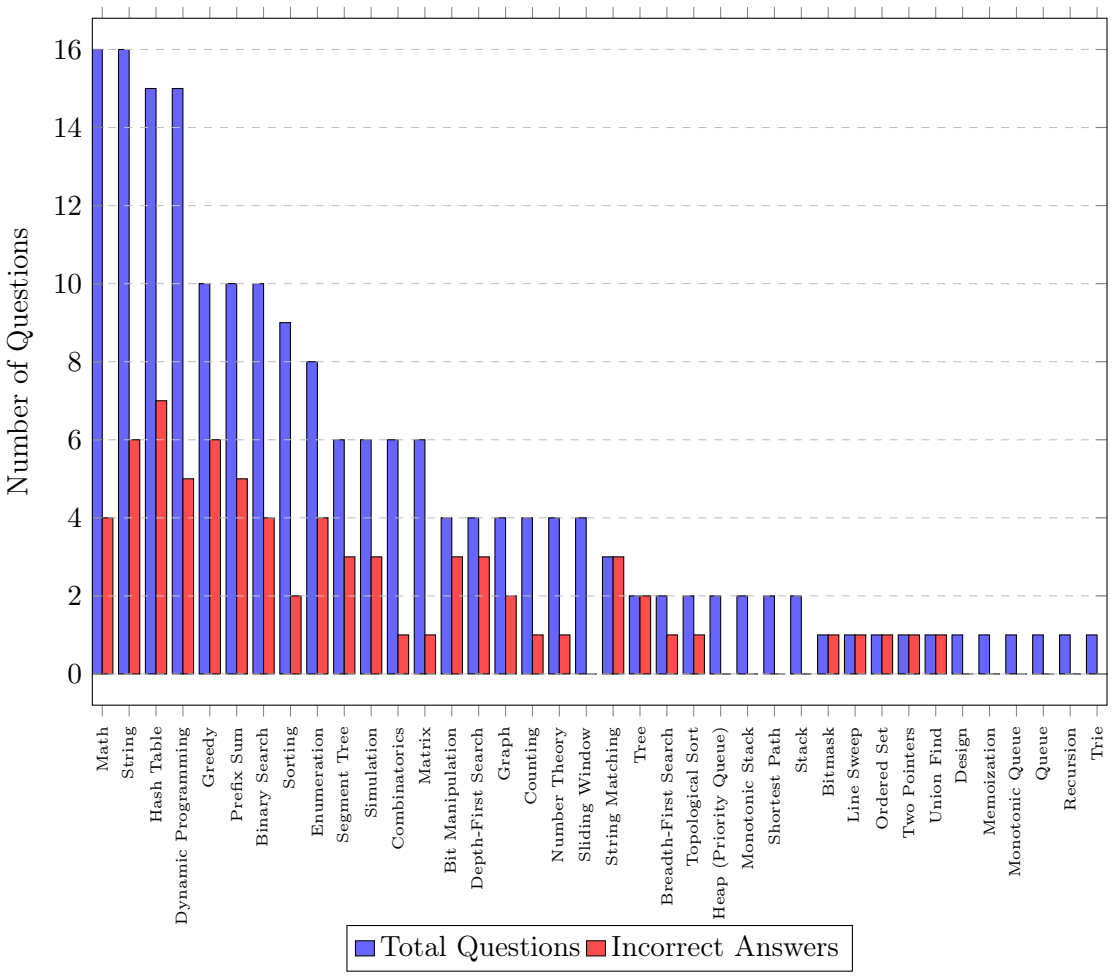

FIGURE 5.4: Gemini 2.5 Flash: Topic-Level Question Breakdown

Gemini 2.5 is perhaps the most surprising result in this evaluation, mainly because it falls well short of my expectations. Given its recent release and the performance improvements suggested by Google, I expected it to be competitive with OpenAI o1 or at least show a clear improvement over Gemini 2.0. Based on the results so far, that improvement has not materialized.

It scores 48 out of 75 in accuracy, which is nearly identical to Gemini 2.0. Across all difficulty levels, the differences between the two models are small and statistically insignificant. The results do not support the idea that Gemini 2.5 offers a meaningful step forward in accuracy.

When it comes to code quality, the model also struggles. It produces the least maintainable code in the entire evaluation, with a maintainability index of 65.68. This is almost the same as Gemini 2.0 and confirms that the newer version does not improve on this front. It also generates some of the longest outputs, with an average of 27.08 lines of code, and tends to use relatively complex structures, with a cyclomatic complexity of 8.07.

Taken together, these results show a model that produces lengthy and harder-to-maintain code without delivering better correctness. I do not recommend Gemini 2.5 for code generation tasks in its current preview form. It may improve once fully released, and I intend to run additional tests in a few months to see if any meaningful progress has been made.

5.5 Claude 3.7

Stats:

- Accuracy: 49/75
- Average Cyclomatic Complexity: 8.27
- Average Source Lines of Code (SLOC): 21.69
- Average Maintainability Index (MI): 82.32
- Weakest Topics: Dynamic Programming, Hash Table, Segment Tree, String, Depth-First Search

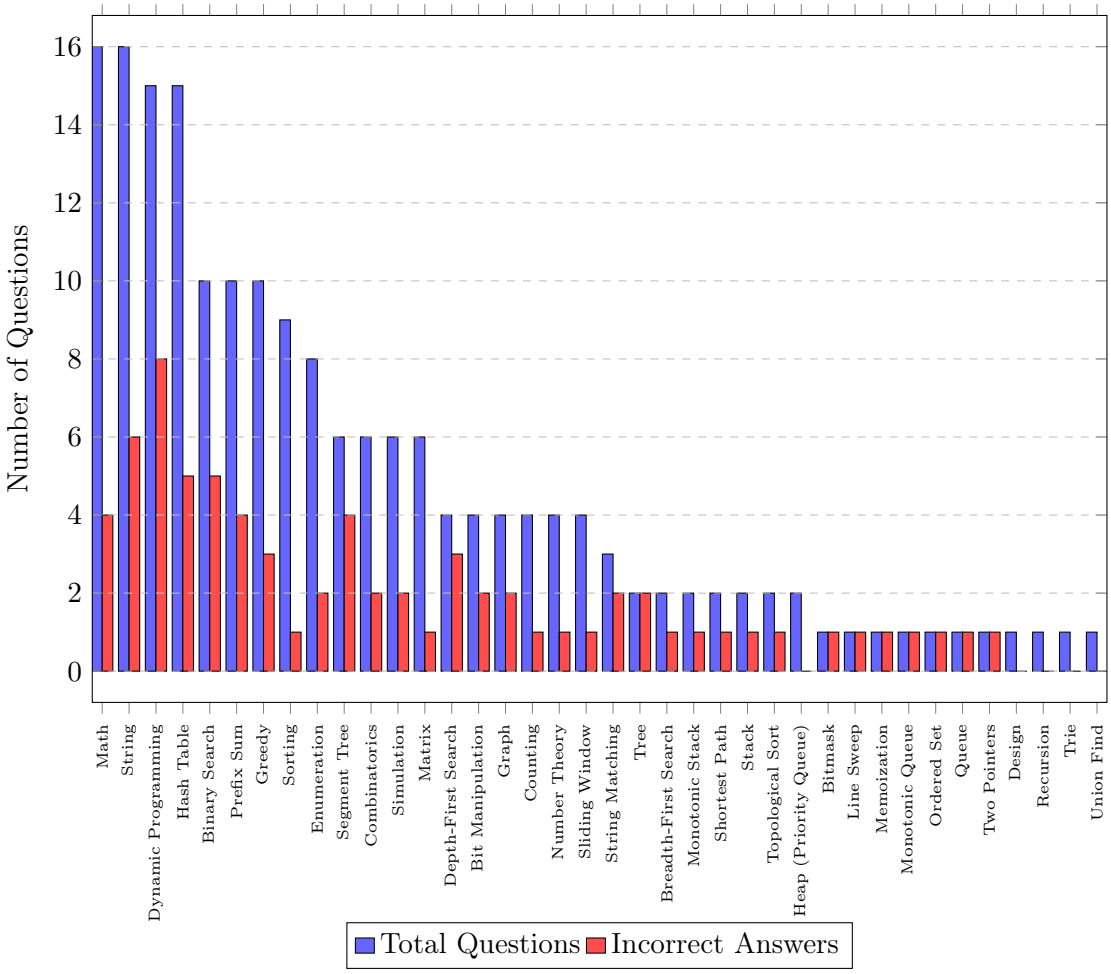

FIGURE 5.5: Claude 3.7 Sonnet: Topic-Level Question Breakdown

Claude 3.7 stands out as the best model in terms of code maintainability. It achieves a maintainability index of 82.32, the highest among all models in this evaluation. Statistical comparisons show that it consistently outperforms the others on this metric, indicating a strong focus on producing code that is clean, readable, and easy to work with.

The average output length is 21.69 lines of code, placing it near the middle of the group. This suggests that Claude strikes a solid balance between being concise and being expressive. Its code avoids extremes and was neither too short to lack clarity nor too long to become cluttered. The cyclomatic complexity is 8.27, which is on the higher side, but still below that of OpenAI o1. This level of complexity does not seem to harm readability or structure.

In terms of accuracy, Claude performs on par with the Gemini models, solving 49 out of 75 problems. This places it in the second tier behind OpenAI o1, which remains the top performer overall. While Claude does not lead in correctness, it consistently performs better than GPT-4o and remains competitive with the other models.

I believe Claude 3.7 is an excellent choice for use cases where clean and maintainable code is important. It is especially well-suited for educational platforms, internal development tools, or any setting where clarity and long-term code quality are a priority. I recommend it for tasks where maintainability matters as much as correctness.

5.6 Individual Problems Analysis

In total, the five models produced 375 solutions for the 75 problem set. Due of the volume, I am unable to individually examine every submission line by line, which is why I settled for a simple binary score (solved or not) for each attempt. In this section, I focus on a small selection of problems and discuss the differences in the generated solutions of each model. For each problem, I include the code generated by each model with comments removed.

OpenAI o1	GPT-4o	Gemini 2.5	Gemini 2.0	Claude 3.7	Problem Count
✓	✓	✓	✓	✓	15
✓	✗	✓	✓	✓	12
✓	✗	✗	✗	✗	7
✓	✗	✓	✓	✗	5
✗	✗	✗	✗	✗	5
✓	✗	✗	✓	✓	4
✓	✗	✗	✗	✓	4
✗	✓	✓	✓	✓	4
✓	✓	✓	✗	✓	3
✓	✗	✓	✗	✓	3
✓	✗	✓	✗	✗	2
✓	✗	✗	✓	✗	2
✗	✓	✓	✓	✗	2
✓	✓	✗	✓	✓	1
✓	✓	✗	✓	✗	1
✓	✓	✗	✗	✓	1
✗	✓	✗	✗	✗	1
✗	✗	✓	✓	✓	1
✗	✗	✓	✗	✗	1
✗	✗	✗	✓	✓	1

TABLE 5.1: Combinations of models that successfully solved each group of problems

The table above lists, for each subset of the five models, the number of problems solved exclusively by that group. For example, the first row shows that 15 out of 75 problems were solved correctly by all of the models. This table reinforces the three tiers defined earlier for the models.

In this section, we will sample problems from the top 3 permutations with the highest frequency. These permutations are:

- All models managed to solve the problem
- All models except GPT-4o managed to solve the problem
- Only OpenAI o1 managed to solve the problem

Problem 1

This is an easy Leetcode problem which all the models were able to solve.

- **Problem Name:** Sum of Variable Length Subarrays

- **Difficulty:** Easy
- **Topics:** Array, Prefix Sum

```
You are given an integer array nums of size n. For each index i where 0 <= i < n,
define a subarray nums[start ... i] where start = max(0, i - nums[i]).

Return the total sum of all elements from the subarray defined for each index in
    the array.

Example 1:
Input: nums = [2,3,1]
Output: 11

Example 2:
Input: nums = [3,1,1,2]
Output: 13

Constraints:
1 <= n == nums.length <= 100
1 <= nums[i] <= 1000
```

Listing 5.1: Sum of Variable Length Subarrays

Model	Correct	MI	CC	SLOC
OpenAI o1	✓	88.8	4	11
GPT-4o	✓	93.6	3	10
Gemini 2.0	✓	69.0	3	8
Gemini 2.5	✓	67.8	4	10
Claude 3.7	✓	86.6	3	9

Table 5.2: Correctness and code-level metrics for the Sum of Variable Length Subarrays problem

```python
class Solution:
    def subarraySum(self, nums: List[int]) -> int:
        n = len(nums)

        prefix_sum = [0] * (n + 1)
        for i in range(n):
            prefix_sum[i+1] = prefix_sum[i] + nums[i]

        total_sum = 0
        for i in range(n):
            start = max(0, i - nums[i])
            total_sum += prefix_sum[i+1] - prefix_sum[start]

        return total_sum
```

Listing 5.2: Sum of Variable Length Subarrays - OpenAI o1 Mini

```python
from typing import List

class Solution:
    def subarraySum(self, nums: List[int]) -> int:
```

```
        total_sum = 0
        n = len(nums)

        for i in range(n):
            start = max(0, i - nums[i])
            subarray_sum = sum(nums[start:i + 1])
            total_sum += subarray_sum

        return total_sum
```

Listing 5.3: Sum of Variable Length Subarrays - GPT-4o Mini

```
class Solution:
    def subarraySum(self, nums):
        total_sum = 0
        for i in range(len(nums)):
            start = max(0, i - nums[i])
            sub_array_sum = sum(nums[start:i+1])
            total_sum += sub_array_sum
        return total_sum
```

Listing 5.4: Sum of Variable Length Subarrays - Gemini 2.0 Flash

```
from typing import List

class Solution:
    def subarraySum(self, nums: List[int]) -> int:
        n = len(nums)
        total_sum = 0

        for i in range(n):
            start = max(0, i - nums[i])
            for j in range(start, i + 1):
                total_sum += nums[j]

        return total_sum
```

Listing 5.5: Sum of Variable Length Subarrays - Gemini 2.5 Flash

```
class Solution:
    def subarraySum(self, nums: List[int]) -> int:
        n = len(nums)
        total_sum = 0

        for i in range(n):
            start = max(0, i - nums[i])
            subarray_sum = sum(nums[start:i+1])
            total_sum += subarray_sum

        return total_sum
```

Listing 5.6: Sum of Variable Length Subarrays - Claude 3.7 Sonnet

For a simple problem such as this, we can see that the metrics, such as Cyclomatic Complexity and Source Lines of Code, remain fairly similar across the different models.

I believe that the o1 solution is the best of those generated. This is because its solution introduces a tiny prefix-sum array and then computes every subarray sum in *O(1)* time. This extra setup turns the overall algorithm into a clean *O(n)* pass compared to the $O(n^2)$ of all the other solutions.

GPT-4o follows the specification almost word for word. It recomputes each subarray sum with Python's built-in `sum` on every iteration, so the complexity is $O(n^2)$. Even so, the code is compact, variables are minimal, and the function reads like a direct transcription of the English prompt.

Both Gemini variants stay close to GPT-4o's brute-force idea, yet they diverge in how they structure the loops. Gemini 2.0 adopts a single pass with `sum` similar to GPT 4o, whereas Gemini 2.5 does the inner summation manually rather than using the built-in function.

Claude 3.7 also mirrors the single-pass brute-force approach and does not attempt the prefix-sum optimisation, resulting in a very similar generation to that of GPT 4o and Gemini 2.0.

Problem 2

This is a problem which GPT-4o got wrong, but all the other models solved correctly.

- **Problem:** Unique 3-digit even numbers
- **Difficulty:** Easy
- **Topics:** Array, Hash Table, Recursion, Enumeration

```
You are given an array of digits called digits. Your task is to determine the number of
    distinct three-digit even numbers that can be formed using these digits.

Note: Each copy of a digit can only be used once per number, and there may not be leading
    zeros.

Example 1:
Input: digits = [1,2,3,4]
Output: 12

Explanation: The 12 distinct 3-digit even numbers that can be formed are 124, 132, 134,
    142, 214, 234, 312, 314, 324, 342, 412, and 432. Note that 222 cannot be formed
    because there is only 1 copy of the digit 2.

Example 2:
Input: digits = [0,2,2]
Output: 2

Explanation: The only 3-digit even numbers that can be formed are 202 and 220. Note that
    the digit 2 can be used twice because it appears twice in the array.

Example 3:
Input: digits = [6,6,6]
Output: 1

Explanation: Only 666 can be formed.

Example 4:
Input: digits = [1,3,5]
Output: 0

Explanation: No even 3-digit numbers can be formed.

Constraints:
3 <= digits.length <= 10
```

```
0 <= digits[i] <= 9
```

Listing 5.7: Problem Description - Unique 3-digit even numbers

Model	Correct	MI	CC	SLOC
OpenAI o1	✓	64.3	5	9
GPT-4o	✗	83.2	9	17
Claude 3.7	✓	82.5	12	21
Gemini 2.0	✓	85.9	10	12
Gemini 2.5	✓	67.5	7	17

Table 5.3: Correctness and code-level metrics for the Unique 3-digit Even Numbers problem

```python
class Solution:
    def totalNumbers(self, digits: List[int]) -> int:
        from itertools import permutations
        distinct_numbers = set()
        for combo in permutations(digits, 3):
            if combo[0] != 0 and combo[2] % 2 == 0:
                number = combo[0] * 100 + combo[1] * 10 + combo[2]
                distinct_numbers.add(number)
        return len(distinct_numbers)
```

Listing 5.8: Unique 3-digit Even Numbers - OpenAI o1 Mini

```python
from typing import List

class Solution:
    def totalNumbers(self, digits: List[int]) -> int:
        digit_set = set(digits)
        even_digits = [d for d in digit_set if d % 2 == 0]
        if not even_digits:
            return 0
        count = 0
        for last_digit in even_digits:
            remaining_digits = digit_set - {last_digit}
            for first_digit in remaining_digits:
                if first_digit == 0:
                    continue
                for second_digit in remaining_digits - {first_digit}:
                    count += 1
        return count
```

Listing 5.9: Unique 3-digit Even Numbers - GPT 4o Mini

```python
from typing import List

class Solution:
    def totalNumbers(self, digits: List[int]) -> int:
        count = 0
        distinct_numbers = set()

        for i in range(len(digits)):
            for j in range(len(digits)):
                for k in range(len(digits)):
```

```
                    if i != j and i != k and j != k and digits[k] % 2 == 0 and digits[i]
    != 0:
                        number = digits[i] * 100 + digits[j] * 10 + digits[k]
                        distinct_numbers.add(number)

        return len(distinct_numbers)
```

Listing 5.10: Unique 3-digit Even Numbers - Gemini 2.0 Flash

```
from typing import List
from collections import Counter

class Solution:
    def totalNumbers(self, digits: List[int]) -> int:
        count = Counter(digits)
        distinct_numbers = set()

        for i in range(100, 1000, 2):
            s_num = str(i)
            temp_count = Counter([int(d) for d in s_num])
            possible = True
            for digit, freq in temp_count.items():
                if count[digit] < freq:
                    possible = False
                    break
            if possible:
                distinct_numbers.add(i)

        return len(distinct_numbers)
```

Listing 5.11: Unique 3-digit Even Numbers - Gemini 2.5 Flash

```
class Solution:
    def totalNumbers(self, digits: List[int]) -> int:
        count = {}
        for digit in digits:
            count[digit] = count.get(digit, 0) + 1

        result = 0
        for first in range(1, 10):
            if first not in count or count[first] == 0:
                continue
            count[first] -= 1
            for second in range(0, 10):
                if second not in count or count[second] == 0:
                    continue
                count[second] -= 1
                for last in [0, 2, 4, 6, 8]:
                    if last not in count or count[last] == 0:
                        continue
                    result += 1
                count[second] += 1
            count[first] += 1
        return result
```

Listing 5.12: Unique 3-digit Even Numbers - Claude 3.7 Sonnet

In this problem, we see a greater variance in code-level metrics across the models. While all models except GPT-4o arrive at the correct solution, their paths to that outcome differ notably in terms of algorithmic design and verbosity.

OpenAI's solution is the shortest, accurate and has the lowest Cyclomatic Complexity. Its time complexity is $O(n^3)$. Although it uses the `permutations` method, which has a complexity of $n!$, the method is only used to generate 3-digit numbers, keeping the overall cost down. The solution works well, but could be made more efficient for larger values of n by generating a set of the possible final values instead of generating permutations from the input itself.

Claude 3.7 offers one of the most efficient solutions with a runtime of just $O(n)$. This is achieved by creating a digit frequency map up front. Once this preprocessing step is done, the rest of the logic runs in constant time because it only loops over a fixed number of valid solutions. Although the implementation is slightly longer than some of the others, it achieves excellent maintainability and performance.

Gemini 2.0 has a time complexity of $O(n^3)$ and essentially mirrors the logic of the o1 model. The only difference is that it explicitly uses three nested loops rather than calling the `permutations` library function. Gemini 2.5, by contrast, follows Claude more closely and has a time complexity of $O(n)$. This efficiency comes from setting up a `Counter` object and checking fixed-length candidates in the range of 100 to 999. Like Claude's submission, it benefits from predictable iteration and a straightforward correctness check.

GPT-4o also has a time complexity of $O(n^3)$, but its logic is flawed. The main issue is that it operates on a `set` of digits rather than the original list, which causes it to ignore repeated digits. For example, given the input `[2, 2, 0]`, it constructs the set `{0, 2}` and concludes that no 3-digit number can be formed, even though valid answers like 220 and 202 exist. This suggests that the model misinterpreted the problem statement, treating it as if each digit could only be used once, regardless of its frequency in the input. A correct solution would need to track digit frequencies explicitly using a list or a counter rather than just generating a set.

Problem 3

This is another problem that was answered correctly by all models except GPT-4o.

- **Problem Name:** Choose K Elements With Maximum Sum
- **Difficulty:** Medium
- **Topics:** Array, Sorting, Heap (Priority Queue)

```
You are given two integer arrays, nums1 and nums2, both of length n, along with a
     positive integer k.

For each index i from 0 to n - 1, perform the following:

Find all indices j where nums1[j] is less than nums1[i].
Choose at most k values of nums2[j] at these indices to maximize the total sum.
Return an array answer of size n, where answer[i] represents the result for the
    corresponding index i.

Example 1:
Input: nums1 = [4,2,1,5,3], nums2 = [10,20,30,40,50], k = 2
Output: [80,30,0,80,50]

Example 2:
Input: nums1 = [2,2,2,2], nums2 = [3,1,2,3], k = 1
```

```
Output: [0,0,0,0]

Constraints:
n == nums1.length == nums2.length
1 <= n <= 10^5
1 <= nums1[i], nums2[i] <= 10^6
1 <= k <= n
```

Listing 5.13: Choose K Elements With Maximum Sum

Model	Correct	MI	CC	SLOC
OpenAI o1	✓	79.9	9	27
GPT-4o	✗	81.4	8	23
Gemini 2.0	✓	60.6	7	20
Gemini 2.5	✓	61.0	7	19
Claude 3.7	✓	96.0	7	11

Table 5.4: Correctness and code-level metrics for the Choose K Elements With Maximum Sum problem

```python
class Solution:
    def findMaxSum(self, nums1, nums2, k):
        import heapq

        n = len(nums1)
        arr = [(nums1[i], nums2[i], i) for i in range(n)]
        arr.sort(key=lambda x: x[0])

        answer = [0] * n
        min_heap = []
        current_sum = 0

        i = 0
        while i < n:
            current_val = arr[i][0]
            start = i
            while i < n and arr[i][0] == current_val:
                _, val2, original_idx = arr[i]
                answer[original_idx] = current_sum
                i += 1
            for j in range(start, i):
                _, val2, _ = arr[j]
                if len(min_heap) < k:
                    heapq.heappush(min_heap, val2)
                    current_sum += val2
                else:
                    if val2 > min_heap[0]:
                        smallest = heapq.heapreplace(min_heap, val2)
                        current_sum += (val2 - smallest)

        return answer
```

Listing 5.14: Choose K Elements With Maximum Sum - OpenAI o1 Mini

```python
from typing import List
import heapq
```

```python
class Solution:
    def findMaxSum(self, nums1: List[int], nums2: List[int], k: int) -> List[int]:
        n = len(nums1)
        answer = [0] * n
        nums2_map = {}

        for index in range(n):
            if nums1[index] not in nums2_map:
                nums2_map[nums1[index]] = []
            nums2_map[nums1[index]].append(nums2[index])

        for key in nums2_map:
            nums2_map[key].sort(reverse=True)

        for i in range(n):
            current_value = nums1[i]
            total_sum = 0
            for value in sorted(nums2_map.keys()):
                if value < current_value:
                    total_sum += sum(nums2_map[value][:k])
                else:
                    break
            answer[i] = total_sum

        return answer
```

Listing 5.15: Choose K Elements With Maximum Sum - GPT 4o Mini

```python
import heapq
from typing import List

class Solution:
    def findMaxSum(self, nums1: List[int], nums2: List[int], k: int) -> List[int]:
        n = len(nums1)
        answer = [0] * n

        for i in range(n):
            eligible_indices = []
            for j in range(n):
                if nums1[j] < nums1[i]:
                    eligible_indices.append(j)

            if not eligible_indices:
                answer[i] = 0
            else:
                eligible_values = []
                for index in eligible_indices:
                    eligible_values.append(nums2[index])
                eligible_values.sort(reverse=True)
                answer[i] = sum(eligible_values[:min(k, len(eligible_values))])

        return answer
```

Listing 5.16: Choose K Elements With Maximum Sum - Gemini 2.0 Flash

```python
import heapq

class Solution:
    def findMaxSum(self, nums1: List[int], nums2: List[int], k: int) -> List[int]:
        n = len(nums1)
        answer = [0] * n
```

```
        indexed_nums1 = sorted([(nums1[i], i) for i in range(n)])

        for i in range(n):
            current_num1, original_index = indexed_nums1[i]
            candidates = []
            for j in range(i):
                prev_num1, prev_original_index = indexed_nums1[j]
                if prev_num1 < current_num1:
                    candidates.append(nums2[prev_original_index])
            candidates.sort(reverse=True)
            current_sum = 0
            for count in range(min(k, len(candidates))):
                current_sum += candidates[count]
            answer[original_index] = current_sum

        return answer
```

Listing 5.17: Choose K Elements With Maximum Sum - Gemini 2.5 Flash

```
class Solution:
    def findMaxSum(self, nums1: List[int], nums2: List[int], k: int) -> List[int]:
        n = len(nums1)
        result = [0] * n

        for i in range(n):
            valid_indices = [j for j in range(n) if nums1[j] < nums1[i]]
            if valid_indices:
                valid_values = [nums2[j] for j in valid_indices]
                valid_values.sort(reverse=True)
                result[i] = sum(valid_values[:min(k, len(valid_values))])

        return result
```

Listing 5.18: Choose K Elements With Maximum Sum - Claude 3.7 Sonnet

For this problem, the metrics are also reasonably spread out. The clear outlier is Claude 3.7 with a very short solution of only 11 Lines of Code. OpenAI o1 has the longest solution at 27 lines, while the other three models all hover around the 20-line mark. The Cyclomatic Complexity of each submission remains quite similar, ranging only from 7 to 9. Claude 3.7 has the lowest Cyclomatic Complexity, and OpenAI o1 has the highest score.

I see the o1 submission as the most efficient again. It sorts the combined data once, keeps a size-k min-heap of the best `nums2` values, and updates a running sum in place. That single structure lifts the bound to $O(n \log n)$ and avoids a full quadratic sweep. The extra Cyclomatic Complexity is easy to justify because it sharply improves the overall efficiency of the program.

Claude 3.7 chooses a simple design. For every i it scans all smaller elements, sorts them, and takes the top k. The implementation works well, but the nested loop costs $O(n^2 \log n)$, so it lags behind o1 in efficiency. Claude's code, however, boasts the highest maintainability index, likely because the solution is short and easy to read.

Both Gemini 2.0 and Gemini 2.5 follow a similar approach to Claude and have a worst-case cost to $O(n^2 \log n)$. The answers are correct, yet they trail in both efficiency and maintainability compared to the other models.

GPT-4o failed one of the test cases during LeetCode verification. This was because inside its per-index loop, it sliced the top k values for each smaller key, so it could sum more than k numbers overall, producing values that were too large. A fix for this would be collecting all eligible values into one list, sorting once, and trimming to k. This would

bring its performance in line with the Gemini variants, and would thus still fall short of the efficiency shown by o1.

Problem 4

This is a problem which only the o1 model has managed to solve correctly.

- **Problem Name:** Minimum Operations to Make Array Values Equal to K
- **Difficulty:** Easy
- **Topics:** Array, Hash Table

```
You are given an integer array nums and an integer k.

An integer h is called valid if all values in the array that are strictly greater
     than h are identical.

For example, if nums = [10, 8, 10, 8], a valid integer is h = 9 because all nums[
    i] > 9 are equal to 10, but 5 is not a valid integer.

You are allowed to perform the following operation on nums:

Select an integer h that is valid for the current values in nums.
For each index i where nums[i] > h, set nums[i] to h.
Return the minimum number of operations required to make every element in nums
    equal to k. If it is impossible to make all elements equal to k, return -1.

Example 1:
Input: nums = [5,2,5,4,5], k = 2
Output: 2

Explanation:
The operations can be performed in order using valid integers 4 and then 2.

Example 2:
Input: nums = [2,1,2], k = 2
Output: -1

Explanation:
It is impossible to make all the values equal to 2.

Example 3:
Input: nums = [9,7,5,3], k = 1
Output: 4

Explanation:
The operations can be performed using valid integers in the order 7, 5, 3, and 1.

Constraints:
1 <= nums.length <= 100
1 <= nums[i] <= 100
1 <= k <= 100
```

Listing 5.19: Minimum Operations to Make Array Values Equal to K

Model	Correct	MI	CC	SLOC
OpenAI o1	✓	97.3	5	8
GPT-4o	✗	91.3	6	13
Gemini 2.0	✗	52.1	35	76
Gemini 2.5	✗	72.5	10	20
Claude 3.7	✗	87.5	12	16

TABLE 5.5: Correctness and code-level metrics for the Minimum Operations to Make Array Values Equal to K problem. Only OpenAI o1 passed.

```python
class Solution:
    def minOperations(self, nums: List[int], k: int) -> int:
        if min(nums) < k:
            return -1

        distinct_vals = {x for x in nums if x >= k}
        distinct_vals.add(k)

        sorted_vals = sorted(distinct_vals, reverse=True)

        return len(sorted_vals) - 1
```

LISTING 5.20: Minimum Operations to Make Array Values Equal to K - OpenAI o1 Mini

```python
from typing import List

class Solution:
    def minOperations(self, nums: List[int], k: int) -> int:
        if k not in nums:
            return -1

        nums.sort()

        operations = 0
        for num in reversed(nums):
            if num > k:
                operations += 1
            elif num < k:
                break

        return operations
```

LISTING 5.21: Minimum Operations to Make Array Values Equal to K - GPT 4o Mini

```python
class Solution:
    def minOperations(self, nums, k):
        ops = 0
        while True:
            all_k = True
            for num in nums:
                if num != k:
                    all_k = False
                    break
            if all_k:
                return ops
```

```
            valid_h = []
            for h in range(1, 101):
                valid = True
                greater_than_h = []
                for num in nums:
                    if num > h:
                        greater_than_h.append(num)
                if len(greater_than_h) > 0:
                    first = greater_than_h[0]
                    for val in greater_than_h:
                        if val != first:
                            valid = False
                            break
                if valid:
                    valid_h.append(h)

            if not valid_h:
                return -1

            best_h = -1
            for h in valid_h:
                can_reach_k = True
                temp_nums = nums[:]
                for i in range(len(temp_nums)):
                    if temp_nums[i] > h:
                        temp_nums[i] = h

                valid_after_op = False
                valid_hs_after = []
                for h_after in range(1, 101):
                    valid_after = True
                    greater_than_h_after = []
                    for num in temp_nums:
                        if num > h_after:
                            greater_than_h_after.append(num)
                    if len(greater_than_h_after) > 0:
                        first = greater_than_h_after[0]
                        for val in greater_than_h_after:
                            if val != first:
                                valid_after = False
                                break
                    if valid_after:
                        valid_hs_after.append(h_after)

                found_k = False
                for h_after in valid_hs_after:
                    temp_nums2 = temp_nums[:]
                    for i in range(len(temp_nums2)):
                        if temp_nums2[i] > h_after:
                            temp_nums2[i] = h_after
                    all_k_now = True
                    for num in temp_nums2:
                        if num != k:
                            all_k_now = False
                            break
                    if all_k_now:
                        found_k = True
                if not found_k:
                    can_reach_k = False

                if can_reach_k:
```

```
                    best_h = h
                    break

            if best_h == -1:
                return -1

            for i in range(len(nums)):
                if nums[i] > best_h:
                    nums[i] = best_h
            ops += 1
```

Listing 5.22: Minimum Operations to Make Array Values Equal to K - Gemini 2.0 Flash

```
from typing import List

class Solution:
    def minOperations(self, nums: List[int], k: int) -> int:
        if any(num < k for num in nums):
            return -1

        unique_greater_than_k = sorted(list(set([num for num in nums if num > k])))

        if not unique_greater_than_k:
            return 0

        operations = 0
        current_max = unique_greater_than_k[-1]

        targets = unique_greater_than_k + [k]

        for i in range(len(targets) - 1, 0, -1):
            if current_max > targets[i-1]:
                operations += 1
                current_max = targets[i-1]
            elif current_max == targets[i-1]:
                continue
            else:
                return -1

        return operations
```

Listing 5.23: Minimum Operations to Make Array Values Equal to K - Gemini 2.5 Flash

```
class Solution:
    def minOperations(self, nums: List[int], k: int) -> int:
        if min(nums) < k:
            return -1

        if all(num == k for num in nums):
            return 0

        distinct_values = sorted(set(num for num in nums if num > k))

        operations = 0
        for h in distinct_values:
            operations += 1
            for i in range(len(nums)):
                if nums[i] > h:
```

```
                    nums[i] = h
            if all(num == k for num in nums):
                return operations

        return -1
```

Listing 5.24: Minimum Operations to Make Array Values Equal to K - Claude 3.7 Sonnet

The ideal solution for this problem, as shown on LeetCode, is relatively straightforward. You first check if any number in the array is less than k; if so, you return -1. Otherwise, you count the number of unique values strictly greater than k, which corresponds exactly to the number of plateau operations required. All five models, to varying degrees, overengineer this simple idea quite heavily.

```
class Solution:
    def minOperations(self, nums: List[int], k: int) -> int:
        st = set()
        for x in nums:
            if x < k:
                return -1
            elif x > k:
                st.add(x)
        return len(st)
```

Listing 5.25: Minimum Operations to Make Array Values Equal to K - Leetcode Solution

Compared to earlier examples, we see a greater variance in the code metrics, though most values remain within a comparable range. The outlier is Gemini 2.0, which produces an extremely convoluted solution that fails to solve the problem while also having 76 Lines of Code and a Cyclomatic Complexity of 35.

The clear standout solution is that of the o1 model. It delivers a working solution while having the lowest Cyclomatic Complexity and Lines of Code, albeit it too has a few unnecessary steps. It correctly begins with the base case check to see if any value is less than k and if so, it returns -1. It then collects all distinct values greater than or equal to k, adds k manually, and finally returns the number of distinct steps needed to reduce all values to k. This results in a correct output with a time complexity of $\mathcal{O}(n \log n)$. That said, a few refinements could simplify it further. For instance, there is no need to include values equal to k in the distinct value set and then add k if it is not already present. Changing these lines eliminates the need to subtract one at the end and simplifies the code. The descending sort done is also redundant since the set is never traversed.

Claude 3.7's solution makes a conceptual error. Like o1, it checks for invalid input at the beginning, which is good. However, instead of just counting distinct values, it simulates each reduction step by manually modifying the array and checking whether the final array contains only k. Because it fails to include k in the list of distinct values to reduce to, the final plateau never occurs. As a result, even correct inputs may return -1. The fix is simple. It is to drop the final array equality checks and just return the number of distinct values above k. The array updates themselves are unnecessary, since the task is not to mutate the array but to count the operations needed.

Gemini 2.5 suffers from a subtler bug. Like Claude, it attempts to simulate the process, building a list of target plateau values and iterating through them in reverse. However, it appends k at the end of the list, meaning the final iteration starts from k instead of the largest value above k. This is because the for loop it has defined starts from the largest

values and works its way down. This results in premature termination and an incorrect count. Simply inserting k at the start of the target list would fix this issue. Like in the case of Claude, this process is actually unnecessary, and it would be appropriate to simply count the number of distinct elements above k.

Gemini 2.0 stands out in the wrong way. The result is a 76 line solution with 35 paths that ultimately fails to pass the test cases. Its logic is difficult to read, follow, or debug, and seems drastically disproportionate to the simplicity of the task.

GPT-4o Mini also returns incorrect results. Its first check is flawed as it returns -1 if k is not present in the array, when in fact it should return -1 only if some values are less than k. The model also counts every value above k, regardless of duplicates, instead of only tracking distinct values. This causes it to overestimate the number of operations needed in arrays with repeated numbers.

5.7 Model Architectures

Based on the data gathered, it is possible to make predictions about the architecture of the underlying models. GPT-4o-mini is built for real-time multimodal use, handling text, images, and audio with fast response times [31]. That kind of optimisation seems to have hurt its performance in logic-heavy code tasks. Its answers were easy to read but often too short to be correct. It did not seem to build up reasoning in steps, which matters when solving problems that require structure or careful handling of edge cases.

Claude 3.7 Sonnet seems to have a different focus. Anthropic trains it using constitutional AI, where rule-based feedback is used to shape its behaviour [8, 9]. In practice, this made it very stable. It consistently produced clean, readable code and rarely made big mistakes. But it seems as though that stability also made it cautious. It did not explore less obvious solution paths and sometimes missed trickier edge cases. In algorithmic tasks where uncertainty plays a role, this kind of conservatism can be a limitation.

The Gemini 2.0 and 2.5 Flash models were not well suited to this kind of work. They seem to be designed for fast responses and interactive tasks, more like chat or search than deep reasoning. Google seems to reserve heavier problem-solving for its Pro models [17, 18]. In my tests, the Flash models gave shallow solutions, especially on harder questions. The jump from Gemini 2.0 to 2.5 Flash did not show any real improvement in logic or structure, which suggests the newer version was not tuned differently for code reasoning.

o1-mini stood out in terms of accuracy and how it approached these problems. Its solutions had the highest average cyclomatic complexity and the longest code overall. It was clearly willing to explore more solution paths. This is in contrast with the other models which often gave shorter, simplified answers. That made them easier to read but less reliable. This likely reflects their design focus. o1-mini appears built for tasks that require deeper logic. The others are intended for speed, adaptability, or end-user interaction, which do not always require precision in reasoning. Each model reflects a different target application domain, and the o1-mini appears to have best prioritised the kind of structured, recursive problem-solving that these types of problems require.

5.8 Conclusion

Each model shows a different balance between accuracy, complexity, and maintainability. OpenAI o1-mini stands out as the most accurate by a large margin, even though its solutions are longer and more complex. Claude 3.7 Sonnet performs lower in correctness

but delivers the cleanest and most maintainable code, making it a strong choice when readability is important.

The Gemini models are reliable but unremarkable. Gemini 2.5 Flash was expected to outperform Gemini 2.0 Flash, but the results show no meaningful improvement. Both models produce average code with low maintainability and fall behind o1 in accuracy. GPT-4o-mini has the lowest accuracy overall, but its solutions are short, simple, and easy to read.

Looking at topic-level performance, each model has its own set of weaknesses. While some overlap exists, such as difficulties with dynamic programming and bit manipulation, there is no single topic that all models struggle with. Instead, weaknesses appear to be shaped by the internal design and priorities of each model.

Therefore, to explicitly answer **RQ1**, my experiments indicate that OpenAI's o1-mini model is most suitable for smaller programming tasks.

Chapter 6

Large Problems - Methodology

This chapter will discuss the methodology used to evaluate how large language models perform on full-scale, practical programming tasks. These are more complex than the small problems from earlier, involving entire applications that resemble real world software projects. The focus is on understanding how models handle longer prompts, larger codebases, and multi-step logic across different programming environments.

I will explain what projects I have chosen, how each one was framed as a prompt, and what evaluation criteria I used to judge the outputs. I also describe the models chosen, the interactive setup inside Cursor IDE, and how I rated the quality, correctness, and usability of the generated code.

6.1 Problem Identification

While there is no universally agreed-upon definition for what is considered a "large problem", here are the general guidelines I chose to follow. The solutions to these problems should be a minimum of 150 lines of code. The codebase should also consist of several interacting components, such as Classes or Services. The problems should necessitate the use of certain frameworks or libraries, and more attention needs to be paid to the non-functional properties of the code, such as maintainability and usability. Unlike the algorithmic problems that test only pure logic and syntax logic, these larger problems aim to simulate real-world development involving more complex control flow and broader architectural considerations.

I intentionally chose a diverse set of projects across various domains and programming languages to assess the general capabilities of the models better. I settled on five unique projects that encompass multiple languages, frameworks and project types.

6.1.1 Landing Page – Frontend

Language used: JavaScript
Framework/Libraries: React

This project involves building a modern React landing page for a fictional smartphone called the NovaPhone. The purpose is to evaluate the model's ability to produce structured, component-based frontend layouts with aesthetic considerations.

```
Create a complete, responsive React landing page for a fictional smartphone product called
     the "NovaPhone".

The page should include the following sections:
```

```
1. Hero Section:
   - A bold product tagline (use placeholder text).
   - A call-to-action button (e.g., "Buy Now").
   - One large featured image of the phone.

2. Features Section:
   - Highlight 3 5  key features of the NovaPhone (e.g., battery life, AI camera,
     holographic display).
   - Use icons or placeholder images.
   - Each feature should have a title and short description.

3. Reviews Section:
   - Display 3 customer testimonials.
   - Use placeholder names and lorem ipsum text for content.
   - Each review should be in a card or box with an avatar placeholder.

4. Specs Comparison Section:
   - Include a comparison table or grid that contrasts NovaPhone with a generic competitor
     (e.g., "OtherPhone").
   - Compare technical specifications such as screen size, battery life, camera resolution
     , and performance.
   - Use static data and make the layout visually clear.

5. Footer:
   - Include links like Privacy Policy, Contact, and Social Media icons (use placeholders)
     .
   - Display brand name(Nova) and copyright.

Design Guidelines:
- Use modular React components for each section.
- Use ReactBootstrap as well as Tailwind.
- Ensure the layout is responsive and works on both desktop and mobile.
- Use clean placeholder text (lorem ipsum) and semantic HTML structure.
- Do not use any backend logic or routing  this is a single-page frontend-only project.
```

Listing 6.1: Prompt for NovaPhone Landing Page

6.1.2 Bug Tracker – Backend

Language used: Java
Framework/Libraries: Spring Boot

This project involves implementing a simple backend API for a bug tracking system using Java and Spring Boot. It tests the model's ability to structure RESTful APIs, define entity relationships, handle CRUD operations, and manage application state using a MongoDB database.

The application manages two main entities:

- **Ticket**: ID, title, description, status (open/closed), priority, tags, assignee (linked to a user)
- **User**: ID, name, role (admin or developer)

```
Create a Java Spring Boot backend for a simple bug tracking system.

Use MongoDB for persistence and include full CRUD functionality for two main models:
    Ticket and User.
```

```
Each model should have its own controller, service, and repository classes.

Model Definitions:
1. Ticket:
   - Fields: id (UUID), title, description, status (enum: OPEN/CLOSED), priority (enum),
    tags (list of strings), assignee (User reference)
2. User:
   - Fields: id (UUID), name, role (enum: ADMIN/DEVELOPER)

Define the following REST API endpoints:

1. GET /tickets                  List all tickets
2. POST /tickets                 Create a new ticket
3. GET /tickets/{id}             Get a specific ticket by ID
4. PUT /tickets/{id}             Update a ticket
5. DELETE /tickets/{id}          Delete a ticket
6. GET /users                    List all users
7. POST /users                   Create a new user
8. GET /users/{id}/tickets       List all tickets assigned to a specific user

MongoDB Details:
Username: <placeholder>
Password: <placeholder>
Schema: <placeholder>

Requirements:
- Use Spring Web, Spring Data MongoDB, and Lombok for boilerplate reduction.
- Validate input data using Javax annotations.
- Return appropriate HTTP status codes and error messages.
- Use UUIDs as IDs.
- Include basic exception handling.

The project does not require authentication or authorization.
```

Listing 6.2: Prompt for Spring Boot Bug Tracker

6.1.3 ToDo List – Full Stack

Language used: JavaScript
Framework/Libraries: Next.js

This project is a simple full-stack to-do list application built using Next.js. It demonstrates the use of both frontend and backend features in a single framework. The backend is implemented using Next.js API routes and interfaces with a lightweight JSON server for data persistence.

The application involves a single model:

- **Task**: id (number), name (string), description (string), dateAdded (date), completed (boolean)

```
Create a full-stack to-do list application using Next.js. Use a JSON server as a mock
    backend (e.g., running at http://localhost:3001/tasks).

Model: Task
- id: number (auto-incremented)
- name: string
- description: string
- dateAdded: date
```

```
- completed: boolean

Frontend:
- Create a single-page layout that lists all tasks.
- Display completed tasks at the top and incomplete tasks below.
- Show task name, description, and formatted date.
- Allow users to:
  - Mark tasks as complete/incomplete (toggle)
  - Edit task name and description
  - Delete tasks
  - Add new tasks with a name, description, and auto-set dateAdded

Backend (via JSON Server):
Define and use the following REST API endpoints:
1. GET    /tasks             Get all tasks
2. POST   /tasks             Add a new task
3. GET    /tasks/{id}        Get a task by ID
4. PUT    /tasks/{id}        Update a task (e.g., edit, toggle completion)
5. DELETE /tasks/{id}        Delete a task

Implementation Notes:
- Use Next.js API routes to wrap or proxy the JSON server calls.
- Handle all state on the frontend using React hooks.
- Ensure the UI is responsive and user-friendly.
- Bootstrap React and Tailwind to help with styling.

Do not implement authentication. Keep the application single-user and local.
```

Listing 6.3: Prompt for Full Stack To-Do App

6.1.4 Flappy Bird - Game

Language used: Python
Framework/Libraries: Pygame

This project involves recreating the classic Flappy Bird game using the Pygame library[15]. The goal is to test the model's ability to handle game loops, real-time input, collision detection, and basic physics such as gravity and velocity. To simplify graphics requirements, the bird will be represented as a triangle and the pipes as vertical rectangles.

```
Write a complete Python program using the Pygame library to create a simplified Flappy
    Bird-style game.

Game Requirements:
1. Bird Mechanics:
   - The player controls a bird represented as an upward-pointing triangle.
   - The bird should fall continuously due to gravity.
   - When the player presses the spacebar, the bird should flap (i.e., jump upward with a
    fixed velocity).
   - The bird should rotate slightly upward when flapping and downward when falling, for
    visual effect.

2. Pipes:
   - The game should continuously generate pairs of rectangular vertical pipes that move
    from right to left.
   - Each pipe pair should have a gap in between that the bird must pass through.
   - The gap size and vertical position should be randomized within reasonable bounds.

3. Collision Detection:
   - The game should detect collisions between the bird and any pipe.
```

```
   - The game should also detect when the bird hits the top or bottom of the screen.
   - On collision, the game should end and show a "Game Over" message.

4. Scoring:
   - Each time the bird successfully passes through a set of pipes, the score should
     increase by 1.
   - The current score should be displayed at the top of the screen during gameplay.

5. Game Loop and Restart:
   - The game should run at a consistent frame rate (e.g., 60 FPS).
   - Upon game over, the player should be able to press a key (e.g., Enter) to restart the
     game.

6. Graphics:
   - Use basic Pygame primitives only:
     - Draw the bird as a triangle.
     - Draw pipes as rectangles.
     - Have a solid blue background as the sky
     - Use simple text for score and messages.
   - Avoid importing any external assets (images, sounds, or fonts).

Generate the code in the current directory and use venv for this folder.
```

Listing 6.4: Prompt for Flappy Bird Game

6.1.5 Pomodoro Tracker – CLI Application

Language used: C#
Framework/Libraries: .NET Core (Console)

This project involves building a command-line Pomodoro timer application using C#. The goal is to assess the model's ability to build time-based systems, manage user input, and structure clean, modular logic for a non-GUI application. This also tests the model's ability to use platform-specific constructs such as timers, loops, and file I/O in the .net ecosystem.

The Pomodoro technique involves 25-minute focused work sessions followed by short breaks[48]. Users track their tasks and progress using a timer.

```
Create a C# .NET Core console application that functions as a Pomodoro timer and task
    tracker.

Requirements:

1. Core Timer:
   - A Pomodoro consists of 25 minutes of focused work followed by a 5-minute break.
   - After every 4 Pomodoros, the break should be 15 minutes instead.
   - The timer should be accurate and display time remaining.
   - Provide a clear command-line interface for starting/stopping sessions.

2. Task Tracking:
   - Allow the user to input a task name before starting each Pomodoro.
   - Log each completed Pomodoro along with the associated task.
   - Store this log in a local file (e.g., JSON or CSV).

3. Session Summary:
   - Display a session summary at the end of each Pomodoro:
     - Task name
     - Time completed
     - Total Pomodoros completed so far today
```

```
4. Commands:
   - Display a main menu with available commands:
     - Start a Pomodoro
     - View today's log
     - Exit

5. Additional Guidelines:
   - Use C# best practices: methods, classes, and appropriate encapsulation.
   - Avoid third-party libraries unless absolutely necessary.
   - Use basic `System.Timers`, `DateTime`, or `async/await` patterns for timing.
   - Ensure the app is easy to run from a terminal or shell.
```

Listing 6.5: Prompt for Pomodoro CLI Tracker

6.1.6 Diversity of DataSet

I believe that the dataset I have chosen is sufficiently varied and covers a wide array of realistic programming use cases. All the programming languages tested, including JavaScript, Java, Python, and C#, appear in the top 10 most-used languages in the Stack Overflow Developer Survey 2025. This confirms their continued relevance in both industry and education. The web frameworks selected, React and Next.js, also rank highly in the frameworks category [56].

The applications I chose to build include a frontend landing page, a backend API, a full-stack app, a game, and a command-line tool. These represent common categories of software that developers regularly work on [58, 2]. They frequently appear in coding bootcamps, university coursework, and real-world software teams. Together, they span a broad range of domains including user interfaces, API construction and system utilities.

6.2 Evaluation Metrics

To evaluate the quality of code generated by language models across the projects above, I define a concise set of three core evaluation metrics:

- Functional Completeness
- Maintainability
- Ease of Prompting

Each metric is rated on a scale of **1 to 5 stars**, where:

- ★ (1 star): Very poor or unacceptable
- ★★ (2 stars): Basic, significant issues
- ★★★ (3 stars): Functional but with limitations
- ★★★★ (4 stars): Solid with minor flaws
- ★★★★★ (5 stars): Excellent

Metric	Description	Evaluation Method
Functional Completeness	Assess whether the generated solution behaves as specified in the task description and correctly implements all required functionality.	Evaluation is based on manual testing of each solution against the original specification.
Maintainability	Assesses code structure, readability, and modularity.	Quantitatively assessed using **SonarQube**. Metrics are scored from A to E and the average score of all 3 metrics is mapped to a 1–5 star score.
Ease of Prompting	Measures how easy it was to obtain the desired solution from the model, including prompt iterations, need for clarification, and overengineering.	Based on my prompting experience per task.

TABLE 6.1: Evaluation rubric used to assess the generated solutions.

This system supports both **qualitative** observations and **quantitative** comparison across the diverse project types, from backends and games to frontend apps and full-stack systems.

The combination of these three metrics offers a comprehensive and balanced perspective on model performance in the context of large-scale software tasks. Functional completeness ensures that the output meets the specification, but this alone does not tell us whether the code is maintainable or adheres to good coding practices. Maintainability, measured through SonarQube's static analysis, introduces an important structural dimension. SonarQube identifies code smells, complexity issues, and design flaws, producing standardised results that are reproducible and not subject to personal judgment [52]. This adds consistency and objectivity to the evaluation. Ease of prompting captures the level of effort required to reach a working solution. This reflects the real-world value of LLMs as collaborative tools, where models that demand fewer corrections and less clarification are far more practical. When taken together, these metrics allow for a detailed comparison that incorporates both the quality of the final code and the usability of the model as a development assistant.

6.3 Cursor IDE & Models Tested

To interface with the LLMs, I used the **Cursor IDE**, a fork of Visual Studio Code that integrates seamless support for multiple large language models (LLMs) via a conversational interface [13]. Cursor was chosen for its streamlined prompting experience and agentic features.

Unlike other chat-based interfaces(like Github Copilot), Cursor operates directly within the code editor, enabling models to make in place edits, suggest refactors, and work across multi-file projects. This aligns closely with the goals of this chapter, which focuses on generating and evaluating complete, standalone software systems. Cursor's ability to access file context and offer proactive suggestions makes it well-suited for studying LLM performance on longer, structured prompts and full-stack projects. Its agentic mode also allows

me to have a hands-off approach and let me test the models independently of any human intervention.

Due to the time-intensive nature of evaluating large software tasks, I chose to limit the scope to one flagship model from each provider. Rather than covering multiple models of each provider, the aim was to explore how each vendor's most capable model performs on realistic, end-to-end programming tasks in greater depth.

The following models were selected based on data collected from the previous experiments:

- **OpenAI o4-mini**: The o1 model significantly outperformed GPT-4o in my preliminary trials. It produced more accurate solutions than any of the models. However, this model does not support agentic behaviour within Cursor. As such, the newer o4 model was used in its place for these experiments.
- **Gemini 2.5 Pro**: Both Gemini 2.0 and Gemini 2.5 demonstrated similar overall performance. I selected Gemini 2.5 as it represents the most recent generation available from Google. In contrast the Flash model used in the previous experiments, the pro Model will be used here.
- **Claude 3.7 Sonnet**: Claude's latest model at the time of writing, Sonnet 3.7, showed to generate reasonably accurate solutions with excellent maintainability for small problems.

For each project, I began by using the initial prompts provided in the project descriptions earlier in this chapter. From there, I interacted with the model iteratively within Cursor, issuing follow-up prompts as needed to refine, fix, or extend the solution. This process continued until the solution met the specification or it became clear that the model could not complete the task satisfactorily. My experience during this interactive prompting process forms the basis of the 'Ease of Prompting' score in the evaluation rubric.

6.4 Conclusion

I designed a set of larger and more realistic software projects to test how well language models perform when building full applications instead of small isolated functions. These tasks were selected to cover different domains and technologies such as frontend development, backend systems, full stack development, games and command line tools. Each project was introduced using a carefully crafted prompt that mirrors what a developer might ask for in a real-world scenario.

To evaluate the models, I used three key metrics, which were functional completeness, maintainability, and ease of prompting. Each was scored using a five star scale that supports both comparison and subjective feedback. The experiments were carried out using the Cursor IDE, which allows models to work directly in the code editor and respond to follow-up instructions. This setup offered a realistic environment to understand how models behave when solving longer multi step software problems.

Chapter 7

Large Problems – Results

This chapter presents the results of the five large-scale software development tasks introduced earlier. For each task, I evaluate the outputs produced by the models using the defined criteria of functional correctness, code quality, and ease of prompting. The analysis is organised by problem type, offering a detailed examination of how each model approached the implementation, how much prompting was required, and what issues or design patterns emerged. Where applicable, static analysis reports and interface screenshots are included to illustrate the strengths and limitations of the generated solutions.

7.1 CLI Application

7.1.1 OpenAI o4

Evaluation Metrics:

- Functional Correctness: ★★★★★ (5 stars)
- Code Quality: ★★★★★ (5 stars)
- Ease of Prompting: ★★★★★ (5 stars)

The solution generated by OpenAI o4 met all functional requirements on the first attempt. The implementation was delivered as a single `Program.cs` file, which, while monolithic, proved effective for a task of this scale. The program offered a menu-driven interface enabling three operations, starting a Pomodoro timer, displaying past sessions, and exiting the application. Data was stored locally in JSON format and handled without error.

No violations were reported by SonarQube in maintainability, reliability, or security, resulting in top ratings across these categories. The code was relatively compact, comprising 178 lines in total. Since the initial output already met the expected criteria, no follow-up interaction with the model was required.

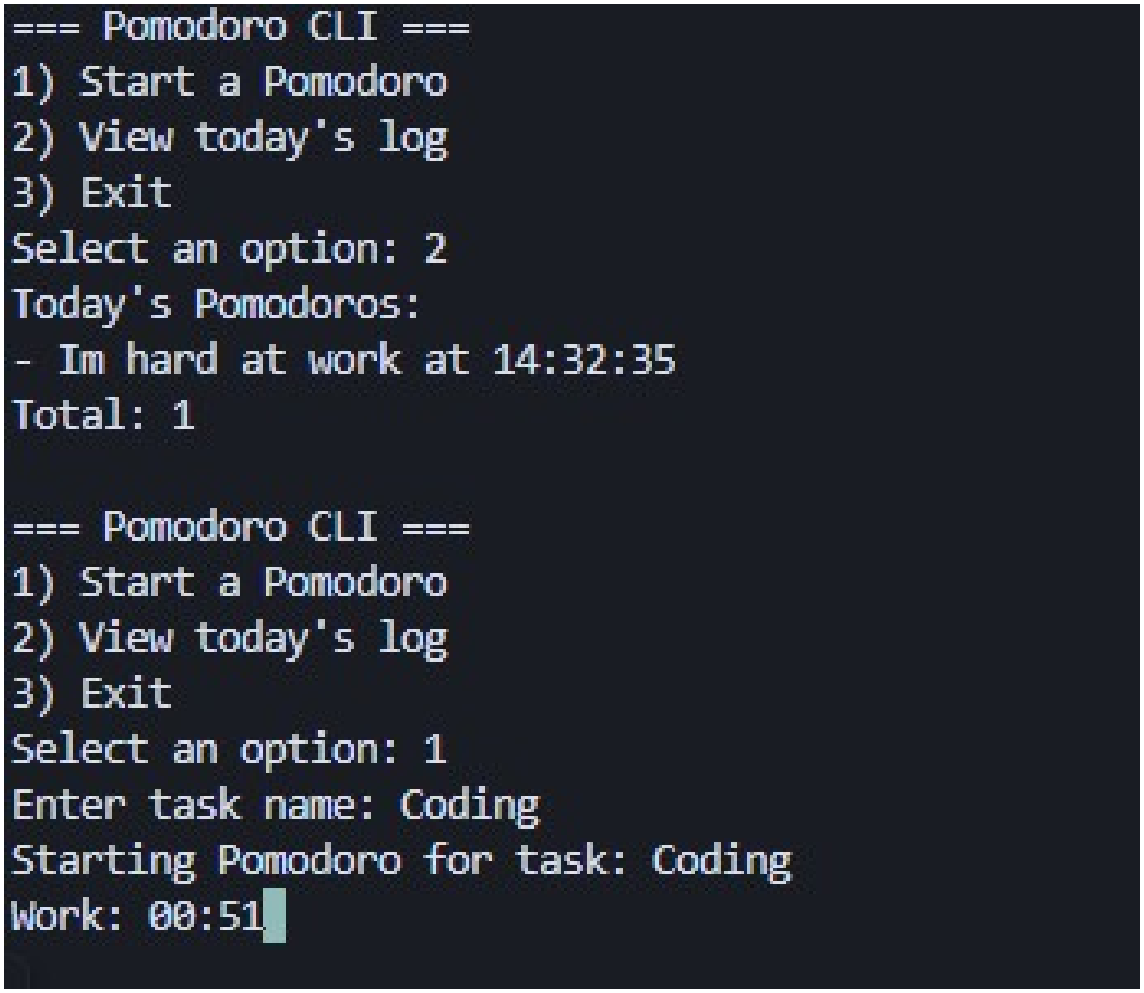

Figure 7.1: CLI Application (OpenAI o4)

Aggregate Score: 15/15

7.1.2 Claude 3.7 Sonnet

Evaluation Metrics:

- Functional Correctness: ★★★★★ (5 stars)
- Code Quality: ★★★★★ (5 stars)
- Ease of Prompting: ★★★★★ (5 stars)

Claude 3.7 Sonnet also satisfied the full specification of the CLI Pomodoro Tracker. Its implementation exhibited a clearer modular structure compared to o4, with separate files used for managing time tracking and session logging. This separation of concerns supported greater readability and may offer better maintainability for future extensions.

SonarQube did not identify any code issues, and the solution was assigned top ratings in all assessment dimensions. The total code length reached 436 lines, which appears attributable to the added abstraction layers rather than inefficiency. The application was generated from a single prompt without requiring corrections or restarts.

Aggregate Score: 15/15

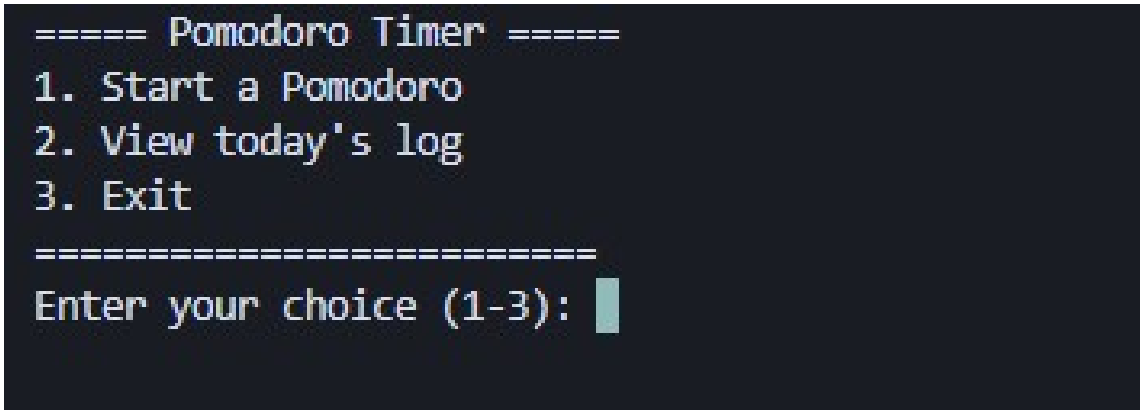

FIGURE 7.2: CLI Application - Menu (Claude 3.7)

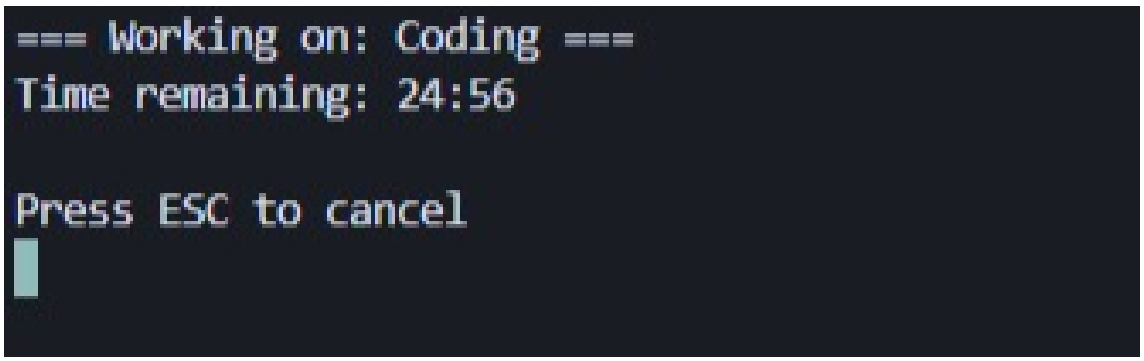

FIGURE 7.3: CLI Application - Timer (Claude 3.7)

7.1.3 Gemini 2.5

Evaluation Metrics:

- Functional Correctness: ★★★★ (4 stars)
- Code Quality: ★★★★★ (5 stars)
- Ease of Prompting: ★★★ (3 stars)

Gemini 2.5 was able to produce a functioning application, though some usability concerns emerged. Rather than updating the countdown timer in place, the program printed a new line every second, which led to an unnecessarily cluttered display. While the core logic worked as intended, this design choice reduced the practical usability of the output.

The generation process also required more effort. The initial version contained runtime errors, and two follow-up prompts did not resolve them. A fresh session was eventually needed to obtain a working implementation. The final code generated passed all static analysis checks, and SonarQube reported A ratings across maintainability, reliability, and security. The total code length was 234 lines.

Aggregate Score: 12/15

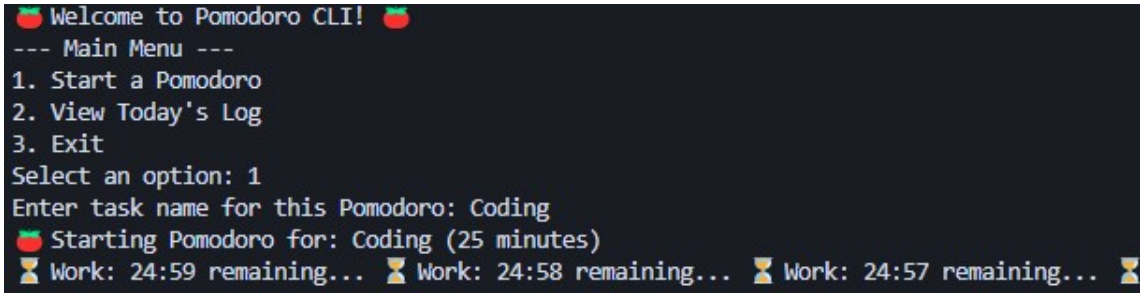

FIGURE 7.4: CLI Application (Gemini 2.5)

7.1.4 Model Comparison

All three models succeeded in generating a working CLI Pomodoro Tracker. The solutions from OpenAI o4 and Claude 3.7 Sonnet received maximum scores based on the evaluation criteria, though they differed in structure and design priorities. OpenAI o4 produced a compact single-file implementation that was easy to evaluate, whereas Claude's output employed a more modular approach that could prove advantageous for maintainability. Gemini 2.5 required more interaction to resolve initial errors and exhibited less refined output formatting, yet it ultimately delivered a correct and standards-compliant program.

7.2 Frontend – Flappy Bird Game

7.2.1 OpenAI o4

Evaluation Metrics:

- Functional Correctness: ★★★★★ (5 stars)
- Code Quality: ★★★★★ (5 stars)
- Ease of Prompting: ★★★★ (4 stars)

The version generated by OpenAI o4 included all core mechanics required for a playable Flappy Bird-style game. User input, gravity, collision handling, and scoring were implemented in a manner consistent with the task definition. The application handled edge conditions, such as off-screen movement, without errors.

The code was implemented in a single Python file and comprised 208 lines. SonarQube reported no issues related to reliability or security, though it did flag one maintainability concern due to the cognitive complexity of the main function. The initial prompt output contained a formatting issue with the "Game Over" message, which lacked contrast and displayed unevenly. A second prompt corrected this successfully.

Aggregate Score: 14/15

FIGURE 7.5: Flappy Bird Game (OpenAI o4)

7.2.2 Claude 3.7 Sonnet

Evaluation Metrics:

- Functional Correctness: ★★★★★ (5 stars)
- Code Quality: ★★★★★ (5 stars)
- Ease of Prompting: ★★★★★ (5 stars)

Claude 3.7 Sonnet produced a version of the game that met the functional requirements and also demonstrated a consistent and well-structured visual layout. All gameplay elements, such as jumping, obstacle interaction, collision logic, and scoring were handled accurately. The interface used appropriate colour choices and layout spacing, contributing to a more coherent visual experience.

The entire solution was written in a single file containing 230 lines of code. SonarQube returned A grades for maintainability, reliability, and security. Two functions were noted for elevated cognitive complexity, though these did not materially affect overall structure or clarity. No further prompting was required beyond the initial instruction.

Aggregate Score: 15/15

FIGURE 7.6: Flappy Bird Game (Claude 3.7)

7.2.3 Gemini 2.5

Evaluation Metrics:

- Functional Correctness: ★★★★ (4 stars)
- Code Quality: ★★★★★ (5 stars)
- Ease of Prompting: ★★★ (3 stars)

Gemini 2.5 implemented the fundamental game logic, including user input handling, pipe generation, and scoring. However, the overall visual composition was less polished. The "Game Over" screen remained oversized and poorly aligned, despite multiple follow-up prompts. These presentation issues persisted even after the logic had been corrected.

SonarQube assigned A ratings across all categories and flagged only one maintainability concern. The prompting process was more iterative than with the other systems, and the final result, while functional, required greater user intervention.

Aggregate Score: 12/15

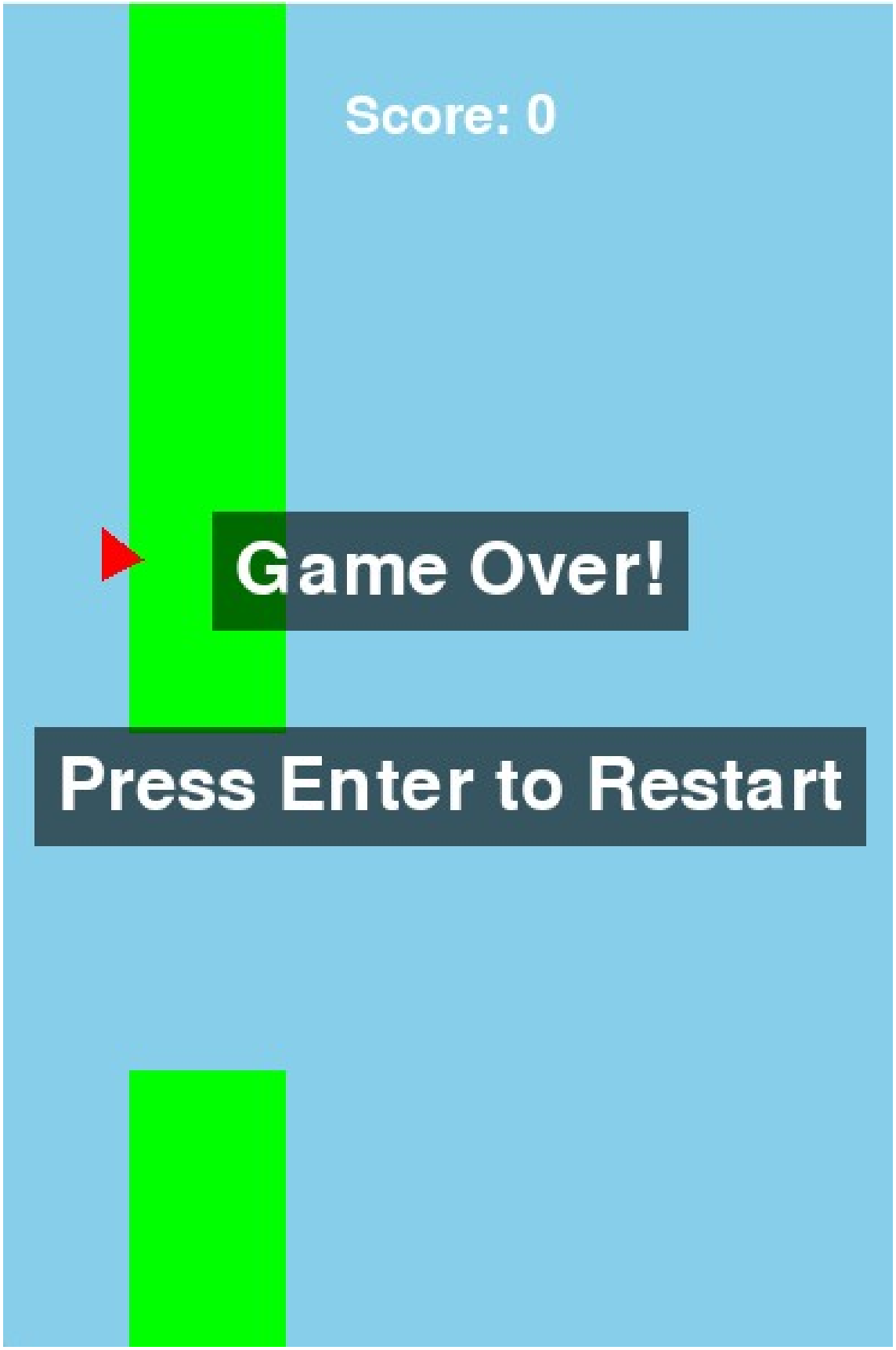

FIGURE 7.7: Flappy Bird Game (Gemini 2.5)

7.2.4 Model Comparison

Each model succeeded in producing a working version of the Flappy Bird game, though the quality and stability of the outputs varied. Claude 3.7 Sonnet delivered the most refined implementation, both in terms of visual presentation and prompt efficiency. OpenAI o4 also met the task requirements, requiring only minimal clarification to resolve a formatting issue. Gemini 2.5 showed competence in game logic but encountered recurring challenges in layout rendering and required more interaction to reach a stable output.

7.3 React Front-End

7.3.1 OpenAI o4

Evaluation Metrics:

- Functional Correctness: ★★★★ (4 stars)
- Code Quality: ★★★★★ (5 stars)
- Ease of Prompting: ★★★★ (4 stars)

The solution produced by OpenAI o4 included all required sections and met the basic layout and functionality criteria. The resulting interface was responsive and rendered effectively across screen sizes, including mobile. While structurally complete, the visual styling lacked emphasis in certain areas. Text accenting and visual hierarchy were minimal, and placeholder images were inserted as plain boxes with dimensions labeled, which reduced the perceived polish of the design.

SonarQube reported A ratings across maintainability, reliability, and security, with no major issues flagged. A single maintainability concern related to the use of array indices as keys was noted. The total codebase comprised 293 lines, written in a single pass.

Prompting required revision after the first attempt. The initial version did not render the placeholder images correctly as the preview merely displayed the image alt text, which distorted the layout. After a follow-up prompt, this issue was resolved.

Aggregate Score: 13/15

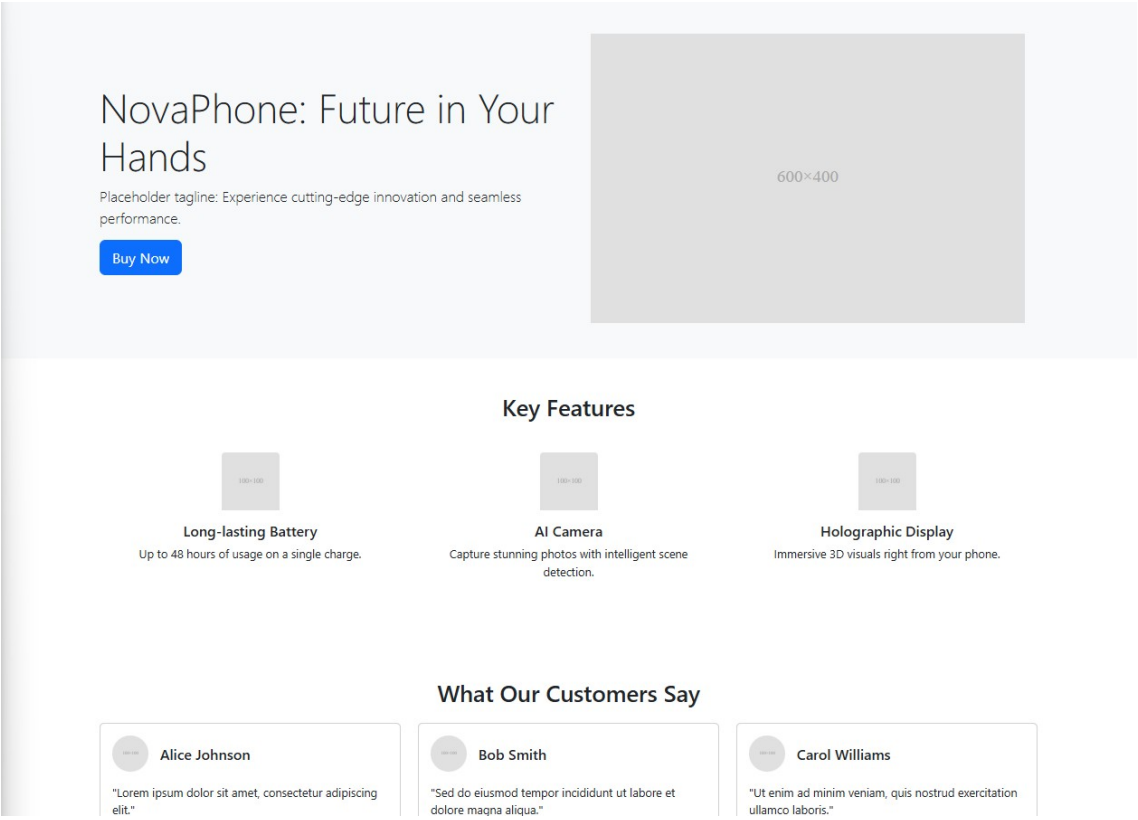

FIGURE 7.8: Hero and Features (OpenAI o4)

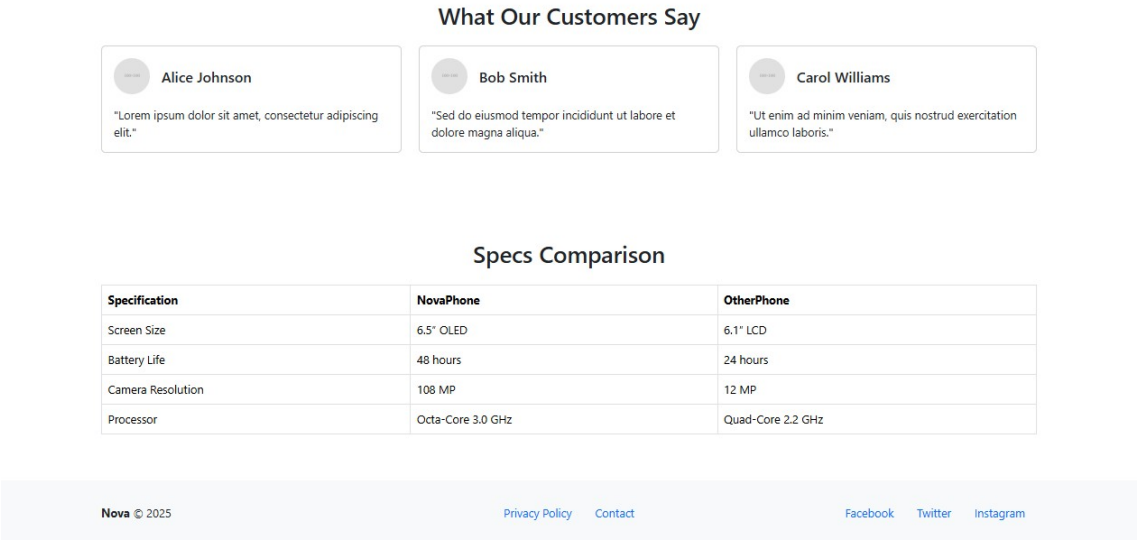

FIGURE 7.9: Specification and Testimonials (OpenAI o4)

7.3.2 Claude 3.7 Sonnet

Evaluation Metrics:

- Functional Correctness: ★★★★★ (5 stars)

- Code Quality: ★★★★ (4 stars)
- Ease of Prompting: ★★★★ (4 stars)

Claude 3.7 Sonnet produced a visually well-structured and aesthetically refined version of the page. All sections specified in the task prompt were correctly implemented, and the layout maintained logical UI flow across devices. The design featured hover states, card shadows, and appropriate text styling, contributing to a coherent user experience. Notably, the model generated an inline SVG asset for a phone graphic and retrieved example profile images for testimonial sections. Icon usage was clear, and visual contrast was well-managed.

SonarQube reported A grades for maintainability and security but issued a B for reliability. This was attributed to improper formatting of footer hyperlinks, which did not meet web accessibility or semantic standards. The solution spanned 693 lines of code.

An issue emerged during the first generation related to Tailwind CSS and its setup. A follow-up prompt was used to instruct the model to replace Tailwind with Bootstrap React, after which the output aligned with the expected requirements.

Aggregate Score: 13/15

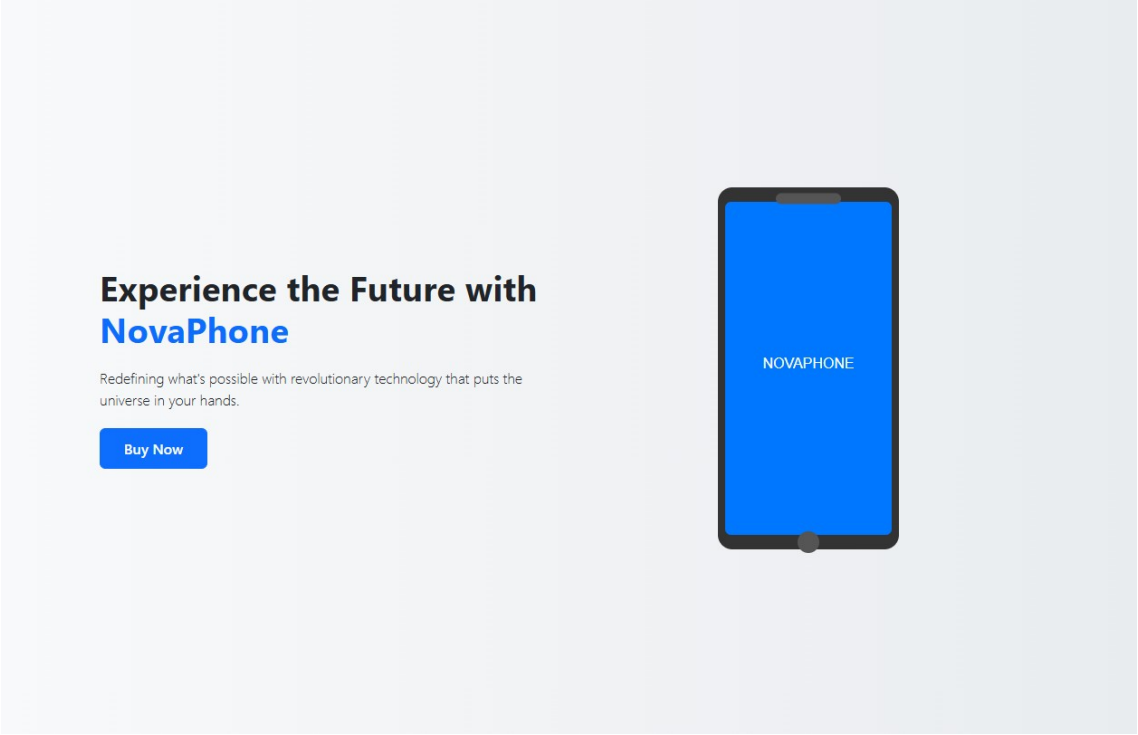

FIGURE 7.10: Hero and Features (Claude 3.7 Sonnet)

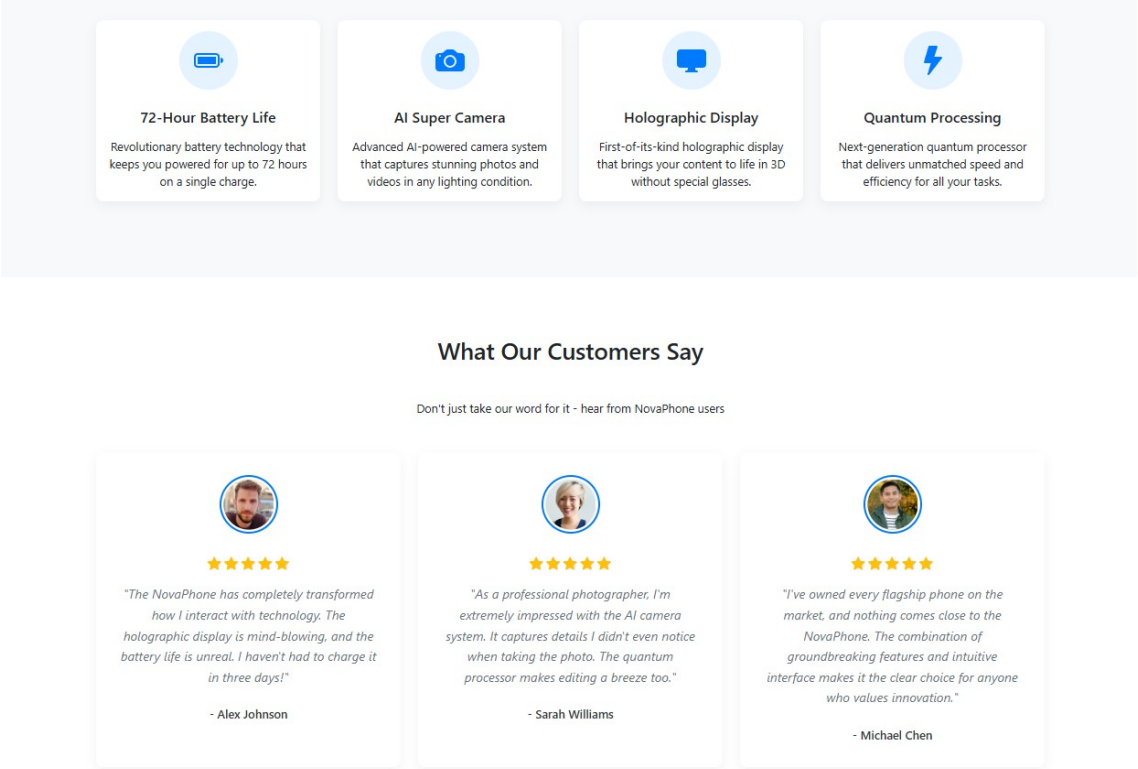

FIGURE 7.11: Specification and Testimonials (Claude 3.7 Sonnet)

7.3.3 Gemini 2.5

Evaluation Metrics:

- Functional Correctness: ★★★ (3 stars)
- Code Quality: ★★★★★ (5 stars)
- Ease of Prompting: ★★★★ (4 stars)

Gemini 2.5 fulfilled the structural requirements of the task by implementing all specified sections, including the hero, specifications, and testimonial blocks. However, the user experience was less refined. The design lacked clear accenting, and the page appeared visually flat, with minimal spacing, contrast, or shadow effects. The placeholder visuals were generic and did not contribute meaningfully to the layout.

SonarQube detected one maintainability issue related to the use of arrays as index keys. No issues were found in reliability or security. The codebase consisted of 371 lines.

Prompting required multiple iterations. The initial response omitted the requested placeholder images, and a follow-up prompt failed to resolve the issue. Only after explicitly requesting SVG-based placeholders in a third prompt did the model produce the expected result.

Aggregate Score: 12/15

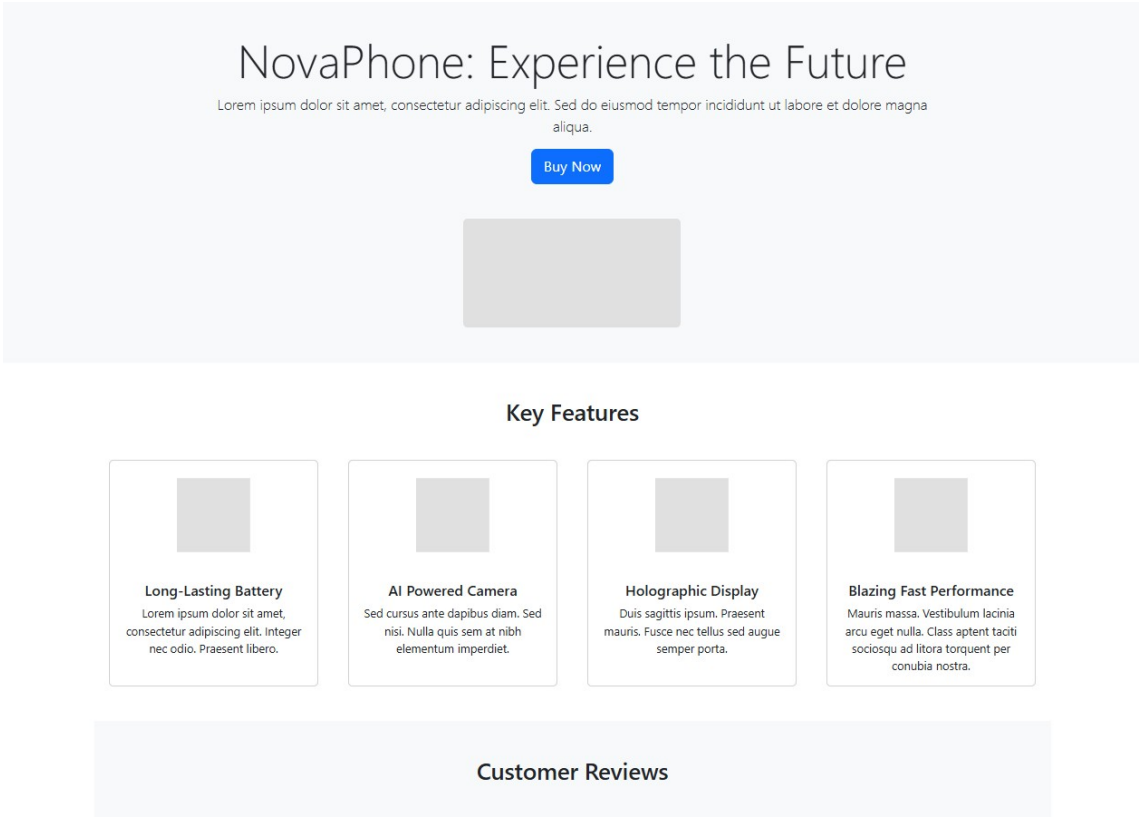

FIGURE 7.12: Hero and Features (Gemini 2.5)

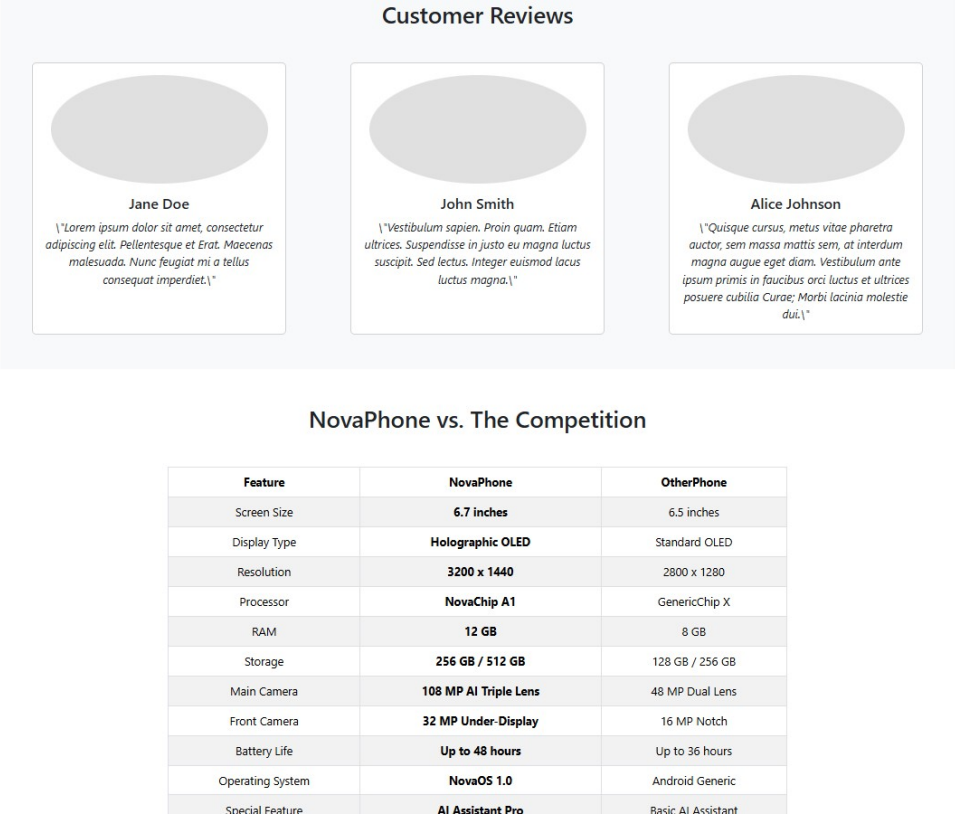

FIGURE 7.13: Specification and Testimonials (Gemini 2.5)

7.3.4 Model Comparison

All three models were able to generate a functional React front-end that followed the structural requirements of the prompt. OpenAI o4 provided a compact and responsive solution that performed well across devices but lacked visual emphasis and polish. Claude 3.7 Sonnet produced a more refined interface with stronger styling and layout decisions. Gemini 2.5 covered the required sections but fell short in terms of visual coherence and overall design quality, requiring multiple rounds of clarification to meet the expected standard.

7.4 Spring Boot Backend

7.4.1 OpenAI o4

Evaluation Metrics:

- Functional Completeness: ★★★★★ (5 stars)
- Code Quality: ★★★★ (4 stars)
- Ease of Prompting: ★★★★★ (5 stars)

OpenAI o4 generated a complete Spring Boot backend that fulfilled all functional requirements. The specified REST endpoints for both Ticket and User models were implemented, and the logic for user-specific ticket retrieval was handled correctly. CRUD operations functioned without issue, and the entity relationships were represented appropriately.

SonarQube flagged a single maintainability concern in the service layer and thus gave an A grade for that criterion. The security and reliability scores were a B. The solution comprised 415 lines of code.

Prompting was efficient. Although the first output contained a syntax error, upon a second prompt with that error included, the final implementation compiled successfully and ran as expected without further refinement.

Aggregate Score: 14/15

7.4.2 Claude 3.7 Sonnet

Evaluation Metrics:

- Functional Completeness: ★★★★★ (5 stars)
- Code Quality: ★★★★★ (5 stars)
- Ease of Prompting: ★★★★ (4 stars)

Claude 3.7 Sonnet produced a robust and well-structured backend that met all the specified functional requirements. Endpoints were correctly defined for both entities, and the system supported user-specific ticket listings. The design showed a clear separation of concerns, making use of service abstractions, DTOs, and exception handlers.

SonarQube reported no violations in any category. The project received A ratings for maintainability, reliability, and security. The implementation included validation annotations, comprehensive error messages, and clean controller logic. At 739 lines, it was the longest submission, largely due to additional helper classes and structured error handling.

The initial code failed to compile due to an issue with the MongoDB connection. Once prompted again with the error, the application ran without further problems.

Aggregate Score: 14/15

7.4.3 Gemini 2.5

Evaluation Metrics:

- Functional Completeness: ★★★★★ (5 stars)
- Code Quality: ★★★★ (4 stars)
- Ease of Prompting: ★★★★ (4 stars)

Gemini 2.5 successfully implemented all endpoints required for the bug tracking system. Both the `Ticket` and `User` entities were supported, and request handling for user-specific ticket queries was implemented as described. The overall logic adhered to RESTful design, and CRUD operations were operational.

SonarQube issued a B rating for reliability due to a lack of null checks and unchecked use of optional values. Maintainability and security were rated A. The sampled LOC totalled 581 lines.

Prompting required two iterations. The first version lacked field-level validation. Only after explicitly specifying the need for input validation did the model include annotations for the fields. The final output ran without further issues.

Aggregate Score: 13/15

7.4.4 Model Comparison

All three models successfully generated a functional Spring Boot backend that fulfilled the core requirements of the bug tracking system. OpenAI o4 and Claude 3.7 Sonnet delivered complete and well-structured implementations, with slight differences in style and complexity. Claude produced the most modular design, incorporating layered exception handling and DTOs, while OpenAI offered a slightly more concise implementation with minor maintainability concerns. Gemini 2.5 also achieved reasonable functional completeness but demonstrated lower reliability due to omitted validation and limited defensive coding. Prompting requirements were moderate across all three models, with only minor revisions needed to address compilation or configuration issues.

7.5 Full-Stack

7.5.1 OpenAI o4

Evaluation Metrics:

- Functional Correctness: ★★★★ (4 stars)
- Code Quality: ★★★★ (4 stars)
- Ease of Prompting: ★★★★★ (5 stars)

OpenAI o4 delivered a complete full-stack to-do list application that met the required functionality. Users were able to add, edit, and toggle task completion status as expected. However, the interface design lacked refinement. Elements such as buttons and form inputs were placed too close together, and there was insufficient spacing, which affected usability and visual clarity.

SonarQube reported a B for reliability, indicating the presence of a medium-risk issue related to unused reactive state updates. The solution received A grades in security and maintainability. At 183 lines, this was the shortest implementation across the three models.

Prompting was straightforward. The initial prompt produced a fully functional application that required no additional clarification.

Aggregate Score: 13/15

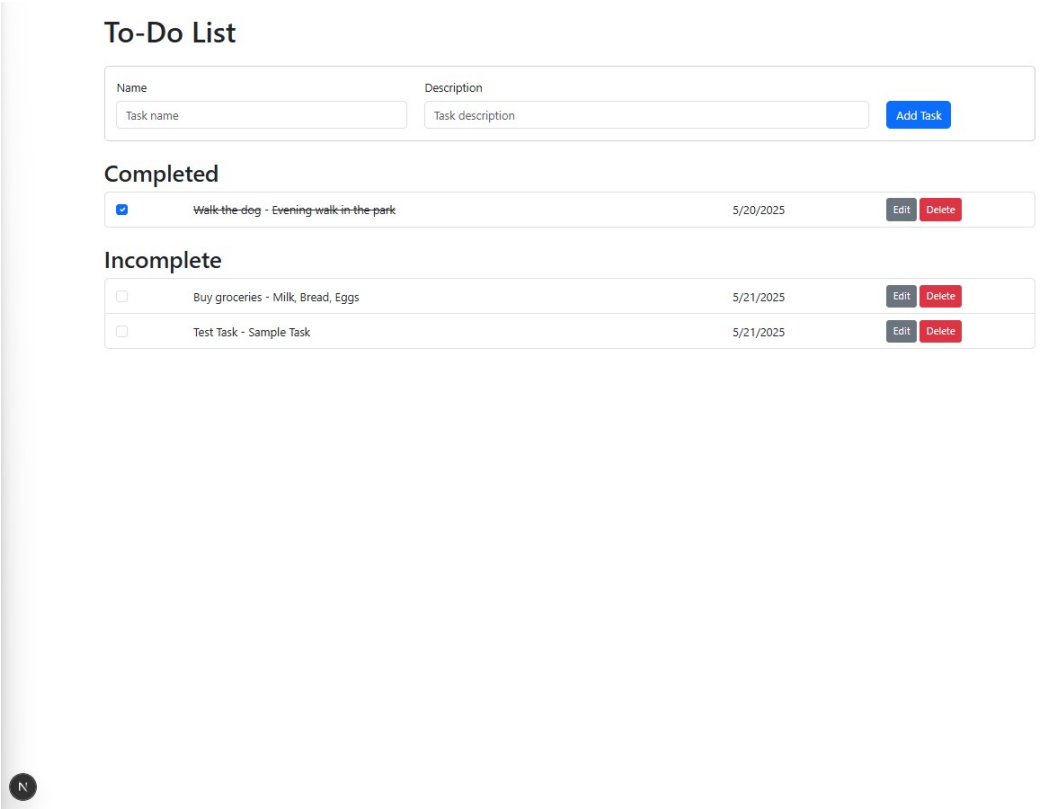

FIGURE 7.14: Main Page (OpenAI o4)

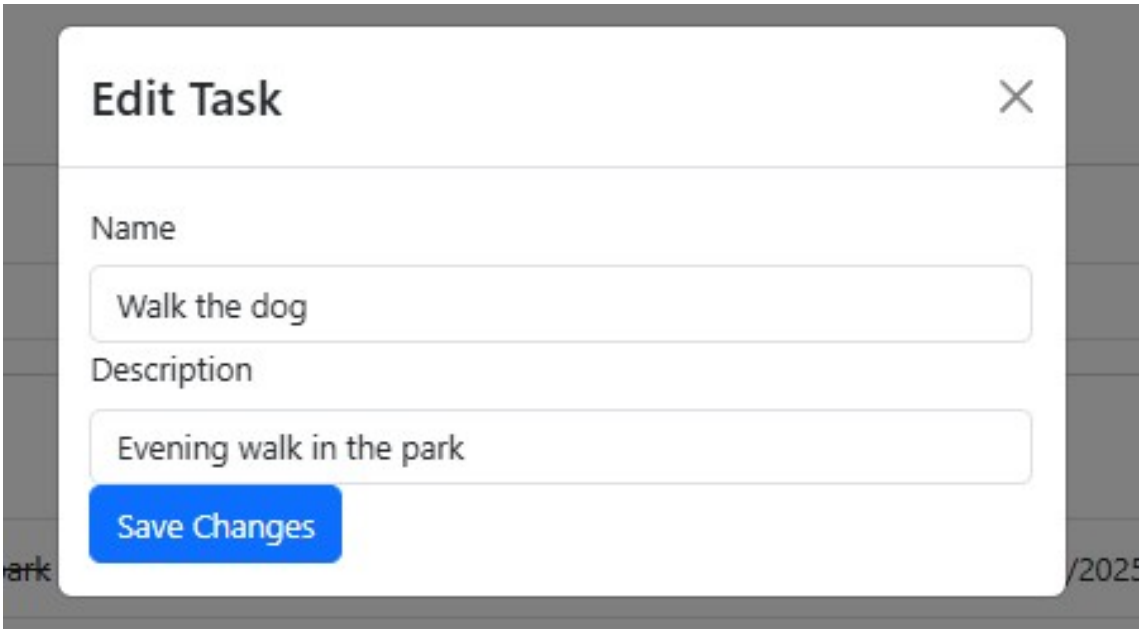

FIGURE 7.15: Edit View (OpenAI o4)

7.5.2 Claude 3.7 Sonnet

Evaluation Metrics:

- Functional Correctness: ★★★★★ (5 stars)

- Code Quality: ★★★★★ (5 stars)
- Ease of Prompting: ★★★★★ (5 stars)

Claude 3.7 Sonnet produced a fully functional and visually refined full-stack application. The front end allowed for item creation, editing, and completion tracking, with strikethrough styling applied to completed tasks. The interface was responsive, clean, and accessible, with well-balanced use of colour and spacing.

SonarQube assigned A grades across all categories. Four maintainability suggestions were flagged, including excessive use of ternary operators and missing prop immutability in some components, though none of these impacted the overall grade. At 808 lines, this was the most extensive implementation generated.

The application was generated in a single prompt, with no additional interaction required. The output was complete, cleanly structured, and ready to run.

Aggregate Score: 15/15

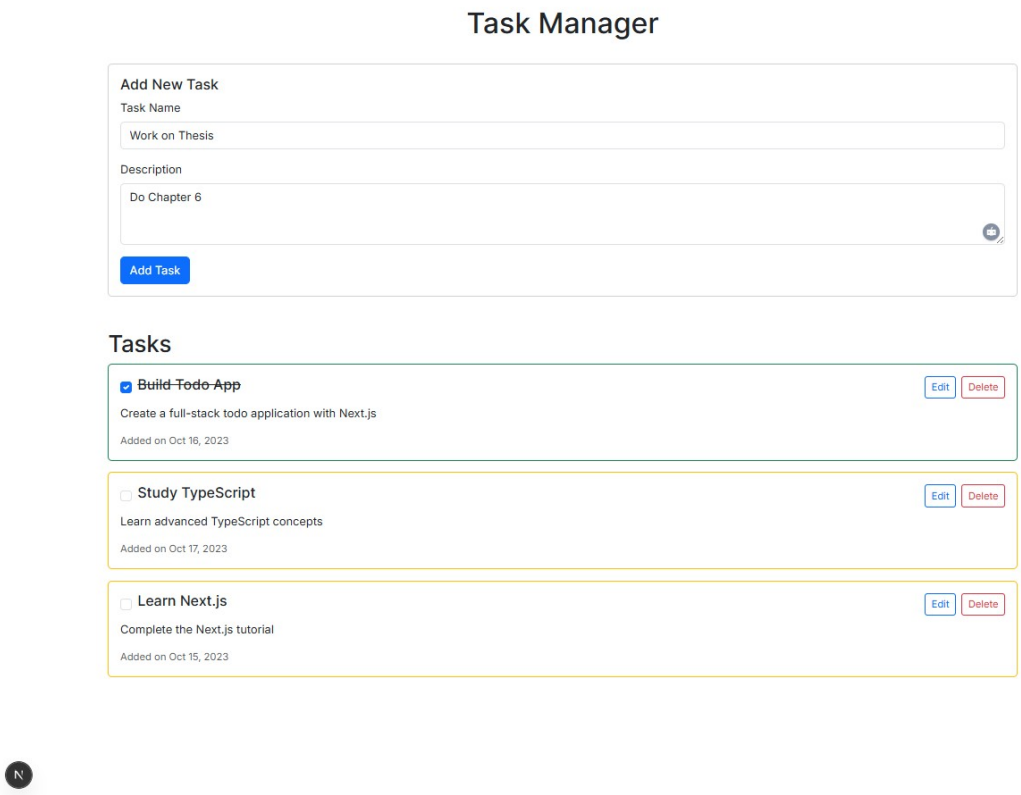

FIGURE 7.16: Main Page (Claude 3.7 Sonnet)

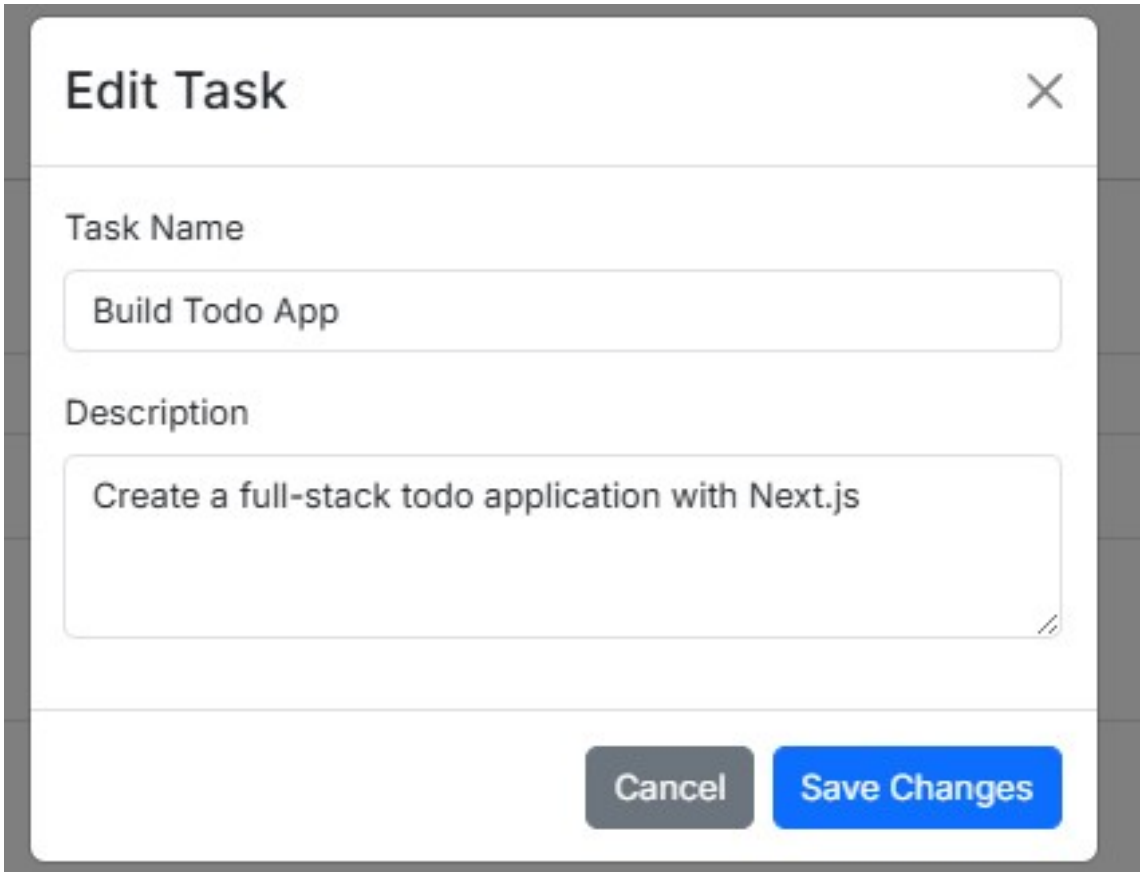

FIGURE 7.17: Edit Page (Claude 3.7 Sonnet)

7.5.3 Gemini 2.5

Evaluation Metrics:

- Functional Correctness: ★★★ (3 stars)
- Code Quality: ★★★ (3 stars)
- Ease of Prompting: ★★★ (3 stars)

Gemini 2.5 implemented core functionality such as task creation, viewing, and marking as completed. However, the application suffered from a major flaw. Once a task was marked as completed, the content disappeared from the interface, leaving behind a blank card. This significantly undermined usability, as completed tasks were no longer identifiable.

SonarQube gave A grades for maintainability and security but rated the solution C for reliability. This was due to event listeners attached to non interactive elements and inconsistent state handling in the frontend. The final solution consisted of 622 lines of code.

Prompting required substantial iteration. The initial versions suffered from poor design choices, including insufficient colour contrast between elements, unreadable input fields, and lack of visual hierarchy. After five rounds of prompting, these visual issues were resolved, but the core bug affecting completed tasks remained unresolved.

Aggregate Score: 9/15

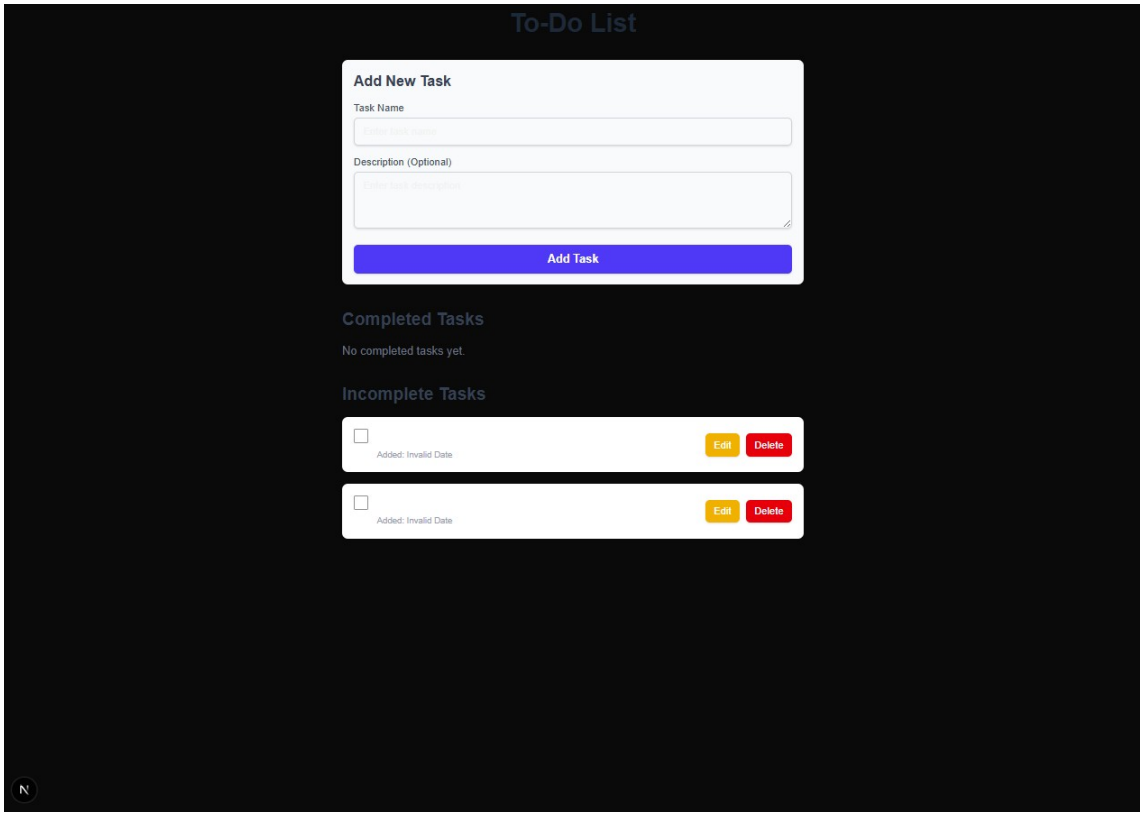

FIGURE 7.18: Initial Output (Gemini 2.5)

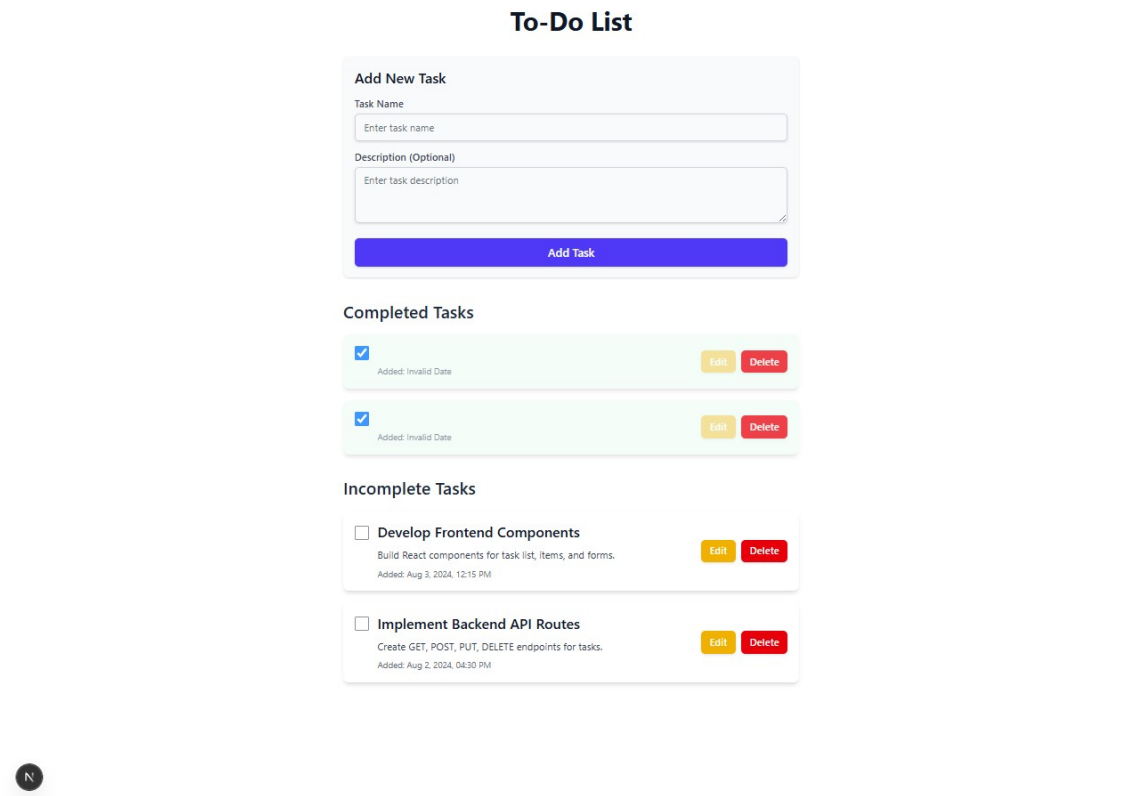

FIGURE 7.19: Final Output (Gemini 2.5)

7.5.4 Model Comparison

All three models successfully generated a full-stack application that adhered to the task specification, though the quality of implementation varied significantly. Claude 3.7 Sonnet produced the most complete and polished solution, requiring no additional prompting and achieving top scores in all evaluation categories. OpenAI o4 performed reliably and with minimal prompting, though its output exhibited weaker UI design and a minor reliability issue. Gemini 2.5 met functional requirements but introduced a significant usability flaw and required multiple rounds of prompting to reach an acceptable level of visual clarity.

7.6 Conclusion

Across the five long-form programming tasks, all three language models demonstrated the capacity to generate functionally complete applications with varying degrees of code quality, usability, and prompting efficiency. Simpler tasks such as the CLI application and the Flappy Bird game were handled with relative ease by all models, often requiring minimal prompting. More complex challenges like the full-stack project and the Spring Boot backend demanded more structural understanding, integration across layers, and stronger design conventions, which exposed clearer differences between the models.

While all systems showed the ability to solve problems to a reasonable degree, Claude 3.7 Sonnet consistently delivered robust and well-structured implementations across different domains. It often required fewer corrections and produced outputs that adhered closely to best practices. OpenAI o4 performed reliably and was prompt effiecent, though its solutions occasionally lacked polish or architectural layering. Gemini 2.5 showed competence but required more prompting and exhibited stability issues in several outputs. Average scores and a more detailed comparison will follow in the next chapter, where broader implications and relative performance trends are discussed.

Chapter 8

Large Problems - Discussions

This chapter provides a focused analysis of how each language model performed across the set of large-scale programming tasks. For each system, I present a summary of its average scores in functional completeness, code quality, and ease of prompting, alongside a qualitative evaluation of its strengths and limitations. The aim is to assess not just whether the models produced working solutions, but how effectively they handled design structure, responded to user input, and adapted across varied development contexts. Each section concludes with a recommendation based on the model's suitability for agentic workflows within Cursor.

8.1 Claude 3.7 Sonnet

Quick Stats

- Average Functional Completeness: 4.8
- Average Code Quality: 4.8
- Average Ease of Prompting: 4.6
- Average Aggregate Score: 14.4
- Average Solution Length: highest

Claude 3.7 Sonnet was the best-performing model across the five evaluated tasks. It maintained almost perfect averages in functional completeness and code quality, while also ranking highly in ease of prompting. Compared to the other models, it produced longer outputs, yet this additional length did not undermine clarity or structure. Its implementations tended to reflect a more deliberate and modular approach to code generation, often anticipating architectural decisions that aligned with current best practices in software engineering.

Its higher solution length was largely due to the inclusion of useful structural features such as additional files to isolate logic or explanatory comments to guide interpretation. This length did not lead to bloated or repetitive code, and SonarQube analysis confirmed that the outputs maintained strong ratings for readability and maintainability. In frontend tasks, it demonstrated clear visual hierarchies and employed layout mechanisms that worked consistently across screen sizes. These qualities were supported by a reliable handling of interactive components, a feature that sometimes posed challenges for the other models.

Prompting efficiency was another strength. Most solutions required only one round of interaction to reach a working state. In cases where follow-up was necessary, the corrections were integrated smoothly. For example, it responded effectively when instructed to replace unsupported Tailwind CSS classes or to resolve environment-specific issues related to backend integration. The ability to incorporate such feedback with minimal prompting suggests a readiness for agentic workflows where human input is sparse or high-level.

Given its strengths in structure, correctness, and responsiveness, Claude 3.7 Sonnet would be my first choice when working in agentic mode within Cursor. It balances autonomy and reliability well, making it particularly effective for tasks that benefit from thoughtful code organisation and minimal prompt iteration.

8.2 OpenAI o4

Quick Stats

- Average Functional Completeness: 4.4
- Average Code Quality: 4.4
- Average Ease of Prompting: 4.6
- Average Aggregate Score: 13.4
- Average Solution Length: shortest

OpenAI o4 delivered consistently functional results across the evaluated tasks. It tended to generate straightforward implementations that adhered to the specification without requiring much iterative refinement. Compared to Claude 3.7 Sonnet, its outputs were generally less structured and omitted more advanced architectural features, but they were still accurate and usable in most cases.

One of its clearest strengths was the conciseness of its responses. The model frequently returned solutions that were significantly shorter than those of the other systems, often concentrating on just the core functionality. This brevity made the outputs easy to evaluate. In contexts where minimalism is preferred, such as rapid prototyping or isolated scripting, this tendency to keep things focused worked well.

The limited length also introduced some downsides. Abstractions like service layers or reusable modules were often missing, and interface components could lack design clarity or refinement. These gaps were more noticeable in larger or multi-layered applications, where the absence of structural depth could affect maintainability and reduce the flexibility of the codebase.

Prompting was efficient and usually required only one interaction to reach a working result. The model responded accurately to feedback and integrated corrections with little friction. Although it did not frequently anticipate design needs or offer enhancements without being asked, it remained reliable when guided with precise instructions.

OpenAI o4 is a solid choice for generating compact, functional code. While it does not consistently apply higher level design principles, it performs well when the task is clearly scoped and direct implementation is preferred. For agentic workflows in tools like Cursor, it would be a dependable secondary option, particularly for tasks that emphasise speed and clarity over architectural sophistication.

8.3 Gemini 2.5

Quick Stats

- Average Functional Completeness: 3.8
- Average Code Quality: 4.4
- Average Ease of Prompting: 3.6
- Average Aggregate Score: 11.8
- Average Solution Length: in the middle

Gemini 2.5 delivered mixed results across the five evaluated tasks. While it was capable of producing functionally correct code, its outputs often lacked consistency. Problems included missing features, formatting issues, and unstable behaviours that emerged even when the prompt was clearly specified. This variability made it difficult to rely on the model for tasks that required a high degree of correctness or stability.

Code quality, as reported by SonarQube, was mostly acceptable. The model received strong ratings for maintainability and security, and few critical issues were flagged during static analysis.

Prompting efficiency was lower than with the other models. While some outputs improved with clarification, others required several iterations before producing a working version. In one task, the model did not respond effectively to feedback, and the session had to be restarted. This level of unpredictability limits its usefulness in settings where low friction interaction is expected.

Agentic performance in Cursor was also limited. The model did not reliably infer user intent or apply adjustments without explicit prompting, and it struggled to maintain consistent direction across iterations. This made its behaviour difficult to anticipate and reduced its effectiveness in workflows that benefit from model initiative.

In its current state, Gemini 2.5 does not offer the reliability or responsiveness needed for agentic development. Given its instability, higher prompting burden, and unpredictable behaviour in Cursor, I would not recommend using this model.

8.4 Model Architectures

Claude 3.7 Sonnet stood out in large-scale programming tasks, both because of how it's built and how well it fits into Cursor's workflows. Unlike models from other companies, Anthropic explicitly publishes metrics focused on agent-like capabilities [7]. In my tests, it often picked up on what I wanted without needing much prompting. Its responses followed best practices even when I did not specify them, which suggests it's been trained to prioritise structure and clarity.

OpenAI's models were somewhat less consistent. For smaller problems, o1-mini clearly performed best. It gave accurate and often fairly complex answers, even when the task was vague. I suspect that's because it's been trained on datasets like Codeforces or SWE-bench. The o4-mini performed well on short tasks but did not scale up as effectively [42]. It rarely built a full solution with the same modularity or maintainability on the same level as Claude. It appears that the model places more emphasis on correctness and less on overall coherence, which could explain its performance on the small problems.

Gemini 2.5 Pro had a very different profile. Its main technical advantage is the massive context window, with a limit of up to two million tokens compared to the roughly 200,000

in Claude 3.7 Sonnet and o4-mini [18]. In theory, that should help with multi-file reasoning or context-heavy tasks, but in practice, I did not see much benefit. It often lost track of what I was asking and needed follow-up prompts to stay on course. It did not seem to use the full context window effectively, and I did not get the sense that it was tuned for agent-like behaviour in coding environments. Its strengths in general language processing did not carry over well here.

Overall, Claude 3.7 Sonnet was a clear standout for agentic behaviour, specifically in Claude. While OpenAI's models were strong on correctness in isolated tasks, they lacked the broader perspective needed for larger systems. Claude's advantage seems to lie in how it reasoned through structure and responded with coherence across multiple steps.

8.5 Conclusion

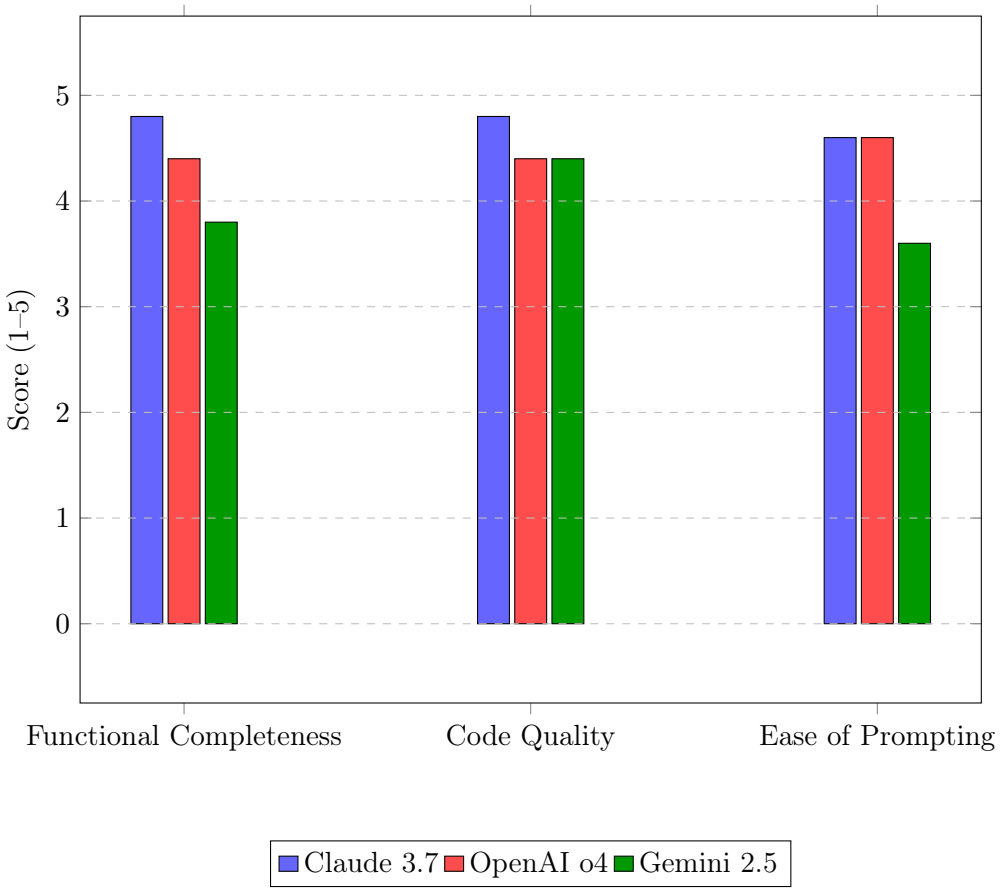

FIGURE 8.1: Model Evaluation Across Metrics

While all three models demonstrated the capacity to generate viable code solutions, their consistency, structure, and interaction requirements varied noticeably.

Claude 3.7 Sonnet delivered the most reliable results, maintaining high scores across all evaluation categories. Its outputs were longer but consistently well-organised, and its agentic behaviour aligned well with real-world development workflows. OpenAI o4 also performed strongly, especially in tasks where compact, focused code was sufficient. Although it lacked some structural depth, it remained efficient and dependable when guided by precise prompts. Gemini 2.5 Pro, by contrast, showed occasional strength in isolated scenarios

but suffered from instability, prompting inefficiency and unreliable follow-through.

These results suggest that while current LLMs are capable of addressing large-scale software development problems, the quality of their outputs varies considerably depending on the model's design preferences and ability to generalise across domains.

Therefore, to explicitly answer **RQ2**, my results show that Anthropics Claude 3.7 Sonnet comes out on top as the best model for large-scale programming problems and agentic capabilities.

Chapter 9

Future Work

This chapter will go over the limitations of my research and how the domain can be explored further. While the experiments establish a reproducible benchmark for assessing model performance on structured problems, the scope of this work remains focused. Several limitations were acknowledged during the design of my methodology, and future work can address these by extending the evaluation in multiple directions.

9.1 Threats to Validity

There are several factors that can be considered threats to the validity of the conclusions made in this paper. Here are the primary threats I have identified:

- **Non-determinism of the models:** The same model prompted with the exact same input on two occasions can yield noticeably different outputs. This is a byproduct of the stochastic behaviour embedded in LLM architectures. While diversity of responses can lead to improved outcomes over repeated trials, it limits the strict reproducibility of results..
- **Preview state of certain models:** Some models evaluated in this study, in particular Gemini 2.5 Flash and Gemini 2.5 Pro, were still in a preview state when the experiments were carried out. These versions may not reflect the final production performance, and the results may shift significantly in future iterations.
- **Prompt dependence and sensitivity:** The outcomes observed were highly dependent on the initial phrasing and structure of the prompts. A different formulation, even with the same intent, could potentially lead to different behaviour. This makes it difficult to isolate model competence from prompt engineering effectiveness, especially when evaluating ease-of-use in agentic experiments.
- **Tooling and interface influence:** All large-problem experiments were conducted using Cursor IDE. While Cursor supports consistent and structured prompting, its features, such as context windowing, auto-suggestion, and agentic mode, could interact differently with each model. Results may not generalise to other development environments.
- **Temporal validity of model performance:** LLMs are rapidly evolving, and several newer models such as GPT 4.5 and Claude 4 were released shortly after the experiments concluded. The benchmark reflects the state of LLMs as of early 2025,

and results may not hold as newer models are released by OpenAI, Google and Anthropic.

9.2 Possible Advancements

9.2.1 Increase Problem Set Diversity

The small programming tasks in my work were drawn exclusively from LeetCode. This platform offers a clear problem format and a wide array of topics, making it suitable for standardised testing. However, this reliance on a single source limits the variety of reasoning patterns and computational strategies that were observed in the results.

Other platforms pose different challenges. HackerRank includes questions with emphasis on practical application areas, and Rosetta Code features problems that require more rigorous algorithmic optimisation [32, 50]. Introducing such alternatives would help test whether model performance holds up under unfamiliar structures or constraints that differ from those found in LeetCode.

Additionally, all problems in my evaluation were solved in Python 3. This choice was made to reduce variability and to work within the most commonly used language on LeetCode. It remains unclear whether the results would generalise to other languages. Languages such as Java, C++, or JavaScript differ in syntax verbosity and idiomatic conventions. Testing the same problems in multiple languages could reveal whether some models exhibit language-specific biases or struggle with particular paradigms.

9.2.2 More Complex Architectures

In the large-problem evaluation, my focus was on relatively self-contained applications. The largest of these was still implementable within less than 1,000 lines of code. These projects allowed me to measure LLM performance across multiple layers of software design, but they did not approach the complexity typically found in production-level systems.

Most real-world applications are composed of multiple interconnected components that operate independently and communicate through shared protocols or interfaces. My work did not assess whether models can generate or coordinate across such distributed systems. A logical next step would be to evaluate model performance on software stacks that include state management across modules or persistent storage strategies across services.

Future work could also test whether models can handle tasks such as deployment setup, environment configuration, and automated scaffolding for application frameworks. These capabilities are increasingly relevant as LLMs are integrated into development workflows, and the current study offers no insight into how well such tasks are handled.

9.2.3 Human Comparison

Throughout this study, my analysis focused on the outputs of LLMs without reference to human-written code. While this allowed for a clear and focused evaluation of model outputs, it limits the interpretability of the findings. Without a human baseline, it is difficult to say whether the generated solutions resemble good practice, beginner-level code, or something entirely outside of developer norms.

Future research could collect solutions from experienced programmers and compare them to the ones produced by models. This would help identify areas where the models fall short in clarity, logic, or maintainability. It would also allow for a better understanding of how readable or idiomatic the generated code is. These comparisons could be quantitative,

using metrics like maintainability or complexity, but also qualitative, involving developer assessments of style, structure, and design. Such an evaluation would be especially useful in assessing how close LLMs are to replacing or augmenting real-world programming tasks.

9.2.4 Model Coverage

All experiments in my work used models that were publicly available as of early 2025. While these models represented a wide range of performance and design strategies, the pace of model development means that the findings may already be outdated.

Since completing my experiments, newer models such as GPT-4.5 and Claude 4 have been released. These models advertise significant improvements, and their inclusion in future benchmarks would enable a more current assessment of the field's state. Re-running my benchmark with these newer models could help determine whether previously observed issues have been improved in any measurable way.

My study also did not include testing with free models. Developers and researchers increasingly use alternatives such as Code Llama or StarCoder due to their accessibility [53, 39]. Including these models would expand the benchmark and enable more inclusive comparisons across a range of model types, not just the major providers.

9.2.5 Test Generation

In all the programming tasks I evaluated, the focus was solely on generating the implementation. I did not assess the models' ability to produce tests. Testing is a key component of the software development process, especially in the context of larger problems.

Testing whether a model can produce meaningful unit or integration tests would introduce a more complete view of its programming abilities. Good testing practices require not just familiarity with syntax but also an understanding of edge cases, potential failure points, and expected user behaviour. My current experiments do not provide any information about how models handle this aspect of the development process.

In future benchmarks, prompts could be adapted to request both code and associated test cases. Evaluation could then include coverage metrics, pass/fail rates, or mutation testing to assess test quality. This would move the benchmark closer to capturing real-world software engineering needs and would add another critical dimension to model evaluation.

9.2.6 Debugging and Error Correction

In my study, error correction was evaluated in a limited and somewhat indirect way. When a generated solution failed, such as in the large programming tasks, I prompted the same model again with the error message or output in order to obtain a fix. This offered a basic way to observe whether the model could resolve faults in its own code, but it did not assess its ability to debug unfamiliar or externally written code.

Future work could design a more systematic evaluation focused specifically on debugging. One approach would be to create a set of intentionally flawed code samples in multiple languages and frameworks. These errors could range from simple syntax issues to more subtle semantic or logical faults. By controlling for the type, frequency, and severity of bugs, it would be possible to benchmark how well different models perform under increasingly difficult debugging scenarios.

Such a dataset would allow researchers to test not just whether a model can fix a bug, but what kind of context it requires, how efficiently it isolates the root cause, and whether it introduces regressions in the process. It could also expose how performance varies across

ecosystems, for instance by comparing results in Python, JavaScript, Java, or C#. Since debugging is a critical part of software development, assessing model proficiency in this area would strengthen the practical relevance of LLM evaluations.

9.2.7 Impact on Education

Another area worth exploring is the effect of LLMs on programming education. While this research evaluated how well models perform tasks, it did not examine how their usage influences learning, especially for those new to programming. Future work could study whether these tools support concept retention or foster over-reliance, particularly when students are exposed to model-generated code before fully grasping underlying principles.

Experiments could be designed where novice programmers solve problems with and without access to LLMs. Their performance, error correction strategies, and ability to learn from model feedback could be compared over time. This would help evaluate whether LLMs can be a useful tool or if they risk automating too much of the learning process too early.

In my own experiments, I was able to correct model errors because of my prior experience with the languages and tools involved. A less experienced user might accept incorrect suggestions without question, potentially reinforcing misconceptions. Research that measures how learners of different skill levels interact with models would be valuable in learning how these tools should be integrated into existing curricula.

9.3 Conclusion

While the current work provides a structured evaluation of LLM capabilities on a range of programming tasks, it remains limited in scope. Future work could expand the dataset to include more diverse platforms and programming languages, evaluate more complex software architectures, and compare the outputs to those of human-written code. Additional areas include expanding model coverage as new systems become available, testing the ability to generate meaningful software tests, and designing tasks that specifically assess debugging performance. These extensions would not only address gaps in the current methodology but also align the benchmark more closely with real-world development needs.

Chapter 10

Conclusion

This thesis set out to evaluate the programming capabilities of major LLMs. The LLMs that were tested were the flagship offerings from OpenAI, Google, and Claude. These models were chosen as they represent the most popular and widely used models today. While the design choices of each individual model may differ, in the background of all these models lies the Transformer architecture. This is essentially a neural network that works with self-attention mechanisms that allow these models to handle long-range dependencies between texts appropriately. Along with strong natural language processing, it allows these models to take in input via natural language and also reply in a human-like manner. The reason for separating the research into two separate groups, one for small problems and one for large problems, was due to the fact that these have been noted to be fundamentally different tasks, and we are testing different capabilities of the system.

The methodology to assess the small problems involved an extensive pipeline. The problem set for these exercises was 75 LeetCode problems, with 25 of each difficulty, easy, medium and hard. These problems encompassed a wide variety of topics such as bit manipulation, arrays, depth-first search, strings, dynamic programming, etc. Five models were tested here: the old and new generations of OpenAI models with GPT-4o and o1-mini, the old and new generations from Google with Gemini 2.0 Flash and Gemini 2.5 Flash, as well as the flagship offering from Anthropic in Claude 3.7 Sonnet. A carefully selected prompt was crafted to get the best possible outputs. The quality of the solutions was evaluated not just on accuracy but also on code quality metrics such as Maintainability Index, Source Lines of Code and Cyclomatic Complexity. All the data from the experiments were stored in a database for easy data retrieval and visualisation. Selenium automations were written to automatically scrape a sample page to retrieve all relevant problem information and to paste a generated solution into the site to verify whether it accurately solved the problem.

The results showed that OpenAI's o1 model had the best accuracy among all the models when it comes to LeetCode problems, with Claude 3.7 Sonnet, Gemini 2.0, and Gemini 2.5 all fairly similar and can be considered in the middle tier. GPT-4o was the worst model and performed significantly worse than the other models. Claude 3.7 Sonnet was shown to generate the code with the best code quality, having the highest Maintainability Index score and reasonable average lines of code and Cyclomatic Complexity per solution. Statistical tests were performed to come to the conclusions mentioned, specifically McNemar's test and Wilcoxon's Signed-Rank Test.

The methodology used to assess the significant problems varied significantly. Five unique programming tasks were identified that encompassed a wide array of languages like Java, Python and JavaScript and different types of projects such as games, websites and CLI applications. The models that were tested were OpenAI o4, Google Gemini 2.5 and

Claude 3.7. Each solution was graded on three metrics, which were graded on a scale from 1 to 5. These metrics evaluated were functional completeness, maintainability and ease of prompting. The models were interfaced with through Cursor IDE and were set to agentic mode to minimise the amount of human intervention.

The results of the above experiments showed that Claude 3.7 Sonnet emerged as the best agentic model for the set of problems tested. It managed to have the highest (or tied highest) score for each of the three metrics that I had laid out. OpenAI o4 came in at second place, with Gemini 2.5 seeming to lag behind in my experiments.

Although the research conducted is extensive, several areas remain for future work to explore. These include testing with more models from different providers, considering a wider array of problems to test, such as programming problems from other sites like HackerRank, and coming up with non-monolithic, large problems that test dependencies more explicitly. Comparisons can also be done to see how exactly LLM-generated code differs from that of human-written code. Lastly, other aspects related to software development, other than just code generation, can be tested, such as the ability of models to write tests and debug and correct errors in codebases.

To conclude, LLM models in their current state are powerful and can certainly prove to be a valuable tool for generating code for both small and large problems. While there are several systematic errors that may arise and their outputs are not perfect, complementing their use with human intervention and knowledge can lead to useful outputs while saving effort and time on the programmer's side.

Bibliography

[1] 13 Code Quality Metrics That You Must Track | Opsera. URL: https://www.opsera.io/blog/13-code-quality-metrics-that-you-must-track.

[2] 21 Inspiring Software Engineering Projects Explored. URL: https://www.springboard.com/blog/software-engineering/software-engineering-projects/.

[3] 23 Amazing Google Gemini Statistics (Users, Facts) - Content Detector AI. URL: https://contentdetector.ai/articles/google-gemini-statistics/.

[4] 7 examples of Gemini's multimodal capabilities in action - Google Developers Blog. URL: https://developers.googleblog.com/en/7-examples-of-geminis-multimodal-capabilities-in-action/.

[5] Bard updates from Google I/O 2023: Images, new features. URL: https://blog.google/technology/ai/google-bard-updates-io-2023/.

[6] Claude 2 \ Anthropic. URL: https://www.anthropic.com/news/claude-2.

[7] Claude 3.7 Sonnet and Claude Code: The best coding LLM is here | by Mehul Gupta | Data Science in Your Pocket | Medium. URL: https://medium.com/data-science-in-your-pocket/claude-3-7-sonnet-and-claude-code-the-best-coding-llm-is-here-7a61d79b96d1.

[8] Claude 3.7 Sonnet \ Anthropic. URL: https://www.anthropic.com/claude/sonnet.

[9] Claude's Constitution \ Anthropic. URL: https://www.anthropic.com/news/claudes-constitution.

[10] Cline - AI Autonomous Coding Agent for VS Code. URL: https://cline.bot/.

[11] Codeforces. URL: https://codeforces.com/.

[12] Codium is now Qodo | Quality-first AI Coding Platform. URL: https://www.qodo.ai/.

[13] Cursor - The AI Code Editor. URL: https://www.cursor.com/.

[14] Filess.io | 100% Free Database Hosting forever. URL: https://filess.io/.

[15] Flappy Bird - Wikipedia. URL: https://en.wikipedia.org/wiki/Flappy_Bird.

[16] Gemini 2.0: Everything you need to know. URL: https://www.androidpolice.com/gemini-2-new-good-and-bad/.

[17] Gemini 2.0: Flash, Flash-Lite and Pro - Google Developers Blog. URL: https://developers.googleblog.com/en/gemini-2-family-expands/.

[18] Gemini 2.5: Our newest Gemini model with thinking. URL: https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/#gemini-2-5-thinking.

[19] Gemini: A Family of Highly Capable Multimodal Models - AI Resources. URL: https://www.modular.com/ai-resources/g.

[20] Gemini Ultra vs Gemini Pro: The Ultimate AI Battle. URL: https://myscale.com/blog/gemini-ultra-vs-gemini-pro-ultimate-showdown/.

[21] GitHub Copilot · Your AI pair programmer. URL: https://github.com/features/copilot.

[22] Google AI: What to know about the PaLM 2 large language model. URL: https://blog.google/technology/ai/google-palm-2-ai-large-language-model/.

[23] Google Gemini: Fact or Fiction? URL: https://cameronrwolfe.substack.com/p/google-gemini-fact-or-fiction.

[24] Google Gemini vs Bard: The Main Differences - UC Today. URL: https://www.uctoday.com/collaboration/google-gemini-vs-bard-the-main-differences/.

[25] Google introduces Gemini 2.0: A new AI model for the agentic era. URL: https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/#ceo-message.

[26] Google Releases Bard, Its AI Chatbot, a Rival to ChatGPT and Bing - The New York Times. URL: https://www.nytimes.com/2023/03/21/technology/google-bard-chatbot.html.

[27] Google's Latest Release: Gemini 2.0 Large-Scale Language Model — A Major Breakthrough in AI Technology | by Ares | Medium. URL: https://medium.com/@ares870802/googles-latest-release-gemini-2-0-9f7a4fa2d7f7.

[28] Google's New Year's Resolution: Help Gemini Catch Up to ChatGPT - WSJ. URL: https://www.wsj.com/tech/ai/google-gemini-2025-chatgpt-openai-b6eb595d.

[29] GPT-3 vs. GPT-4: What's the Difference? | Grammarly. URL: https://www.grammarly.com/blog/ai/gpt-3-vs-gpt-4/.

[30] GPT-4 | OpenAI. URL: https://openai.com/index/gpt-4/.

[31] GPT-4o mini: advancing cost-efficient intelligence | OpenAI. URL: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/.

[32] HackerRank - Online Coding Tests and Technical Interviews. URL: https://www.hackerrank.com/.

[33] Hello GPT-4o | OpenAI. URL: https://openai.com/index/hello-gpt-4o/.

[34] History Of ChatGPT: A Timeline Of Generative AI Chatbots. URL: https://www.searchenginejournal.com/history-of-chatgpt-timeline/488370/.

[35] How to measure code quality: 10 metrics you must track. URL: https://www.future-processing.com/blog/code-quality-metrics-that-you-should-measure/.

[36] How Transformers Work: A Detailed Exploration of Transformer Architecture | DataCamp. URL: https://www.datacamp.com/tutorial/how-transformers-work.

[37] Introducing Claude 3.5 Sonnet \ Anthropic. URL: https://www.anthropic.com/news/claude-3-5-sonnet.

[38] Introducing Claude \ Anthropic. URL: https://www.anthropic.com/news/introducing-claude.

[39] Introducing Code Llama, a state-of-the-art large language model for coding. URL: https://ai.meta.com/blog/code-llama-large-language-model-coding/.

[40] Introducing Gemini 1.5, Google's next-generation AI model. URL: https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/#sundar-note.

[41] Introducing Gemini: Google's most capable AI model yet. URL: https://blog.google/technology/ai/google-gemini-ai/.

[42] Introducing OpenAI o3 and o4-mini | OpenAI. URL: https://openai.com/index/introducing-o3-and-o4-mini/.

[43] LaMDA: our breakthrough conversation technology. URL: https://blog.google/technology/ai/lamda/.

[44] OpenAI o1-mini | OpenAI. URL: https://openai.com/index/openai-o1-mini-advancing-cost-efficient-reasoning/.

[45] OpenAI o3-mini | OpenAI. URL: https://openai.com/index/openai-o3-mini/.

[46] OpenRouter. URL: https://openrouter.ai/.

[47] OWASP Top Ten | OWASP Foundation. URL: https://owasp.org/www-project-top-ten/.

[48] Pomodoro Technique - Wikipedia. URL: https://en.wikipedia.org/wiki/Pomodoro_Technique.

[49] radon · PyPI. URL: https://pypi.org/project/radon/.

[50] Rosetta Code. URL: https://rosettacode.org/wiki/Rosetta_Code.

[51] Selenium. URL: https://www.selenium.dev/.

[52] SonarQube | Sonar | Sonar. URL: https://www.sonarsource.com/sem/products/sonarqube/.

[53] StarCoder: A State-of-the-Art LLM for Code. URL: https://huggingface.co/blog/starcoder.

[54] Start building with Gemini 2.5 Flash - Google Developers Blog. URL: https://developers.googleblog.com/en/start-building-with-gemini-25-flash/.

[55] SWE-bench Leaderboard. URL: https://www.swebench.com/.

[56] Technology | 2024 Stack Overflow Developer Survey. URL: https://survey.stackoverflow.co/2024/technology#most-popular-technologies-language.

[57] The Transformer Model - MachineLearningMastery.com. URL: https://machinelearningmastery.com/the-transformer-model/.

[58] Top 15 Software Engineering Projects 2025 - GeeksforGeeks. URL: https://www.geeksforgeeks.org/software-engineering-projects/.

[59] Top Generative AI Chatbots by Market Share – May 2025 – First Page Sage. URL: https://firstpagesage.com/reports/top-generative-ai-chatbots/.

[60] View of Clarifying the Dialogue-Level Performance of GPT-3.5 and GPT-4 in Task-Oriented and Non-Task-Oriented Dialogue Systems. URL: https://ojs.aaai.org/index.php/AAAI-SS/article/view/27668/27441.

[61] What Does 'ChatGPT' Stand For? URL: https://www.lifewire.com/what-does-chatgpt-stand-for-8673919.

[62] What is Vibe Coding? | IBM. URL: https://www.ibm.com/think/topics/vibe-coding.

[63] Why Does ChatGPT Use Only Decoder Architecture? URL: https://www.analyticsvidhya.com/blog/2024/06/why-does-chatgpt-use-only-decoder-architecture/.

[64] Syeda Nahida Akter, Zichun Yu, Aashiq Muhamed, Tianyue Ou, Alex Bäuerle, Ángel Alexander Cabrera, Krish Dholakia, Chenyan Xiong, and Graham Neubig. An In-depth Look at Gemini's Language Abilities. 12 2023. URL: https://arxiv.org/pdf/2312.11444.

[65] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program Synthesis with Large Language Models. 8 2021. URL: https://arxiv.org/pdf/2108.07732.

[66] Md Mustakim Billah, Palash Ranjan Roy, Zadia Codabux, and Banani Roy. Are Large Language Models a Threat to Programming Platforms? An Exploratory Study. *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 10:292–301, 9 2024. URL: https://dl.acm.org/doi/10.1145/3674805.3686689, doi:10.1145/3674805.3686689.

[67] Alessio Buscemi. A Comparative Study of Code Generation using ChatGPT 3.5 across 10 Programming Languages. 8 2023. URL: https://arxiv.org/pdf/2308.04477.

[68] Lingjiao Chen, Matei Zaharia, and James Zou. How is ChatGPT's behavior changing over time? *Harvard Data Science Review*, 6(2), 7 2023. doi:10.1162/99608f92.5317da47.

[69] Carlos Eduardo Andino Coello, Mohammed Nazeh Alimam, and Rand Kouatly. Effectiveness of ChatGPT in Coding: A Comparative Analysis of Popular Large Language Models. *Digital 2024, Vol. 4, Pages 114-125*, 4(1):114–125, 1 2024. URL: https://www.mdpi.com/2673-6470/4/1/5, doi:10.3390/DIGITAL4010005.

[70] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Introduction to Algorithms, Third Edition.

[71] Domenico Cotroneo, Roberta De Luca, and Pietro Liguori. DeVAIC: A Tool for Security Assessment of AI-generated Code. *Information and Software Technology*, 177, 4 2024. URL: https://www.sciencedirect.com/science/article/pii/S0950584924001770.

[72] Frank Deremer and Hans Kron. Programming-in-the large versus programming-in-the-small. *Proceedings of the 1975 International Conference on Reliable Software*, pages 114–121, 4 1975. URL: https://doi.org/10.1145/800027.808431.

[73] Muhammad Fawad, Akbar Khan, Max Ramsdell, Erik Falor, and Hamid Karimi. Assessing the Promise and Pitfalls of ChatGPT for Automated Code Generation. *Educational Data Mining*, 1, 11 2023. doi:10.5281/zenodo.12729778.

[74] Anas Jebreen Atyeh Husain. Potentials of ChatGPT in Computer Programming: Insights from Programming Instructors. *Journal of Information Technology Education: Research*, 23(2):002–undefined, 1 2024. doi:10.28945/5240.

[75] Baskhad Idrisov and Tim Schlippe. Program Code Generation with Generative AIs. *Algorithms 2024, Vol. 17, Page 62*, 17(2):62, 1 2024. URL: https://www.mdpi.com/1999-4893/17/2/62, doi:10.3390/A17020062.

[76] Bailey Kimmel, Austin Geisert, Lily Yaro, Brendan Gipson, Taylor Hotchkiss, Sidney Osae-Asante, Hunter Vaught, Grant Wininger, and Chase Yamaguchi. Enhancing Programming Error Messages in Real Time with Gener-ative AI. doi:10.1145/3613905.3647967.

[77] Umberto Leòn-Domínguez. Potential Cognitive Risks of Generative Transformer-Based AI Chatbots on Higher Order Executive Functions. *Neuropsychology*, 38(4):293–308, 2 2024. doi:10.1037/NEU0000948.

[78] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT. *IEEE Transactions on Software Engineering*, 50(6):1548–1584, 8 2023. doi:10.1109/TSE.2024.3392499.

[79] Manuel Merkel and Jens Dörpinghaus. A case study on the transformative potential of AI in software engineering on LeetCode and ChatGPT. 1 2025. URL: https://arxiv.org/pdf/2501.03639.

[80] Alec Radford Openai, Karthik Narasimhan Openai, Tim Salimans Openai, and Ilya Sutskever Openai. Improving Language Understanding by Generative Pre-Training. URL: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

[81] Mirza Masfiqur Rahman, Ashish Kundu, Ramana Kompella, and Elisa Bertino. Code Hallucination. 7 2024. URL: https://arxiv.org/pdf/2407.04831.

[82] Laria Reynolds and Kyle McDonell. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. *Conference on Human Factors in Computing Systems - Proceedings*, 2 2021. doi:10.1145/3411763.3451760.

[83] Fardin Ahsan Sakib, Saadat Hasan Khan, and A. H. M. Rezaul Karim. Extending the Frontier of ChatGPT: Code Generation and Debugging. *International Conference on Electrical, Computer and Energy Technologies*, 7 2023. doi:10.1109/ICECET61485.2024.10698405.

[84] Sivasurya Santhanam, Tobias Hecking, Andreas Schreiber, and Stefan Wagner. Bots in software engineering: a systematic mapping study. *PeerJ Computer Science*, 8:e866, 2 2022. URL: https://peerj.com/articles/cs-866, doi:10.7717/PEERJ-CS.866/SUPP-3.

[85] Euibeom Shin, Yifan Yu, Robert R. Bies, and Murali Ramanathan. Evaluation of ChatGPT and Gemini large language models for pharmacometrics with NONMEM. *Journal of Pharmacokinetics and Pharmacodynamics*, 51(3):187–197, 6 2024. URL: https://pubmed.ncbi.nlm.nih.gov/38656706/, doi:10.1007/s10928-024-09921-y.

[86] Norbert Tihanyi, Tamas Bisztray, Mohamed Amine Ferrag, Ridhi Jain, and Lucas C. Cordeiro. How secure is AI-generated code: a large-scale comparison of large language models. *Empirical Software Engineering*, 30(2), 12 2024. URL: https://dl.acm.org/doi/10.1007/s10664-024-10590-1, doi:10.1007/S10664-024-10590-1.

[87] Catherine Tony, Mohana Balasubramanian, Nicolás E.Díaz Ferreyra, and Riccardo Scandariato. Conversational DevBots for Secure Programming: An Empirical Study on SKF Chatbot. *ACM International Conference Proceeding Series*, pages 276–281, 6 2022. URL: https://dl.acm.org/doi/10.1145/3530019.3535307, doi:10.1145/3530019.3535307.

[88] Waqas Uzair and Sameen Naz. Six-Tier Architecture for AI-Generated Software Development: A Large Language Models Approach. 6 2023. URL: https://www.researchsquare.com/article/rs-3086026/v1, doi:10.21203/RS.3.RS-3086026/V1.

[89] Asokan Vasudevan, Alma Vorfi Lama, and Zohaib Hassan Sain. The Game-Changing Impact of AI Chatbots on Education ChatGPT and Beyond. *Journal of Information Systems and Technology Research*, 3(1):38–44, 1 2024. URL: https://journal.aira.or.id/index.php/jistr/article/view/770, doi:10.55537/JISTR.V3I1.770.

[90] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *Advances in Neural Information Processing Systems*, 2017-December:5999–6009, 6 2017. URL: https://papers.nips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[91] Lun Wang, Chuanqi Shi, Shaoshui Du, Yiyi Tao, Yixian Shen, Hang Zheng, Yanxin Shen, and Xinyu Qiu. Performance Review on LLM for solving leetcode problems. 2 2025. URL: https://arxiv.org/pdf/2502.15770v1.

[92] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems*, 35, 1 2022. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf.

[93] Marvin Wyrich, Daniel Graziotin, and Stefan Wagner. A theory on individual characteristics of successful coding challenge solvers. *PeerJ Computer Science*, 2019(2):e173, 2 2019. URL: https://peerj.com/articles/cs-173, doi:10.7717/PEERJ-CS.173/SUPP-2.

[94] Wang Xian, Chen Guomin, Varsha Arya, and Kwok Tai Chui. Examining the Influence of AI Chatbots on Semantic Web-Based Global Information Management in Various Industries. *https://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/IJSWIS.344037*, 20(1):1–14, 1 1. URL: www.igi-global.com/gateway/article/344037, doi:10.4018/IJSWIS.344037.

Declarations

A range of digital tools and platforms were used to support the writing and research processes involved in this thesis. I take full responsibility for the final content and interpretations presented herein, including the output resulting from the use of these tools.

Writing Assistance

To support the clarity and correctness of the written text, I made use of the **Grammarly** extension. This tool helped to identify and correct grammatical inconsistencies and improve sentence structure. Conversational AI systems such as **Perplexity** and **ChatGPT** were also sometimes used. The models employed in these systems were primarily OpenAI o1 and GPT-4o. These tools were used primarily to help restructure certain passages and enhance the overall readability of the document. At all times, the intellectual content and arguments remained my own, these tools were used solely to improve the style and presentation of my writing.

Research Assistance

In conducting the literature review and sourcing relevant academic material, I used AI enhanced scholarly search platforms, namely **Semantic Scholar** and **SciteAI**. These platforms assisted in efficiently locating high quality, peer-reviewed articles and in addressing specific research questions. Additionally, all references were managed and organised using **Mendeley**, facilitating citation insertion and bibliography generation.