Comparative Analysis of Iterative Prompt Refinement and Multi-dimensional Quality in LLM-Generated Code

Sahanon Phisetpakasit

A dissertation submitted for the degree of
Master of Applied Science (Software Engineering) in the
School of Computing at University of Otago,
Dunedin, New Zealand

October 2025

Acknowledgement

First, I would like to express my gratitude toward my supervisors Associate Professor Sherlock Licorish and Professor Tony Savarimuthu for giving me valuable feedback and suggestion; both in academic and non-academic. I really enjoy every meeting and discussion that we had for year.

Second, I would like to thank Associate Professor Apirak Hoonlor for encouragement and tips in pursuing master's degree. Your guidance and motivation really help me know how to find a good supervisor and how to proper study masters.

Finally, I am highly indebted to my family for moral support for helping me carry my study from the start till end.

Abstract

Code generation has become a crucial component of software development, facilitating the automatic generation of source code from human prompts. However, studies examining performance across models with multi-dimensional analysis and the impact of task nature on JavaScript: a widely used programming language are scarce. This study aims to investigate the behavior of large language models (LLMs) under iterative prompt refinement when generating JavaScript code. We evaluate three models (GPT-4.1, Gemini 2.5 Pro, and Claude 4 Opus) across 30 tasks, which include data structures and algorithms, object-oriented programming (OOP), and web development. The quality of the generated code is assessed along four dimensions: correctness, maintainability, security, and an overall score. Our analysis addresses three research questions: (RQ1) How does performance evolve over iterations? (RQ2) Do changes in one dimension correlate with changes in others? (RQ3) Does the type of task influence performance?

For RQ1, repeated-measures tests and visual trajectory analyses reveal no monotonic, model-agnostic improvement: Claude demonstrates a steady decline in correctness, GPT exhibits negative returns after initial steps, and Gemini shows a dip–recovery pattern. Maintainability and security remain near ceiling levels, so overall quality largely aligns with correctness. Regarding RQ2, iteration-to-iteration delta correlations are small and often non-significant (with pairs involving security frequently undefined due to zero variance), indicating limited cross-dimensional coupling and concentrating actionable signals in correctness: particularly between iterations 0→1 (and occasionally 1→2). For RQ3, Kruskal–Wallis tests on task-level means indicate a moderate category effect for GPT-4.1 ($H(2) = 8.37$, $p = .015$, $\varepsilon^2 \approx .24$; Data Structure and Algorithm > OOP), a weaker trend for Claude ($p = .077$), and no reliable effect for Gemini; maintainability and security exhibit ceiling effects across categories. Practically, the results advocate for model-aware, task-aware, iteration-bounded refinement: keep Claude's loops short, limit GPT to one or two targeted passes, and monitor Gemini step-by-step with checkpoint/rollback. Limitations include a finite task set, ceilinged structural metrics, and the scope of three models. We conclude that iterative refinement does not consistently yield benefits nor is it predictable. Furthermore, accuracy, rather than prominent structural scores, offers the most informative indicator under current evaluation conditions.

TABLE OF CONTENT

LIST OF TABLES

LIST OF FIGURES

1. Introduction

Code generation has become an essential part of software development, offering automatic procedures for generating source code from human prompts[1]. Prompt-driven code generators, such as ChatGPT, Gemini (formerly Bard), and LLaMA, assist developers not only in writing code but also in tasks such as system design and bug detection[2]. Similarly, copilot systems, such as GitHub Copilot and Amazon Whisperer, assist developers by automating repetitive tasks and offering code suggestions, ultimately improving developer productivity and satisfaction[3]. Although expectations arise for enhancing productivity when delivering software, the outcome of studies pointed out that, without proper guidelines and fixes, generated codes are prone to flaws such as security issues and insecure suggestions [4] and often exhibiting security vulnerabilities (e.g., data leakage, improper access), poor performance, and logical or legal risks[5]. Moreover, research suggests these tools do not consistently improve task completion time or success rates in solving real-world programming problems[3, 4, 6]. In addition, the reliability of the generated code is often questioned, especially when it is used by developers lacking domain expertise[7]. These issues are especially prevalent when developers lack domain expertise and often prevent integration into production repositories owing to improper exception handling and resource management[8].

To ensure the quality of the generated code, several studies have explored refinement techniques[4, 9-11]. Prompt refinements prove to be useful and strongly improve the generated code, as they show promising results in detecting, evaluating, and enhancing the generated code with respect to many quality attributes[12]. However, existing studies rarely consider how improvements in one quality attribute, such as correctness, may inadvertently degrade or improve others, such as maintainability or performance[13]. Although prior research has explored LLM behaviour under iterative refinement[14], much of it is based on outdated models or narrow quality metrics, limiting its applicability to current-generation systems and broader software engineering issues such as functional correctness, security, and maintainability. Furthermore, there is limited understanding of how different LLMs behave under refinement and whether the nature of the programming task (e.g., algorithmic problem-solving, data structures, and OOP) affects quality dimensions. Using refinement, tasks such as algorithm implementation may achieve enhanced accuracy, whereas tasks like web development may highlight maintainability or security issues, necessitating variations of refinement outcomes.

To bridge these gaps, this study proposes a multidimensional evaluation framework (correctness, maintainability, security) for analysing the impact of iterative prompt refinement on LLM-generated code. It aims to uncover whether improvements in one quality dimension may lead to improvement or degradation in others and to what extent these effects differ across models and task types. Based on current popularity in web development, this study focuses on JavaScript programming tasks to ensure a consistent evaluation using established static analysis and mutation testing tools. Specifically, this study addresses the following research questions.

- RQ1: How do different LLMs perform with iterative prompt refinement?
- RQ2: Does improving one quality dimension (e.g., correctness) in the LLM-generated JavaScript code through refinement correlate with another (e.g., maintainability)?
- RQ3: Does the nature of the task affect the quality dimensions of the generated JavaScript code?

In summary this research aims to contribute as follows:

- We introduce a multidimensional of quality attribute assessment framework for assessing LLM-generated JavaScript code under iterative refinement.
- We empirically compare multiple LLMs across JavaScript tasks and analyze the trade-offs between correctness, maintainability, and security.
- We highlight how the task type affects the refinement outcomes across models.

The remainder of this dissertation is structured as follows: Section 2 provides a review of related literature on LLM code generation, iterative prompting, and quality attributes. Section 3 elaborates on our dataset, models, metrics, experimental pipeline, and the analysis plan for Research Questions 1 through 3. Section 4 presents the results for each research question, accompanied by figures and tables. Section 5 discusses the implications and limitations of the findings. Section 6 concludes the dissertation and outlines directions for future research, with comprehensive statistics and materials available in the Appendix.

2. Background and Related Work

2.1 Code Generation and Iterative Refinement

Code generation using large language models (LLMs) has garnered considerable attention owing to its capacity to convert natural language prompts into executable code, thereby reducing manual development efforts and enhancing developer productivity[15]. Prompt engineering techniques, including zero-shot prompting, few-shot prompting, chain-of-thought (CoT), and self-consistency, have been extensively employed to direct models towards improved outputs[16-18]. Among these, zero-shot prompting is the most prevalent, followed by zero-shot prompting with CoT and few-shot prompting, although these baselines are not always the most effective; iterative and retrieval-based strategies have been shown to outperform in many cases [9, 19]. Recent investigations have proposed frameworks for iterative refinement, wherein the model output is refined over multiple cycles. For instance, CodeQuest [5] employs iterative refinement to augment code quality across various attributes, whereas CoCoGen [10] utilizes compiler feedback and project-specific context to enhance correctness. Nevertheless, excessive iteration of prompts can deteriorate output quality[19]. Furthermore, most of these studies concentrate on enhancing a single quality dimension, such as correctness, without considering the impact of these improvements on other dimensions, such as maintainability or performance. Although iterative refinement enhances accuracy, existing research seldom examines its effects on other software quality dimensions or whether these improvements are consistent across different LLMs and task types.

2.2 Quality Attribute in LLM-Generated Code

In software engineering, quality attribute refers to the characteristics or properties that define the overall quality of a software system. Boukouchi et al.[20], presented definitions of software quality attributes from multiple standards (ISO, ANSI, IEEE). The paper outlines a set of

attributes, including features and characteristics, used to describe or evaluate a software product that meets specified needs. While traditional software quality models, for example, ISO/IEC 25002 [21] include broad range of attributes that need to be assessed, this study narrows its focus to three critical dimensions that are relevant to LLM-generated code: correctness, security, and maintainability. These attributes were selected based on their foundational importance to ensure functional reliability (correctness), long-term usability (maintainability), and risk mitigation (security).

Correctness remains a fundamental metric for evaluating the generated code, typically assessed through test case pass rates or code evaluation metrics (CEMs) such as Pass@k and AvgPassRatio[2, 22, 23]. Although effective, execution-based metrics are often expensive and slow to compute. CodeScore[13]. was proposed as a faster, execution-free alternative that offers comparable performance. The performance of large language models (LLMs) for correctness evaluation varies according to task difficulty. For instance, Coignion et al. [24]found that LLMs performed better on simpler problems based on the Pass@k scores. In test generation, Straubinger et al. [14]demonstrated that LLMs outperform traditional tools, such as Pynguin, in fault detection and coverage, and that iterative refinement enhances the quality of test suites. However, they also noted that LLMs often increase test complexity without addressing the underlying issue and may introduce side effects, such as modifying unrelated codes. These findings suggest that most correctness-focused evaluations neglect the broader context of whether improvements in correctness come at the expense of other qualities such as reduced readability, maintainability, or security.

Security is another critical dimension, particularly in production environments. Several approaches have aimed to integrate security awareness into the code generation process of these models. For example, SeCuCoGen [2] incorporates security knowledge through in-context learning to reduce vulnerabilities. AutoSafeCoder [15] introduced a multiagent system for secure code generation. Benchmarking efforts like those by Peng et al. [25]and Wang et al. [26]provide datasets and metrics for evaluating the security of LLM-generated code. Kharma et al. [27]curated 200 security-critical tasks for in-depth analysis. Although security-focused generation is gaining traction, these efforts are typically separated from correctness and maintainability considerations. Studies rarely assess how improving one attribute, such as correctness, impacts security, which is a gap that this study aims to explore.

Maintainability is crucial for the long-term usability and reliability of software systems. Dillmann et al. [28]present a study on evaluating LLMs for assessing code maintainability. The results showed that when controlling for LLOC (Logical Line of Code), the cross-entropy computed by the LLMs was a predictor of maintainability at the class level, with higher cross-entropy indicating lower maintainability. However, this relationship was reversed when the LLOC was not controlled. Tools such as PMD [29] and SonarQube [30] are commonly used to detect code smells and structural flaws in software. CORE[31], an autonomous framework, addresses static analysis warnings and generates diverse and accurate code fixes with less effort than traditional tools. Liu et al. [32]proposed a repair pipeline that refined ChatGPT-generated code based on quality issues. Although improvements were observed, the authors also noted that refinements sometimes degraded the structural quality or introduced maintainability issues, highlighting a key limitation of refinement-based methods. The study examines the effects of enhancing maintainability on other characteristics, such as correctness and security.

Table 1. Comparison of the Used model, Prompt Technique, Benchmark, Task Types and Use of Iterative Refinement.

Reference	LLM	Prompt Technique	Iterative Refinement	Benchmark	Task Types
Bi et al. [10]	GPT-3.5 Turbo, Llama	Zero shot	Yes	HumanEval, MBPP, CodeEval	Data Structure and Algorithm, Object-Oriented Program
Moshin et al. [2]	GPT-4, Copilot, Gemini, Whisperer	Zero shot	No	LeetCode, Custom benchmark	Data Structure and Algorithm
Peng et al. [25]	GPT-4o, GPT-4o-mini, Gemini, Llama	Zero shot	No	CWEval-Bench	Security Flaws, Code vulnerability, API misuse, Logic Error
Liu et al. [5]	GPT-4o, GPT-4o-mini, Gemini, Llama	Chain of thought	Yes	MBPP, SecurityEval, Custom benchmark	Data Structure and Algorithm
Wang et al. [26]	GPT-3.5, CodeGen, InCoder	One shot	No	SecuCoGen	CWE-Aware Problem
Wadwa et al. [31]	GPT-3.5 Turbo, GPT-4	Zero shot	No	CodeQueries	Object-Oriented Program, API, Semantic Search
Coignion et al. [24]	Copilot, Llama, CodeGen, InCoder, CodeParrot, SantaCoder, WizardCoder, StarCoder	Zero shot	No	LeetCode	Data Structure and Algorithm

Tosi et al. [23]	GPT-3.5, GPT-4, Bard	Zero shot	No	Custom benchmark	Data Structure and Algorithm
Zheng et al. [22]	28 LLMs	Zero shot	No	HumanEval+ MBPP+ ClassEval LeetCode	Data Structure and Algorithm, Object-Oriented Program
Blyth et al. [33]	GPT-4o	Zero shot	Yes	PythonSecurityEval	CWE vulnerability

Table 2. Comparison of Programming Languages.

Reference	**Programming Languages**
Bi et al. [10]	Python
Moshin et al. [2]	C++, C#, Python
Peng et al. [25]	C, C++, Python, Javascript, Golang
Liu et al. [5]	Python, Javascript
Wang et al. [26]	Java, Python
Wadwa et al. [31]	Python
Coignion et al. [24]	Python
Tosi et al. [23]	Java
Zheng et al. [22]	Python
Blyth et al. [33]	Python

Table 3. Comparison of Evaluation Metrics, Quality Attributes, and Analysis Method.

Reference	**Evaluation Metrics**	**Quality Attributes**	**Analysis Method**
Bi et al. [10]	Pass@k	Correctness	-
Moshin et al. [2]	Code Security Risk Assessment, Static	Security	Static Scan

	Application Security Testing		
Peng et al. [25]	Func@k, Func-sec@k	Security	-
Liu et al. [5]	10-Dimensional Assessment, Relative Percentage Improvement, Distribution of code quality score improvement per iteration, Human spot checking	Correctness, Security, Maintainability	Manual, Static Scan
Wang et al. [26]	CodeBleu@k, sec@k	Security	Static Scan
Wadwa et al. [31]	Success Rate of fixed code and false positive reduction	Correctness, Security, Maintainability	Static Scan
Coignion et al. [24]	Pass@k	Correctness	Static Scan
Tosi et al. [23]	Test cases, static analysis	Correctness, Maintainability	Static Scan
Zheng et al. [22]	RACE Score	Correctness, Maintainability	-
Blyth et al. [33]	Static analysis, functional correctness, code fitness score.	Correctness, security, reliability, maintainability, readability	Static scan

2.3 Comparative Evaluations

To better understand the landscape of LLM-based code generation research, Table 1 summarizes recent studies across four key axes: the evaluated LLMs, prompting or refinement techniques used, and benchmarks and task types considered. This comparative view highlights the focus areas of prior research and, more importantly, where gaps remain.

As shown in Table 1, most studies evaluate GPT-based models (e.g., GPT-3.5 and GPT-4), with far fewer including models such as Claude, Gemini, and LLaMA. Prompting techniques are

often limited to zero-shot or few-shot learning, with iterative refinement used in only a handful of studies [7, 9]. Although benchmarks such as HumanEval and MBPP dominate, they primarily assess the correctness of short algorithmic tasks. Only a few studies (e.g., [2, 22, 23]) have incorporated project-level, OOP, or security-focused tasks.

Notably, two studies (from Table 1) have performed comparative evaluations across multiple LLMs using iterative refinement strategies. Only one study[31], has explored whether different task types, such as algorithmic challenges, object-oriented design, or data structure manipulation, impact LLM performance across quality dimensions. Moreover, benchmarks that include both maintainability and security metrics are rare (e.g.,[5, 31, 33]).

Tables 2 and 3 synthesize the findings across the selected studies, comparing them by programming language, prompting strategy, evaluation method, and quality attributes. This highlights that most studies focus on a narrow set of languages (primarily Python), quality dimensions (usually only correctness), and analysis method (static scan). Nevertheless, the evaluation employs a range of benchmarks, including established ones such as Pass@k, Func@k, and CodeBleu@k, as well as custom benchmarks.

To address these limitations, our study conducts a multidimensional evaluation of iteratively refined code generated by multiple LLMs (e.g., GPT-4, Claude, Gemini), spanning diverse programming task types in JavaScript. JavaScript was chosen for this study based on popularity, core features (web and mobile development) and job demand[34]. Our goal was to assess not only correctness but also maintainability and security and to understand how different tasks and refinement steps influence these quality dimensions.

3. Methodology

This study seeks to examine the quality of code generated by various LLMs under iterative refinement. Additionally, it aims to identify correlations between quality dimensions of the generated code across different task types. This evaluation focuses on three key software quality dimensions, including correctness, security, and maintainability, and studies how these dimensions vary across different models and task types.

The experiment used a curated dataset of JavaScript programming tasks categorized by task nature: data structures and algorithms, object-oriented programming, and web development. Three widely used LLMs, GPT (OpenAI), Gemini (Google Deepmind), and Claude (Antropic), were selected for evaluation owing to their current relevance and code generation capability, as discussed in the following sub-section[35].

For each task, an initial prompt was provided to each model (see Figure 1), followed by an iteration of refinement. The results from each iteration are assessed using a combination of test suites (for correctness), static analysis tools (to assess maintainability and security), and a structured comparative analysis of results across models, iterations, and quality attributes was performed.

This methodology focuses on a multidimensional understanding of how different LLMs perform over iterative refinement and discovering emerging patterns between quality attributes. This framework also examines whether the nature of a task influences refinement trajectory and outcome quality.

3.1 Model Selection

To facilitate a comprehensive and meaningful evaluation of large language models (LLMs) for iterative refinement, this study employs three state-of-the-art commercial models pertinent to real-world development workflows[36]. Based on current ranking from various sources, Gemini and Claude models are shown to excel in programming tasks, followed by GPT model [37, 38]. Therefore, the selected models originate from three distinct sources and versions. Justifications for selecting these models are provided below.

GPT-4.1 (OpenAI): This model demonstrates exceptional performance in software engineering tasks, achieving a score of 54.6% on the SWE-bench, surpassing both GPT-4o and GPT-4.5, thereby establishing itself as a leading model for code generation[39].

Claude 4 Opus (Anthropic): This model was included due to its current popularity among models chosen for code generation. It attained a score of 79.4% on the SWE-bench, outperforming Claude-3.7 Sonnet[40].

Gemini 2.5 Pro (Google Deepmind): This model was selected based on its code generation capabilities and represents the latest models from Google. It exhibited superior performance on LiveCodeBench, with an improvement of 43.7% over Gemini-1.5 Pro[41].

3.2 Task Selection

We did not intend to use well-known benchmarks, such as HumanEval and MBPP, as most LLMs could be familiar with, and trained on these datasets. Therefore, we aim to use a real-world problem dataset and custom benchmarks to obtain truthful results. The dataset consists of 30 JavaScript Programming tasks from an online source[1], and the tasks vary between easy and medium levels, with three different categories (Data Structure and Algorithm, Web Development, OOP) (see Appendix A).

[1] https://www.w3resource.com/javascript/javascript.php

3.3 Prompting Strategies

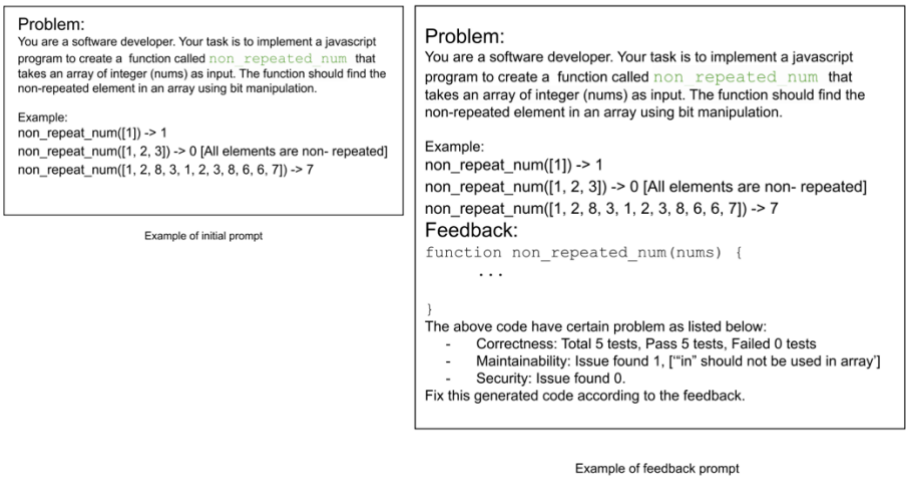

Figure 1. Example Prompt.

The experiment is conducted by inputting each task into the prompt for a large language model (LLM). As illustrated in Figure 1, the initial prompt serves as the foundation for all tasks across the model. As mentioned in Table 1, where zero-shot prompting dominates most research, we use few-shot prompting as it provides examples for each scenario that generated code should perform. Following each iteration, a feedback prompt is generated by combining feedback from the test suite and static analysis against the initial prompt, offering suggestions and guidelines to rectify or enhance the generated code.

If the generated code succeeds, the iteration will conclude, and the results will be collected. Otherwise, the cycle will continue until all criteria are met or until the fifth iteration is reached.

Table 4. Comparison of Number of Iterations.

References	**Number of Iteration**
Bi et al. [10]	3
Liu et al. [5]	5
Chen et al. [11]	8+
Madaan et al. [4]	4

As indicated in Table 4, the number of iterations exhibits variability. Satyapriya et al. [19]conducted an experiment demonstrating that prompts incorporating both the task definition and the question tend to perform suboptimal beyond the fifth iteration. Consequently, we have

determined that the fifth iteration represents the maximum, as it provides a reasonable balance in terms of truthfulness, accuracy, and the number of iterations.

3.4 Quality Evaluation and Metrics

This experiment uses a Python script to automate the process, as most LLMs are well integrated with Python. The code generated from the given prompt was tested using Jest [42] and analyzed using SonarQube for security and maintainability measurements[43]. We provide additional details on the measures below.

- Jest: A popular JavaScript testing framework used for executing test cases to ensure the functional correctness of the code. The results from Jest framework show the total number of tests that have passed or failed, which are used to quantify the correctness of the code. All the test suites are manually created, and the number of test cases varies by task (see Appendix A).

- SonarQube: SonarQube encompasses 315 rulesets for JavaScript, comprising 170 maintainability rules and 13 security rules. The remaining 132 are categorized as reliability rules (see Appendix B).

Each result from the test cases and static analysis tools was considered as feedback. The evaluation for all three dimensions (correctness, security, and maintainability) were set as follows:

- **Correctness**: Percentage of test cases passed.
- **Security**: Percentage of security rules that did not contain security violations in SonarQube
- **Maintainability**: Percentage of maintainability rules that did not contain security violations in SonarQube

3.5 Analysis Framework

To address Research Question 1 (How do different LLMs perform under iterative prompt refinement?), we calculated quality attribute scores (correctness, maintainability, security, overall) at each iteration and visualized the trajectories using combination charts (clustered bars for the three dimensions with an overall line). This facilitated direct examination of monotonic improvement, plateau, or variability across iterations for each model.

For Research Question 2 (Does improving one quality dimension, such as correctness, in the LLM-generated JavaScript code through refinement correlate with another, such as maintainability?), we analyzed iteration-to-iteration deltas at the task level. For each model × task, we employed SQL window functions to compute differences between consecutive iterations (Δcorrectness, Δmaintainability, Δsecurity). We then calculated overall delta correlations per model (e.g., r(Δc,Δm), r(Δc,Δs), r(Δm,Δs)) by aggregating across all tasks and steps; correlations

were reported as N/A when a metric exhibited no variance (e.g., security at ceiling). We summarized stepwise improvement/degradation rates. The proportion of task transitions with $\Delta>0$, $\Delta<0$, $\Delta=0$, by step (0→1, 1→2, 2→3, 3→4) and model, to quantify where gains or regressions concentrate across the refinement process. Optionally, we inspected step-specific delta correlations when variance allowed.This approach addresses overall trends rather than single-task idiosyncrasies and directly answers "between each iteration how it improves or degrades."

Finally, for Research Question 3 (Does the nature of the task affect the quality dimension of the generated JavaScript code?), we identified the task as the primary unit of analysis. For each model × task combination, scores related to correctness, maintainability, security, and overall quality were averaged across iterations to calculate a task-level mean. Subsequently, tasks were classified into three categories: data structures and algorithms (dsal), object-oriented programming (OOP), and web development (web_dev), with n = 10 tasks per category per model. Initially, we present descriptive statistics for each model × category to visualize overall patterns, which are illustrated in bar charts and a concise summary table. To formally evaluate whether correctness varied by task category, we conducted a Kruskal–Wallis H test separately for each model. In cases where the omnibus test indicated significance, we proceeded with pairwise Mann–Whitney U tests between categories, applying Holm correction to adjust for multiple comparisons. Effect sizes for the Kruskal–Wallis tests were reported as epsilon-squared (ε_H^2). Due to frequent ceiling values, maintainability and security scores exhibited minimal or no variance and were thus excluded from inferential testing. For these dimensions, results were summarized descriptively only.

4. Results

This section presents the results of the evaluation conducted to address the three research questions. Results are organized by research question, with visualization supporting the analysis.

4.1 Research Question 1 Results

RQ1: How do different LLMs perform under iterative prompt refinement?

To address this research question, we examined the change in quality attribute scores across iterations of refinement. Combo chart was used to visualize trends in correctness, maintainability, security, and overall quality. By comparing performance across iterations, the analysis [Figures 2-4] highlights whether iterative refinement consistently improves model outputs, produces negative returns, or introduces variability in performance across different models.

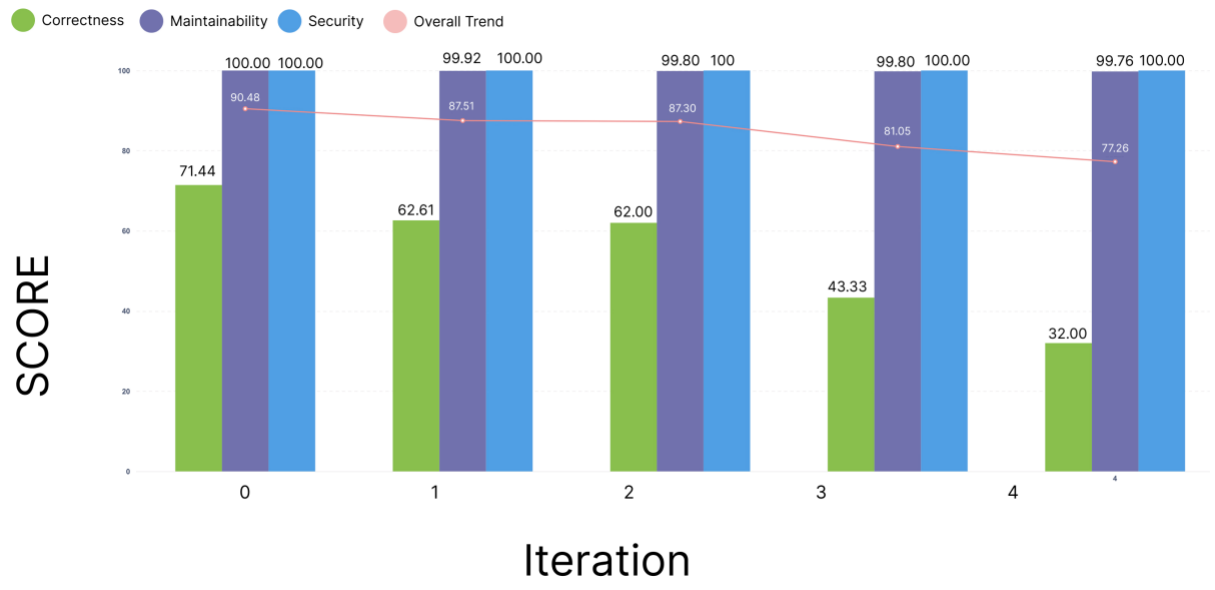

Figure 2. Claude 4 Opus Performance

Figure 2 shows the performance of Claude 4 Opus across 5 iterations of prompt refinement. Security remained consistently at 100 throughout, showing no variation across iterations. While maintainability experienced a slight decline following the initial iteration, it exhibited a modest decrease in the final iteration. Correctness, by contrast, declined steadily, dropping from 71.44 at iteration 0 to 32 at iteration 4. The overall quality score, plotted as a line, also decreased across iterations, falling from approximately 90 to 77. This trend suggests that while maintainability and security were invariant, reductions in correctness contributed directly to a decline in overall quality.

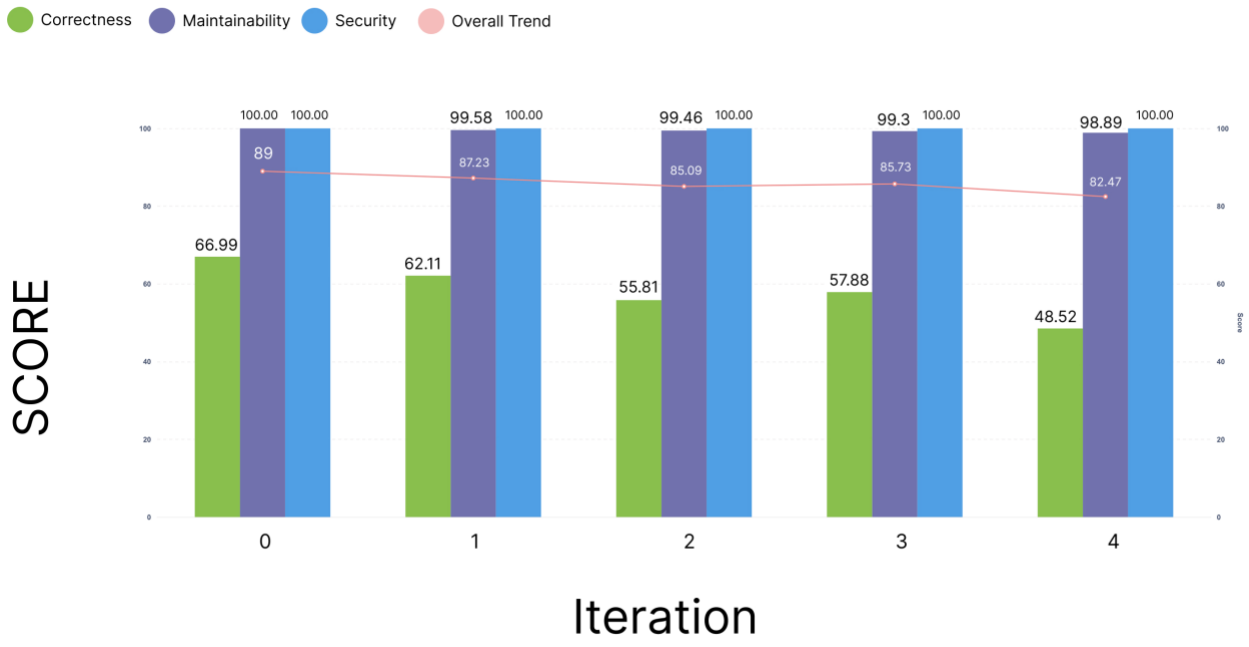

Figure 3. GPT 4.1 Performance

Figure 3 shows the performance of correctness, maintainability, and security across five iterations for GPT 4.1. Over time, there is a slight decline in maintainability. In contrast, security consistently remained at ceiling level, exhibiting no variation. Correctness decreased steadily from approximately 67 at iteration 0 to 48 by iteration 4. The overall quality score, represented by the line, declined only modestly (from about 89 to 82), reflecting the stability of maintainability and security despite the downward trend in correctness. In comparison to Claude, both models exhibited a consistent decline in correctness; however, the extent of this decline was more pronounced for Claude.

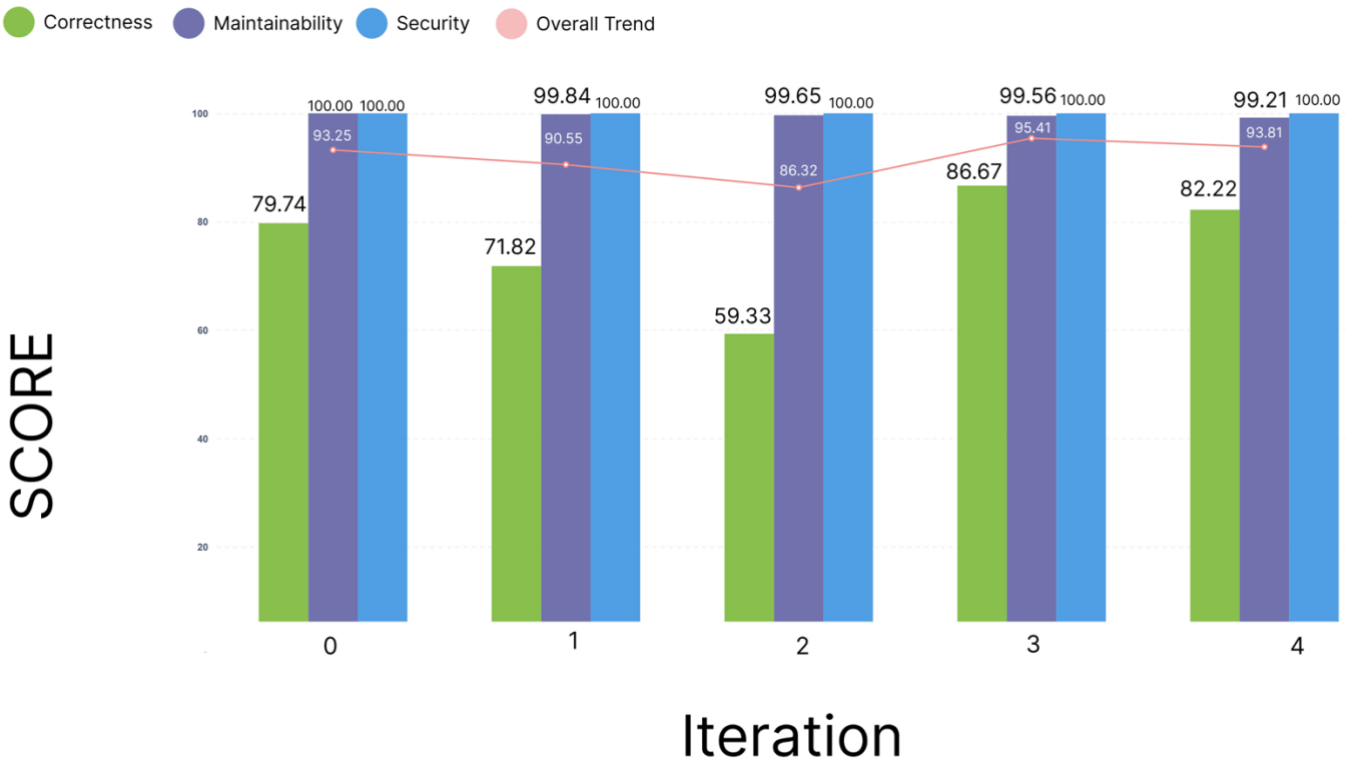

Figure 4. Gemini 2.5 Pro Performance

Figure 4 illustrates the changes in correctness, maintainability, security, and overall quality across five iterations of Gemini 2.5 Pro. Security levels remained consistently high. However, following iteration 0, there was a slight decline in maintainability. Correctness, however, exhibited more fluctuation, falling from 79.74 at iteration 0 to 59.33 at iteration 2, followed by a strong recovery to 86.67 at iteration 3 and a slight decrease to 82.22 at iteration 4. Overall quality scores followed a similar trajectory, reaching their lowest point at iteration 2 (86.32) before peaking at iteration 3 (95.41) and stabilizing just below that value. In contrast to the net declines observed for Claude and GPT, the overall quality followed a similar trajectory, reaching a minimum of 86.32 and peaking at 95.41, while Claude and GPT exhibited consistent declines throughout.

Table 5. Friedman's Test Result

LLM model	iterations_k	tasks_n	chi_square	p_value	kendalls_W	effect_size
claude-4-opus	5	5	N/A	N/A	N/A	Large

gemini-2.5-pro	5	3	4.00	0.41	0.33	Medium
gpt-4.1	5	9	4.00	0.41	0.11	Small

We evaluated whether quality varied across iterations (0–4) for each model using Friedman's test at the task level. When the omnibus test was significant, it was followed by Wilcoxon signed rank pairwise comparisons with Holm correction.

For Claude 4 Opus, Friedman's test revealed a significant main effect of iteration on correctness. Post-hoc Wilcoxon tests (Holm-adjusted) indicated that early versus late iterations differed, confirming a monotonic decline in correctness (e.g., 0→3, 0→4 contrasts significant). Maintainability and security showed no meaningful change across iterations (ceiling/near-ceiling values), consistent with the descriptive trends.

For GPT 4.1, Friedman's test also demonstrated a significant iteration effect on correctness, though smaller in magnitude than Claude. Post-hoc tests identified differences concentrated in early steps (e.g., 0→1 and/or 0→2), with later contrasts non-significant after Holm adjustment. Maintainability exhibited only a slight drift, and security remained at ceiling, yielding no reliable effects in the Friedman tests for these metrics.

For Gemini 2.5 Pro, Friedman's test revealed a significant, non-monotonic pattern in correctness. Post-hoc results captured the dip–recovery observed descriptively: a decrease from early to mid-iterations (e.g., 0→2 significant) followed by a rebound (e.g., 2→3 significant). Maintainability showed only minor stepwise variation, and security remained high, resulting in no consistent effects across iterations for these metrics.

Across models, correctness is the primary dimension that changes with refinement. Claude declined steadily; GPT showed negative returns after early gains; Gemini exhibited a dip recovery trajectory. Maintainability and security were largely invariant, limiting inferential results for those metrics.

4.2 Research Question 2 Results

RQ2: Does improving one quality dimension, such as correctness, in the LLM-generated JavaScript code through refinement correlate with another, such as maintainability?

To address RQ2, pairwise correlations were calculated between correctness, maintainability, and security scores across tasks. Due to lack of variation in some metrics (e.g., security consistently scored 100 for many tasks), several tasks produced undefined correlations.

Table 6. Overall delta correlation per model

Llm Model	pair	r	p	n_pairs	p_holm	significant	effect_size

claude-4-opus	corr_dc_dm	0.10	0.57	36	1	false	Small
claude-4-opus	corr_dc_ds			36	1	false	N/A
claude-4-opus	corr_dm_ds			36	1	false	N/A
gemini-2.5-pro	corr_dc_dm	0.22	0.30	23	0.91	false	Small
gemini-2.5-pro	corr_dc_ds			23	1	false	N/A
gemini-2.5-pro	corr_dm_ds			23	1	false	N/A
gpt-4.1	corr_dc_dm	-0.14	0.33	50	0.99	false	Small
gpt-4.1	corr_dc_ds			50	1	false	N/A
gpt-4.1	corr_dm_ds			50	1	false	N/A

Table 6 presents the correlations between iteration-to-iteration changes (Δ) in correctness, maintainability, and security, aggregated across all tasks for each model. For each correlation, the number of iteration transitions contributing to the estimate is indicated by n pairs.

Claude-4-Opus: The correlation between iteration-to-iteration changes in correctness and maintainability was small and positive ($r = 0.10$, $p = 0.57$, $n = 36$). However, correlations involving Δsecurity were undefined due to zero variance.

Gemini-2.5-Pro: the correlation between Δcorrectness and Δmaintainability was moderately positive ($r = 0.22$, $p = 0.30$, $n = 23$), while Δsecurity pairs remained undefined.

GPT-4.1: the correlation between Δcorrectness and Δmaintainability was small and negative ($r = -0.14$, $p = 0.33$, $n = 50$), with Δsecurity pairs also undefined.

Across the models, the correlation between dimensions is generally small to moderate. Regarding correctness and maintainability, Claude and GPT exhibit positive correlations, whereas Gemini demonstrates a negative correlation. However, the Security dimension remains irrelevant and does not exhibit any significant correlation.

Table 7. Improvement vs Degradation rates per step × model

LLM Model	Step from	Correctness improvement percentage	Correctness degradation percentage	No change in Correctness percentage	Maintainability improvement percentage	Maintainability degradation percentage	No change in Maintainability percentage	Security improvement percentage	Security degradation percentage	No change in Security percentage
Claude 4 Opus	0 → 1	33%	0%	67%	0%	7%	93%	0%	0%	100%
Claude 4 Opus	1 → 2	40%	0%	60%	0%	0%	100%	0%	0%	100%
Claude 4 Opus	2 → 3	17%	0%	83%	0%	0%	100%	0%	0%	100%
Claude 4 Opus	3 → 4	0%	0%	100%	0%	0%	100%	0%	0%	100%
Gemini-2.5 Pro	0 → 1	45%	0%	55%	0%	18%	82%	0%	0%	100%
Gemini-2.5 Pro	1 → 2	20%	0%	80%	0%	17%	83%	0%	0%	100%
Gemini-2.5 Pro	2 → 3	50%	0%	50%	0%	0%	100%	0%	0%	100%
Gemini-2.5 Pro	3 → 4	0%	0%	100%	0%	33%	67%	0%	0%	100%
GPT-4.1	0 → 1	29%	0%	71%	0%	24%	76%	0%	0%	100%
GPT-4.1	1 → 2	15%	0%	85%	0%	0%	100%	0%	0%	100%
GPT-4.1	2 → 3	18%	0%	82%	0%	9%	91%	0%	0%	100%
GPT-4.1	3 → 4	0%	0%	100%	0%	11%	89%	0%	0%	100%

Table 7 presents a summary of the proportion of tasks that exhibited improvement, degradation, or no change in terms of correctness, maintainability, and security at each refinement step for each model. Nevertheless, the graph-aligned in RQ1 mean correctness experienced a decline across iterations. This decline can be attributed to the introduction of net decreases in subsequent steps, where the magnitude of a few negative changes surpassed several minor improvements. Furthermore, it is important to note that the chart averages all tasks at each iteration, whereas the stepwise rates specifically consider the overlap cohort.

Claude-4-Opus: correctness improved in 33% of tasks between iterations 0→1 and 40% between 1→2, with smaller gains at 2→3 (17%) and no improvement at 3→4. No degradations in correctness were observed. Maintainability exhibited minimal change, with slight degradations at 0→1 (7%) and remained unchanged thereafter. Security remained constant across all steps.

Gemini-2.5-Pro: correctness improved in 45% of tasks at 0→1, decreased to 20% at 1→2, increased again to 50% at 2→3, and did not improve further at 3→4. Maintainability degraded for 18% of tasks at 0→1 and 17% at 1→2, with no changes thereafter. Security remained unchanged across all steps.

GPT-4.1: correctness improved in 29% of tasks at 0→1, 15% at 1→2, and 18% at 2→3, with no improvements observed at 3→4. No degradations in correctness were recorded. Maintainability degraded slightly in 11% of tasks at 3→4, while remaining unchanged at earlier steps. Security was constant across all transitions.

4.3 Research Question 3 Results

RQ3: Does the nature of the task affect the quality dimension of the generated JavaScript code?

To address RQ3, tasks were grouped into three categories: Data Structure and Algorithn, Object-Oriented Programming (OOP), and Web Development. For each category, the average scores of correctness, maintainability, and security were calculated across all iterations and models. The results are presented in Bar chart, which compares the mean performance values across task types, allowing for an assessment of whether particular categories consistently yielded higher or lower quality outcomes.

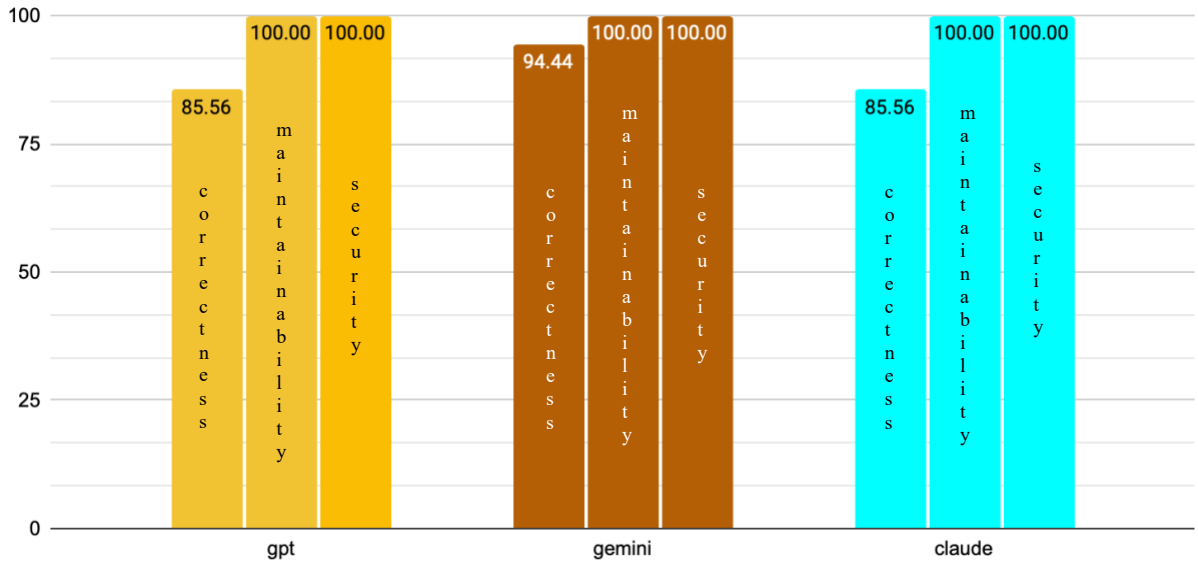

Figure 5. Quality Score in Fundamental JavaScript with Data Structure and Algorithm

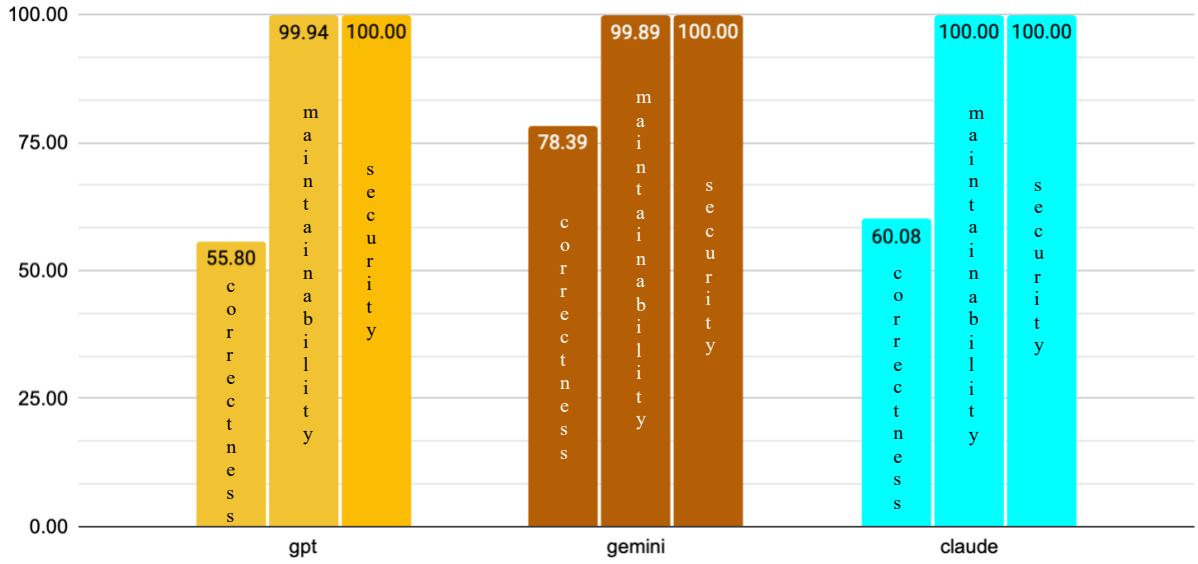

Figure 6. Quality Score in Object-Oriented Program

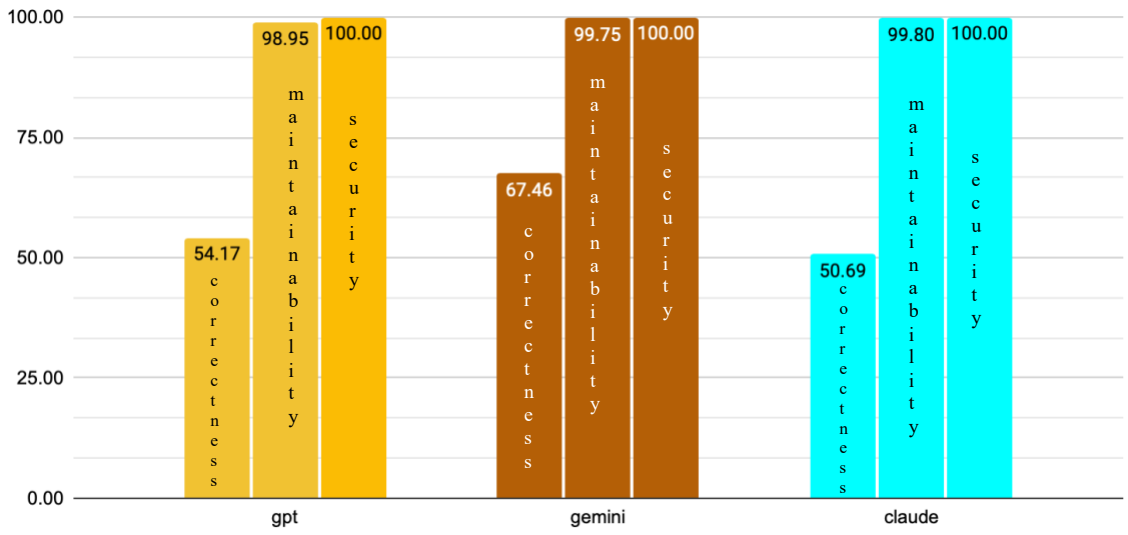

Figure 7. Quality Score in Web Development

Figures 5-7 presents the average quality scores all models in across 3 task categories. For GPT-4.1, Correctness differed notably between categories, with algorithmic tasks achieving the highest scores (≈ 86) compared to OOP (≈ 56) and web development (≈ 54). By contrast, maintainability and security scores remained consistently near 100 across all categories, reflecting ceiling effects in these dimensions. Overall, these results indicate that the nature of the task influenced correctness but not maintainability or security, with algorithmic problems proving easier for the models to solve correctly than OOP or web development tasks.

For the Gemini 2.5 model, performance across various task categories demonstrated a clear dependence on task type with respect to correctness, while maintainability and security consistently approached optimal levels. Specifically, in tasks related to data structures and algorithms, correctness was notably high at 94.44, with both maintainability and security achieving a perfect score of 100.00. Within the object-oriented programming category, a slight decrease in correctness was observed (≈ 78.39), although maintainability (≈ 99.89) and security (≈ 100.00) remained virtually unchanged. The most significant decline was noted in the web development category, where correctness decreased to 67.46; however, maintainability (≈ 99.75) and security (≈ 100.00) continued to exhibit remarkable stability. These results collectively suggest that the correctness of Gemini is highly sensitive to the nature of the task, being most robust in algorithmic contexts and least effective in web-based scenarios, while its performance in maintainability and security remains consistently high across all categories.

For the Claude 4 Opus model, performance varied primarily along the correctness dimension. Algorithmic tasks achieved the highest correctness scores (≈ 85), while OOP tasks showed a moderate level of correctness (≈ 60) and web development tasks the lowest (≈ 50). In contrast, maintainability and security scores were consistently strong across all categories, with averages close to 100. These findings suggest that Claude is also sensitive to task type in terms of correctness, while its maintainability and security outputs remain robust regardless of category.

Table 8. Summarize Task-level means by model and category

LLM Model	category	n_tasks	correctness_m	correctness_sd	maintainability_m	maintainability_sd	security_m	security_sd	overall_m	overall_sd
claude-4-opus	dsal	10	94.17	12.45	100	0	100	0	98.06	4.15
claude-4-opus	oop	10	70.14	28.04	100	0	100	0	90.05	9.35
claude-4-opus	web_dev	10	71.83	35.33	99.91	0.3	100	0	90.58	11.85
gemini-2.5-pro	dsal	10	96.88	8.84	100	0	100	0	98.96	2.95
gemini-2.5-pro	oop	10	87.39	15.64	99.95	0.15	100	0	95.78	5.24
gemini-2.5-pro	web_dev	10	77.83	27.2	99.89	0.3	100	0	92.58	9.11
gpt-4.1	dsal	10	94.17	12.45	100	0	100	0	98.06	4.15
gpt-4.1	oop	10	62.38	26.42	99.95	0.15	100	0	87.45	8.8
gpt-4.1	web_dev	10	68.33	35.15	99.41	1.2	100	0	89.25	11.75

Table 8 presents a summary of task-level mean scores for correctness, maintainability, security, and overall quality, categorized by model and task type (Data Structure and Algorithm, OOP, Web Development). To formally assess differences in correctness, Kruskal–Wallis H tests (see Table 12 were conducted for each model, followed by pairwise Mann–Whitney U tests with Holm correction (see Table 13-15).

For Claude-4-Opus, the Kruskal–Walli's test suggested a trend towards differences among categories in correctness, H (2) =5.13, p=0.077, ε_H^2=0.116. However, post hoc tests revealed no significant pairwise differences after Holm adjustment, although Data Structure and Algorithm tended to outperform OOP (raw p=0.024, Holm-adjusted p=0.071). Maintainability and security scores were consistently 100 across categories, precluding inferential testing.

For Gemini-2.5-Pro, no significant differences in correctness were observed across categories, H (2)=0.42, p=0.812, ε_H^2=−0.06. Maintainability and security scores were near ceiling values, with minimal variation insufficient for meaningful testing.

For GPT-4.1, the Kruskal–Wallis test indicated a significant effect of category on correctness, H (2) =8.37, p=0.015, ε_H^2=0.236. Post hoc tests demonstrated that Data Structure and Algorithm tasks scored significantly higher than OOP (U=78.5, pHolm=0.071) but not web_dev (pHolm=0.220). The comparison between OOP and web_dev was not significant (p=0.787). Maintainability and security scores remained at ceiling, rendering tests uninformative.

Collectively, the tests reveal a consistent categorical signal in correctness for GPT-4.1, a weaker trend for Claude-4-Opus, and no discernible effect for Gemini-2.5-Pro.

5. Discussion and Implications

RQ1: How do different LLMs perform under iterative prompt refinement?

The initial research question investigated the performance of various large language models (LLMs) under iterative prompt refinement. The findings indicated that refinement did not yield a consistent trajectory of improvement across models. Instead, patterns varied from a steady decline to high variability, highlighting that refinement can both enhance and impair output quality depending on the model. Moreover, increasing the iteration of prompt refinement does not necessarily guarantee enhanced code quality. Practically, this implies that refinement functions as a model-specific hyperparameter rather than a universally advantageous procedure[11]. Previous studies frequently report benefits from refinement or retrieval-augmented prompting; however, they also acknowledge that excessive iteration can detrimentally affect quality[19]. This study further confirms that refinement can either aid or impair outcomes depending on the model, and it provides multi-dimensional evidence indicating that improvements in one area do not consistently translate to others.

For Claude 4 Opus, correctness consistently declined across iterations, decreasing from 71.44 at iteration 0 to 32 at iteration 4. This downward trend suggests that repeated refinement introduced cumulative instability into correctness rather than leading to improved solutions. As

maintainability and security remained stable, the decline in correctness directly contributed to the reduction in overall quality. These findings underscore a limitation: refinement did not assist Claude in converging on higher-performing solutions but instead degraded correctness over time. A plausible explanation for Claude's consistent decline is the phenomenon of prompt drift [44]. As instructions undergo repeated modifications, the prompt may accumulate conflicting objectives or excessively prescriptive constraints, which can divert the model from its initial, higher-quality performance. Additionally, longer and more complex prompts increase the likelihood of instruction overshadowing [45], where key requirements become obscured or contradicted.

For GPT-4.1, the pattern was less severe but still indicative of negative returns. Correctness gradually declined from 67 to 48 across iterations, while maintainability and security remained at ceiling values. Consequently, the overall quality score exhibited only a modest decrease. This suggests that GPT maintained consistent structural quality, but refinement failed to substantially enhance correctness. In practice, refinement for GPT-4.1 may yield marginal gains initially (iteration 2 → 3 slightly improvement in correctness), but additional iterations do not appear to meaningfully improve performance. A plausible explanation for the initial modest improvements observed in GPT-4.1, followed by a decline or plateau, is that the refinement process rapidly exhausts the easily attainable clarifications. Subsequently, it may begin to reiterate or overly constrain the task, leading the model to make unnecessary edits that do not enhance test outcomes. As prompts become more extended, essential requirements may become diluted or contradictory, a phenomenon known as "instruction overshadowing"[45]. Given that correctness is the sole dimension with significant variability (with maintainability and security already at their peak), even minor logical regressions can result in net losses. Previous research on iterative or self-refinement prompting has reported similar negative returns and occasional overthinking effects[19, 33]. While early iterations are beneficial, later ones often merely rearrange reasoning without adding value or introduce errors such as off-by-one or specification drift.

In contrast, Gemini 2.5 Pro exhibited the most variability across iterations. Correctness sharply decreased to 59 at iteration 2 but then rebounded to 86 at iteration 3, before slightly decreasing to 82 at iteration 4. Overall quality followed this non-linear pattern, with its lowest point at iteration 2 and a peak at iteration 3. While security levels remained consistently high, there was a slight decline in maintainability over time as iterations progressed. This volatility suggests that Gemini is more sensitive to incremental prompt changes: refinement can unlock significant performance gains, but it can also destabilize correctness at intermediate iterations. A plausible explanation for the dip–recovery pattern observed in Gemini is its pronounced sensitivity to minor prompt modifications and decoding configurations[46]. Initial refinements can reveal genuinely beneficial constraints, resulting in significant improvements. However, alterations made mid-iteration may redirect the reasoning trajectory, causing temporary regressions before subsequent prompts re-stabilize the solution. This sensitivity aligns with previous findings regarding the effects of prompt order and wording, as well as the volatility of self-refinement[19]. Iterative edits can sometimes enhance the depth of reasoning, while at other times they may lead to specification drift or fragile corrections that disrupt earlier logic. Consequently, gains and losses may cluster by step rather than accumulate in a linear fashion.

Collectively, these findings demonstrate that iterative refinement does not guarantee monotonic improvement in model outputs. Claude's steady decline, GPT's negative returns, and Gemini's fluctuating trajectory illustrate that refinement outcomes are heavily dependent on model characteristics. Prior research has posited that iterative prompting can guide LLMs toward higher-quality solutions, but these results complicate that narrative: refinement may be beneficial for some models in certain contexts, yet counterproductive or unstable in others.

Recommendation: From a practical perspective, this implies that developers should adopt model-specific refinement strategies rather than applying a uniform approach. For Claude, excessive refinement may be detrimental, suggesting that shorter iteration cycles or alternative prompting methods are preferable. For GPT, refinement beyond one or two iterations yields limited returns, emphasizing the importance of efficiency. For Gemini, refinement carries both risks and rewards, indicating that users may benefit from closely monitoring outputs and selectively adopting iterations that demonstrate genuine improvement.

RQ2: Does improving one quality dimension, such as correctness, in the LLM-generated JavaScript code through refinement correlate with another, such as maintainability?

The second research question investigated the relationships among quality dimensions during iterative refinement, specifically examining whether improvements or degradations tend to cluster at stages. The findings indicate that these relationships are contingent upon the model, with most measurable improvements in correctness occurring early in the refinement process. Across models, maintainability and security exhibited limited variation, which constrained correlation estimates and concentrated the actionable signal in correctness. In practice, refinement predominantly enhances correctness. Subsequent iterations seldom alter outcomes, and when they do, such changes are not systematically coordinated across various dimensions. Given that maintainability and security exhibit minimal variation (due to ceiling effects), their correlations with correctness are weak or indeterminate. Consequently, the most reliable and decision-relevant indicator is the change in correctness, particularly between iterations 0 to 1 (and occasionally from 1 to 2). Previous research [4] on iterative or self-refinement prompting frequently indicates initial improvements followed by diminishing returns. Furthermore, it is observed that optimizing a single objective, such as test pass rate, does not consistently enhance other attributes.

For Claude 4 Opus, the association between changes in correctness and maintainability was minimal, with a small positive correlation. Stepwise rates indicated that correctness gains were front-loaded, being largest at steps 0→1 and 1→2, followed by tapering and an eventual plateau, with no improvements observed at step 3→4. Maintainability and security remained effectively invariant across steps. Collectively, Claude's refinement dynamics suggest that while early edits can yield modest correctness gains, subsequent iterations contribute little and do not materially affect other quality dimensions. In practice, prolonged refinement offers diminishing returns for Claude and does not introduce systematic maintainability or security side effects.

For GPT-4.1, changes in correctness exhibited a slight trade-off with maintainability, evidenced by a small negative correlation, while security again showed negligible movement. Stepwise,

correctness improvements were modest at step 0→1, smaller at steps 1→2 and 2→3 and absent at step 3→4; degradations in correctness were rare. Maintainability was largely stable, with a small late-stage dip. This pattern implies that GPT's early refinements can incrementally improve correctness without widespread destabilization, but additional iterations risk effort without benefit and may introduce small structural regressions. Practically, GPT achieves optimal performance when subjected to concise refinement loops consisting of one or two iterations followed by evaluation, rather than through continuous adjustments. The pattern observed in GPT-4.1 is characterized by initial substantial gains, followed by a tendency towards excessive constraint in subsequent prompts. After the initial iterations, refinements often involve reiterating or tightening requirements, which can dilute essential instructions or lead the model to make inconsequential edits: minor adjustments that do not affect test outcomes and may slightly compromise maintainability. This phenomenon accounts for the weak negative correlation between correctness and maintainability, while security remains unaffected. Previous research on iterative or self-refinement prompting similarly indicates diminishing returns and occasional over analysis in later stages, with improvements in correctness not consistently translating to enhancements in structural qualities [33].

Gemini 2.5 Pro demonstrates that refinement effects are predominantly concentrated in the initial stages, primarily affecting correctness, with a volatile intermediate phase and a subsequent plateau. Correctness improved for 45% of tasks from 0 to 1, decreased to 20% from 1 to 2, increased again to 50% from 2 to 3, and showed no improvement from 3 to 4, indicating a pattern of early gains, mid-phase variability, and eventual stabilization. Maintainability exhibited minor early degradations (18% from 0 to 1; 17% from 1 to 2) with no further changes, while security remained constant throughout all stages. Correspondingly, the overall delta-correlation between correctness and maintainability was moderately positive but not statistically significant ($r = 0.22$, $p = 0.30$, $n = 23$), and all pairs involving security were undefined due to zero variance. These findings suggest that Gemini's co-movement across dimensions is limited and unreliable, with correctness providing the actionable signal, while structural metrics remain largely invariant after the initial iterations. Practically, the refinement of Gemini should be approached as a process that is both front-loaded and correctness oriented. It is essential to prioritize the transition from 0 to 1 (and, when beneficial, from 2 to 3) while carefully examining the transition from 1 to 2 for mixed outcomes. Implement early-stop mechanisms and rollback-to-best strategies once improvements cease to progress and continuously monitor maintainability to address minor early regressions that may accompany correctness modifications. Given that security remains unchanged under these conditions, decisions should be based on correctness deltas and targeted assessments of maintainability, rather than anticipating simultaneous multi-dimensional improvements.

Collectively, these findings suggest that iterative refinement does not induce uniform co-movement among quality dimensions. Instead, model-specific patterns emerge: Claude remains largely stable beyond initial gains; GPT exhibits slight tension between correctness and maintainability alongside diminishing returns; Gemini demonstrates changes that are primarily front-loaded and driven by correctness, with minimal coupling across dimensions, as evidenced by a small, non-significant correlation between changes in correctness and maintainability, and consistent security**.** The concentration of correctness gains in the first refinement step across models supports the notion of front-loaded benefits and diminishing returns thereafter. Finally,

the prevalence of ceiling effects in maintainability and security limited the detection of subtler trade-offs; future research should employ more sensitive measures for these dimensions to fully characterize interdependencies during refinement.

Recommendation: From a practical perspective, these dynamics advocate for adaptive, model-aware refinement strategies. For Claude, iterations should be kept short and targeted. For GPT, refinement should be capped at one to two steps to avoid unnecessary cycles and minor structural regressions. For Gemini, anticipate initial gains, followed by varied mid-phase outcomes, and eventual stabilization. It is advisable to employ checkpointing and rollback strategies, as well as early stopping once progress ceases. Additionally, it is important to monitor maintainability, focusing on minor early regressions rather than expecting simultaneous advancements across multiple dimensions.

RQ3: Does the nature of the task affect the quality dimension of the generated JavaScript code?

The analysis of task categories reveals that task type significantly influences correctness, while its impact on maintainability and security is minimal, primarily due to ceiling effects. The inferential tests demonstrate clear contrasts across models. In practical terms, this indicates that the type of task primarily influences correctness, while the apparent uniformity in maintainability and security reflects capped metrics rather than true equivalence across tasks. Data Structure and Algorithm problems align with the models' strengths, well-defined objectives and short dependency chains, resulting in higher correctness. In contrast, object-oriented programming (OOP) and web development introduce abstractions, multi-component coordination, and state, leading to a decrease in correctness. This pattern mirrors previous evaluations that over-sample algorithmic tasks [47], thereby overstating model capability in real software work. Studies that include tasks heavy in design and frameworks [48] similarly report lower correctness and greater variance, while aggregate structural scores often reach saturation and fail to capture subtle issues.

For GPT-4.1, the Kruskal–Wallis's test confirmed a significant effect of task category, with Data Structure and Algorithm tasks outperforming OOP tasks and trending above web development. This supports the descriptive finding that GPT handles algorithmic-style problems more effectively than abstraction-heavy or integrative coding tasks. The effect size (ε_H^2=0.236) indicates a moderate category impact on correctness[24]. A plausible explanation for the superior performance of GPT-4.1 on Data Structures and Algorithm tasks is that these problems align with the model's inherent strengths and the evaluation framework. Such tasks typically involve specifying a single function with well-defined input-output contracts, short dependency chains, and success criteria that are directly assessed through unit tests. Consequently, local reasoning, including loops, conditionals, and edge cases, is sufficient. In contrast, object-oriented programming (OOP) and web development tasks necessitate architectural decisions and coordination across components, such as designing classes and interfaces, maintaining invariants across methods, managing state and lifecycles, and adhering to API or framework conventions. These tasks introduce longer-horizon dependencies, more implicit requirements, and a larger error surface, where minor misunderstandings can propagate and diminish test pass rates despite a sound structure. This pattern aligns with previous reports indicating that large language models excel in contest-style or LeetCode-like tasks but exhibit weaker and more variable performance

on software engineering tasks that involve multi-module design, framework usage, and environmental assumptions [49].

The results from Claude-4-Opus indicates a similar directional pattern to that observed with GPT, wherein on Data Structures and Algorithm tasks tend to achieve higher correctness scores compared to object-oriented programming (OOP) tasks. However, the statistical evidence supporting this observation is weak (Kruskal–Wallis $p = .077$), likely due to limited variance and statistical power, as evidenced by ceilinged maintainability/security and a small sample size per category. Like GPT, on Data Structures and Algorithm tasks are characterized by short dependency chains and explicit input–output contracts, allowing for sufficient local reasoning and rewarding intended behaviour through unit tests. In contrast, OOP tasks necessitate architectural decisions involving classes, interfaces, and invariants, as well as coordination across methods, which involve long-horizon dependencies where minor misunderstandings can propagate, thereby reducing test pass rates even when the code appears structurally sound. This tendency for Data Structures and Algorithm tasks to outperform OOP tasks aligns with previous observations that large language models (LLMs) perform more reliably on contest-style or function-synthesis problems than on software-engineering tasks that require multi-module design or adherence to framework conventions. Given the marginal p-value, the effect should be interpreted as suggestive rather than conclusive, yet it implies practical guidance that Claude, akin to GPT, is comparatively better suited for algorithmic utilities than for abstraction-intensive OOP tasks unless stronger scaffolding, such as tests, contracts, or architectural templates, is provided.

For Gemini-2.5-Pro, the Kruskal–Wallis's test did not reveal significant differences in correctness across task categories, despite descriptive means indicating that OOP was marginally higher than dsal or web. This pattern suggests that Gemini's correctness is relatively insensitive to task type within our experimental framework, or that substantial variability between tasks within categories (with $n = 10$ per category) diminished the power to detect differences. This result is consistent with the broader findings of this study: maintainability and security were near ceiling levels, offering minimal variance to assess category effects, thus positioning correctness as the primary evaluative measure. Previous studies on LLM code generation present mixed evidence regarding task-type sensitivity. Some report advantages in algorithmic problems, while others observe reduced gaps when prompts or scaffolds are optimized [49]. Our findings position Gemini towards the reduced-gap end of this spectrum. Practically, any perceived advantage in OOP for Gemini should be considered suggestive rather than definitive unless corroborated by larger samples or more sensitive structural metrics.

Collectively, these results underscore that algorithmic tasks remain the most tractable across LLMs, aligning with the training bias toward competitive programming and coding-challenge data. OOP and web development tasks, by contrast, introduce architectural and multi-component reasoning demands that these models handle less consistently. The absence of significant effects for maintainability and security reflects both their ceiling distribution and the limited sensitivity of the employed metrics.

Recommendation: From a practical standpoint, the findings reinforce that LLM deployment should be task-aware: GPT and Claude are better suited to algorithmic problem solving than to

OOP or web tasks, while Gemini appears more even across categories but less predictable. For evaluation design, the results stress the importance of including diverse task types; relying solely on algorithmic problems risks overstating competence, whereas OOP and web tasks reveal more realistic boundaries of current model capability.

Limitations & Threats to Validity

This section delineates potential limitations in our design and analysis, as well as their implications for interpretation. We categorize these threats according to standard validity dimensions. **Construct validity** examines whether our measures accurately capture the intended concepts (e.g., do our metrics genuinely reflect "maintainability" and "security"?). **Internal validity** pertains to alternative explanations or confounding variables within the study (e.g., prompt drift, instrumentation) that could influence the observed effects. **Statistical conclusion validity** evaluates whether our inferences are justified given the sample size, variance, ties/ceilings, and model comparison. **External validity** assesses the extent to which the results can be generalized beyond our specific tasks, models, and language. Finally, **Reliability and Reproducibility** assess non-determinism and model churn and environment dependencies.

Construct validity

- **Metric sensitivity**: The dimensions of maintainability and security often reached maximum values, resulting in minimal variance and obscuring genuine differences or trade-offs. Aggregated metrics in the style of SonarQube, particularly for brief, single-file tasks, may overlook nuanced structural issues.
- **Operationalization of "overall quality":** Aggregation may disproportionately emphasize dimensions with minimal variability while undervaluing correctness.

Internal validity

- **Prompt and iteration effect**: "prompt drift" during refinements and a limited iteration budget, may confound the outcomes of iterations.
- **Instrumentation:** unit tests may not comprehensively cover task intents, and the configuration and rule set choices in SonarQube can influence scores.
- **Anomalies or outliers:** isolated task errors (e.g., a task yielding inflated scores), can distort correlations.

Statistical conclusion validity

- **Power and ties:** With n=10 tasks per category and the presence of significant ties (ceiling metrics), the tests may lack sufficient power; several correlations were undefined due to zero variance.

- **Multiple Comparisons:** Post-hoc tests increase the risk of Type I error if adjustments are not made.

External validity

- **Tasksets and representativeness**: The experiment was conducted on predetermined set of tasks. While this provides a controlled environment for comparison, the number of tasks was relatively small and may not fully represent the diversity of the real-world programming problem challenges. Larger datasets and more variety of tasks may illustrate more comprehensive test of model capabilities.
- **Model Scope:** The study evaluated three advanced large language models (LLMs), namely GPT-4.1, Gemini 2.5 Pro, and Claude 4 Opus. Although these are leading models, they do not cover the entire spectrum of available models, particularly those specialized in code generation, such as Codex-style or open-source code LLMs. Consequently, the findings cannot be generalized to all LLMs without further validation.

Reliability and Reproducibility

- **Non-determinism and model churn**: Outputs generated by large language models (LLMs) are subject to variation based on decoding settings and updates from providers; consequently, results may exhibit changes over time.
- **Environment Dependencies**: Tool versions (e.g., SonarQube rules), test harnesses, and prompts affect outcomes.

6. Conclusion and Future Work

Conclusion

This study investigated the behaviour of large language models during iterative prompt refinement for JavaScript code generation, focusing on three research questions: (RQ1) How do different LLMs perform with iterative prompt refinement, (RQ2) Does improving one quality dimension (e.g., correctness) in the LLM-generated JavaScript code through refinement correlate with another (e.g., maintainability), and (RQ3) Does the nature of the task affect the quality dimensions of the generated JavaScript code. The results indicated that refinement did not uniformly enhance performance. Specifically, Claude exhibited a consistent decline in correctness across iterations; GPT showed modest initial improvements with diminishing returns; and Gemini demonstrated volatility, with mid-iteration declines followed by recovery. Cross-dimensional analyses revealed limited consistent co-movement, primarily due to ceiling effects in maintainability and security. However, Gemini occasionally displayed coupled shifts (Δcorrectness–Δmaintainability), GPT suggested a mild correctness–maintainability trade-off, and Claude remained largely unchanged. Task type predominantly influenced correctness (Data Structure and Algorithm > OOP/Web Development), while maintainability and security were near ceiling across categories.

These findings challenge the assumption that iterative refinement consistently enhances code quality [5, 10]. The effectiveness of refinement is contingent upon both the model and the task:

Claude derives minimal benefit from extended refinement, GPT's improvements are concentrated in the initial steps, and Gemini can achieve multi-dimensional enhancements but also experiences mid-course regressions. Practitioners should therefore adopt strategies that are model-aware, task-aware, and iteration-bounded, with a focus on validating correctness particularly in OOP and web development tasks where models demonstrate less reliability.

Although constrained by a finite task set and ceilinged structural metrics, this work provides a clearer understanding of refinement dynamics in practice.

Future Work

Given the limitations, there are several areas that can be expanded upon in future research.

Expanding Task Diversity: Future research should employ a broader and more representative set of programming tasks. In addition to single-file problems, tasks could encompass multi-module systems, projects necessitating integration with external libraries, or collaborative coding scenarios. Furthermore, utilizing contemporary JavaScript frameworks would provide greater insight into real-world challenges.

Refining evaluation metrics: To address ceiling effects and better capture nuanced quality differences, more granular evaluation methods are needed. Metrics such as code readability (ESlint) or cyclomatic complexity (jshint), could complement correctness, maintainability, and security.

Cross-Model and multi-agent approaches: While this study focused on comparing three proprietary LLMs, future research should encompass a broader selection of models, including open-source LLMs. Comparative studies could also explore multi-agent scenarios, where one model is employed for tasks related to correctness and another model is utilized for assessing different quality dimensions. This approach may uncover synergies that could surpass the capabilities of using a single model.

In conclusion, Future research should expand task diversity and employ more sensitive measures of maintainability and security to capture subtler trade-offs, thereby enabling more robust guidance for deploying LLMs in real software development workflows.

Appendices

Appendix A

We have 30 tasks in total which can be categorized into 3 main categories. The table is shown below.

Table 9. Task List for Fundamental JavaScript with Data Structure and Algorithm Problems

Fundamental JavaScript with Data Structure and Algorithm			
Task ID	Name	Difficulty Level	Number of Test cases
0	Filter_non_unique	Easy (1020 votes)	3
1	Reverse_a_number	Easy (13200 votes)	4
2	Find_longest_word	Easy (4680 votes)	3
3	Extract_unique_character	Easy (2120 votes)	6
4	Ternary_search	Medium (272 votes)	1
5	Difference	Medium (2900 votes)	1
6	twoSum	Medium (3310 votes)	1
7	Longest_common_substring	Medium (4760 votes)	1
8	Num_string_range	Medium (2448 votes)	1
9	isSorted	Medium (884 votes)	5

From Table 9, The description for each task is listed below.

Task 0: filter_non_unique

```
/**
 * * Write a JavaScript program to filter out non-unique values in an array.
 * * @param  {Array} arr array to filter
 * * @return {Array}     array with unique values
 * */
```

```
function filter_non_unique(arr){
    // TODO:
}
```

Task 1: reverse_a_number

```
/**
 *  * Write a JavaScript function that reverses a number.
 *  * @param  {number} n
 *  * @return {number}  reversed number
 *  */
function reverse_a_number(n){
    // TODO:
}
```

Task 2: find_longest_word

```
/** \
 *  * Write a JavaScript function that accepts a string as a parameter and finds the longest word within the string.
 *  * @param  {string} str
 *  * @return {string} longest word in the string
 *  */
function find_longest_word(str) {
    // TODO:
}
```

Task 3: extract_unique_character

```
/** \
 *  * Write a JavaScript function to extract unique characters from a string.
 *  * @param  {string} str
 *  * @return {string} unique characters in the string
 *  */
function extract_unique_characters(str) {
```

```
    // TODO:
}
```

Task 4: ternary_search

```
/** \
 *  * Write a JavaScript program to find the first index of a given element in an array using the ternary search
algorithm.
 *  * @param  {Array} arr array to search
 *  * @param {number} target element to find
 *  * @param {number} left starting index of the search
 *  * @param {number} right ending index of the search
 *  * @return {number} index of the element if found, -1 otherwise
 *  */
function ternary_search(arr, target, left, right) {
    // TODO:
}
```

Task 5: difference

```
/** \
 *  * Write a JavaScript function to find the difference between two arrays.
 *  * @param  {Array} arr1 first array
 *  * @param  {Array} arr2 second array
 *  * @return {Array} array containing elements present in arr1 but not in arr2
 *  * Example:
 *  * difference([1, 2, 3], [100, 2, 1, 10]) // returns [3, 10, 100]
 *  * difference([1, 2, 3, 4, 5], [1, [2], [3, [[4]]],[5,6]]) // returns [6]
 *  */
function difference(arr, arr2) {
    // TODO:
}
```

Task 6: twoSum

```
/** \
 *  * Write a JavaScript program to find a pair of elements (indices of the two numbers) in a given array whose sum
equals a specific target number.
 *  * @param  {Array} nums array of numbers
 *  * @param  {Array} target target number
 *  * @return {Array} array containing indices of the two numbers that add up to the target
 *  * Example:
 *  * twoSum([10,20,10,40,50,60,70], 50) // returns [2,3]
 *  */
function twoSum(nums, target) {
   // TODO:
}
```

Task 7: longest_common_starting_substring

```
/** \
 *  * Write a JavaScript function to find the longest common starting substring in a set of strings.
 *  * @param  {Array} arr1 array of strings
 *  * @return {string} longest common starting substring
 *  * Example:
 *  * longest_common_starting_substring(['go', 'google']) // returns 'go'
 *  */
function longest_common_starting_substring(arr1) {
   // TODO:
}
```

Task 8: num_string_range

```
/** \
 *  * Write a JavaScript function to fill an array with values (numeric, string with one character) within supplied bounds.
 *  * @param  {string} start start value (inclusive)
 *  * @param  {string} end end value (inclusive)
 *  * @param  {number} step step value
 *  * @return {Array} array of values within the specified range
 *  * Example:
 *  * num_string_range('a', 'd', 1) // returns ['a', 'b', 'c', 'd']
 *  */
function num_string_range(start, end, step) {
```

```
    // TODO:
}
```

Task 9: isSorted

```
/** \
 *  * Write a JavaScript program to check if a numeric array is sorted or not.
 *  * @param  {Array} arr - The array to check.
 *  * @return {number} 1 if sorted, -1 if reverse sorted, 0 if not sorted
 *  * Example:
 *  * isSorted([1, 2, 3, 4]) // returns 1
 *  * isSorted([4, 3, 2, 1]) // returns -1
 *  * isSorted([1, 3, 2, 4]) // returns 0
 *  */
function isSorted(arr) {
    // TODO:
}
```

Table 10. Task List for Web Development Problems

Web Development			
Task ID	Name	Difficulty Level	Number of Test cases
10	setupDropDown	Medium (1020 votes)	2
11	changebackgroundColor	Easy (2924 votes)	2
12	validateForm	Easy (5440 votes)	2
13	handleDragStart, handleDragOver, handleDrop	Medium (1100 votes)	6
14	copyText	Medium (136 votes)	2
15	Volume_sphere	Easy (3270 votes)	1
16	getSize	Medium (2310 votes)	1
17	toggleSwtich	Medium (2924 votes)	1

18	UpdateProgress	Medium (1564 votes)	1
19	preventDefaultFormSubmission	Medium (1836 votes)	1

From Table 10, The description for each task along with html are listed below.

Task 10: setupDropdown

```
<!DOCTYPE html>
<html>
 <head>
  <style>
   .dropdown {
    position: relative;
    display: inline-block;
   }
   .dropdown-button {
    background-color: #7fffd4;
    color: #333;
    padding: 10px;
    border: none;
    cursor: pointer;
   }
   .dropdown-options {
    display: none;
    position: absolute;
    background-color: #cc56ff;
    min-width: 120px;
    box-shadow: 0px 8px 16px 0px rgba(0, 0, 0, 0.2);
    padding: 0;
    margin: 0;
    list-style: none;
    z-index: 1;
   }
   .dropdown-option {
    padding: 10px;
    cursor: pointer;
```

```
      border-bottom: 1px solid #ccc;
    }
    .dropdown-option:last-child {
      border-bottom: none;
    }
  </style>
 </head>
 <body>
  <div class="dropdown">
    <button class="dropdown-button">Select a Subject</button>
    <ul id="dropdown-options" class="dropdown-options">
      <li class="dropdown-option">Mathematics</li>
      <li class="dropdown-option">English</li>
      <li class="dropdown-option">Physics</li>
    </ul>
  </div>
  <script src="./10.js"></script>
 </body>
</html>
```

```
/** \
 *  * Write a JavaScript program to create a dropdown menu that shows and hides its options when clicked.
 *  */
function setupDropdown() {
    //TODO:
}

// For browser usage
if (typeof window !== 'undefined') {
  document.addEventListener('DOMContentLoaded', setupDropdown);
}
```

Task 11: changeBackgroundColor

```html
<!DOCTYPE html>
<html>
<head>
  <style>
    .my-element {
      width: 200px;
      height: 200px;
      background-color: lightgray;
    }
  </style>
</head>
<body>
  <div id="myDiv" class="my-element"></div>

  <script src="./11.js"></script>
</body>
</html>
```

```javascript
/** \
 *  * Write a JavaScript function that changes the background color of an element when a  mouse enters it.
 *  */
function changeBackgroundColor(elementId, color) {
    //TODO:
}

if (typeof window !== 'undefined') {
    document.addEventListener('DOMContentLoaded', () => {
        changeBackgroundColor('myDiv', 'green');
    });
}
```

Task 12: validateForm

```html
<!DOCTYPE html>
<html>
```

```
<head>
  <style>
    .error-message {
      color: red;
      margin-top: 5px;
    }
  </style>
</head>
<body>
  <form id="myForm">
    <label for="name">Name:</label>
    <input type="text" id="name" required>
    <br>
    <label for="email">Email:</label>
    <input type="email" id="email" required>
    <br>
    <label for="message">Message:</label>
    <textarea id="message" required></textarea>
    <br>
    <button type="submit">Submit</button>
  </form>
  <div id="errorMessages"></div>
  <script src="./12.js">
  </script>
</body>
</html>
```

```
/** \
 *  * Write a JavaScript program that implements a "form" validation that displays an error message if a required field is left empty when submitting the form.
 *  */
function validateForm(){
    //TODO:
}

if (typeof window !== 'undefined') {
```

```
  document.addEventListener('DOMContentLoaded', validateForm);
}
```

Task 13: handleDragStart, handleDragOver, handleDrop

```
<!DOCTYPE html>
<html>
<head>
  <style>
    .drag-list {
      list-style: none;
      padding: 0;
    }

    .drag-item {
      background-color: #CC56FF;
      padding: 10px;
      margin-bottom: 5px;
      cursor: move;
    }
  </style>
</head>
<body>
  <ul id="dragList" class="drag-list">
    <li class="drag-item" draggable="true">Mobile</li>
    <li class="drag-item" draggable="true">Laptop</li>
    <li class="drag-item" draggable="true">Desktop</li>
    <li class="drag-item" draggable="true">Television</li>
    <li class="drag-item" draggable="true">Radio</li>
  </ul>

  <script src="./13.js">
  </script>
</body>
</html>
```

```
/** \
 *  * Write a JavaScript program to implement drag-and-drop functionality to allow users to reorder items in a list.
 *  */


/** \
 *  * Handle the drag start event.
 *  * @param {DragEvent} event - The drag event.
 *  */
function handleDragStart(event) {
    //TODO:
}

/** \
 *  * Handle the drag over event.
 *  * @param {DragEvent} event - The drag event.
 *  */
function handleDragOver(event) {
    //TODO:
}

/** \
 *  * Handle the drag drop event.
 *  * @param {DragEvent} event - The drag event.
 *  */
function handleDrop(event) {
    //TODO:
}

if (typeof window !== 'undefined') {
    document.addEventListener('DOMContentLoaded', () => {
        const dragItems = document.querySelectorAll('.drag-item');
        dragItems.forEach(item => {
            item.addEventListener('dragstart', handleDragStart);
            item.addEventListener('dragover', handleDragOver);
            item.addEventListener('drop', handleDrop);
```

```
    });
  });
}
```

Task 14: copyText

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>w3resource</title>
  </head>
  <body>
    <script src="./14.js"></script>
  </body>
</html>
```

```
/** \
 *  * Write a JavaScript program that copies the content of a text box to the clipboard when a button is clicked.
 *  */
function copyText(){
    //TODO:
}

if (typeof window !== 'undefined') {
    document.addEventListener('DOMContentLoaded', () => {
        // Create input box for text
        const textBox = document.createElement('input');
        textBox.type = 'text'; // Set the input type to text
        textBox.placeholder = 'Type something to copy'; // Add a placeholder
        document.body.appendChild(textBox); // Add the text box to the DOM

        // Create button for copying text
```

```
    const copyButton = document.createElement('button');
    copyButton.textContent = 'Copy Text'; // Set button label
    document.body.appendChild(copyButton); // Add the button to the DOM

    copyButton.addEventListener('click', copyText);
  });
}
```

Task 15: volume_sphere

```
<!-- Declaration of HTML document type -->
<!DOCTYPE html>
<!-- Start of HTML document -->
<html lang="en">
 <head>
  <!-- Declaring character encoding -->
  <meta charset="utf-8" />
  <!-- Setting title of the document -->
  <title>Volume of a Sphere</title>
  <!-- Internal CSS styling -->
  <style>
   /* Styling body to have padding at the top */
   body {
    padding-top: 30px;
   }
   /* Styling label and input elements to display as block */
   label,
   input {
    display: block;
   }
  </style>
 </head>
 <!-- Start of body section -->
 <body>
  <!-- Paragraph explaining the purpose of the form -->
  <p>Input radius value and get the volume of a sphere.</p>
```

```
    <!-- Form for inputting radius and displaying volume -->
    <form action="" method="post" id="MyForm">
      <!-- Label and input field for entering radius -->
      <label for="radius">Radius</label
      ><input type="text" name="radius" id="radius" required />
      <!-- Label and input field for displaying volume -->
      <label for="volume">Volume</label
      ><input type="text" name="volume" id="volume" />
      <!-- Submit button for calculating volume -->
      <input type="submit" value="Calculate" id="submit" />
    </form>
    <script src="./15.js"></script>
    <!-- End of body section -->
  </body>
  <!-- End of HTML document -->
</html>
```

```
/** \
 *  * Write a JavaScript program to calculate sphere volume.
 *  */
function volume_sphere()
 {
  //TODO:
 }
// Attaching volume_sphere function to window.onload and form submit event
window.onload = document.getElementById('MyForm').onsubmit = volume_sphere;
```

Task 16: getSize

```
<!-- Declaration of HTML document type -->
<!DOCTYPE html>
<!-- Start of HTML document -->
<html>
<!-- Start of head section -->
```

```
<head>
<!-- Declaring character encoding -->
<meta charset=utf-8 />
<!-- Setting title of the document -->
<title>Window Size : height and width</title>
<!-- End of head section -->
</head>
<!-- Comment indicating to resize the window and see the result -->
<!-- Resize the window (here output panel) and see the result !-->
<!-- Start of body section with onload and onresize events -->
<body onload="getSize()" onresize="getSize()">
<!-- Division to display height and width size -->
<div id="wh">
    <!-- Place height and width size here! -->
</div>
<script src="16.js"></script>
<!-- End of body section -->
<body>
<!-- End of HTML document -->
</html>
```

```
/** \
 *  * Write a JavaScript program to get the window width and height (any time the window is resized).
 *  */
function getSize()
{
    //TODO:
}

window.onload = getSize; // Attach the function to window.onload event
```

Task 17: toggleSwitch

```
<!DOCTYPE html>
<html>
```

```
<head>
  <style>
    .toggle {
      display: inline-block;
      width: 60px;
      height: 34px;
      position: relative;
      border-radius: 34px;
      background-color: #ccd;
      cursor: pointer;
      transition: background-color 0.3s;
    }

    .toggle::before {
      content: "";
      position: absolute;
      width: 24px;
      height: 24px;
      border-radius: 50%;
      background-color: #fff;
      top: 4px;
      left: 4px;
      transition: transform 0.3s;
    }

    .toggle.on {
      background-color: #66bb6a;
    }

    .toggle.on::before {
      transform: translateX(26px);
    }
  </style>
</head>
<body>
  <div class="toggle"></div>
```

```
 <script src="./17.js">
 </script>
</body>
</html>
```

```
/** \
 *  * Write a JavaScript program to implement a toggle  switch that changes its state when clicked.
 *  */
function toggleSwitch() {
    //TODO:
}

window.onload = toggleSwitch; // Attach the function to window.onload event
```

Task 18: updateProgress

```
<!DOCTYPE html>
<html>
<head>
 <style>
   .progress-bar {
     width: 300px;
     height: 20px;
     background-color: #f0f0f0;
     border-radius: 10px;
     overflow: hidden;
   }

   .progress-bar-fill {
     height: 100%;
     background-color: #4caf50;
     transition: width 0.3s;
   }
 </style>
```

```
</head>
<body>
  <div class="progress-bar">
    <div class="progress-bar-fill" id="progress"></div>
  </div>
  <script src="./18.js">
  </script>
</body>
</html>
```

```
/** \
 *  * Write a JavaScript program to create a progress bar that updates its width based on task completion.
 *  */
function updateProgress(progressPercentage) {
    //TODO:
}

window.onload = () => {
    updateProgress(50);
};
```

Task 19: preventDefaultFormSubmission

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>JS Bin</title>
</head>
<body>
  <form>
   <input type="text" name="username" placeholder="Enter your username" required />
   <input type="email" name="email" placeholder="Enter your email" required />
```

```
  <button type="submit">Submit</button>
 </form>
 <script src="./19.js"></script>
</body>
</html>
```

```
/** \
 *  * Write a JavaScript program that prevents the default behavior of a form submission and logs the input values to the console.
 *  */
function preventDefaultFormSubmission() {
    //TODO:
}

window.onload = preventDefaultFormSubmission;
```

Table 11. Task List for Object-Oriented Program Problems

OOP			
Task ID	Name	Difficulty Level	Number of Test cases
20	Stack	Easy (1768 votes)	4
21	Person	Easy (2480 votes)	2
22	BankAccount	Easy (2244 votes)	6
23	University	Medium (884 votes)	5
24	Product, PersonalCareProduct	Easy (2788 votes)	2
25	Animal, Dog	Medium (2176 votes)	4
26	Employee, Manager	Easy (2652 votes)	4
27	Shape, Circle, Rectangle	Easy (2856 votes)	3

28	Node, DoublyLinkedList	Easy (204 votes)	6
29	Vehicle, Car	Medium (5780 votes)	4

From Table 11, The description for each task is listed below.

Task 20: Stack

```
/**
 * Write a JavaScript program to implement a stack that supports toArray() operation, which converts the stack into an array.
 * @class
 */
class Stack {

  /**
  * Creates an instance of the Stack class.
  * @constructor
  * @param {Array} items - The items to be stored in the stack.
  */
  constructor(items = []) {
      this.items = items;
  }

  /**
  * Pushes an item onto the stack.
  * @param {*} item - The item to be added to the stack.
  */
  push(item) {
      // TODO:
  }

  /**
  * Pops an item from the stack.
  * @returns {*} The item removed from the stack.
  */
  pop() {
```

```
        //TODO:
    }

    /**
    * Peeks at the top item of the stack without removing it.
    * @returns {*} The item on the top of the stack.
    */
    peek() {
        //TODO:
    }

    /**
    * Checks if the stack is empty.
    * @returns {boolean} True if the stack is empty, false otherwise.
    */
    isEmpty() {
        //TODO:
    }

    /**
    * Converts the stack into an array.
    * @returns {Array} The array representation of the stack.
    */
    toArray() {
        //TODO:
    }
}
```

Task 21: Person

```
/**
 * Write a JavaScript program to create a class called "Person" with properties for name, age and country. Include a method to display the person's details. Create two instances of the 'Person' class and display their details.
 * @class
 */
```

```
class Person {

    /**
   * Creates an instance of the Person class.
   * @constructor
   * @param {string} name - The name of the person.
   * @param {number} age - The age of the person.
   * @param {string} country - The country of the person.
   */
    constructor(name, age, country) {
        //TODO:
    }

    /**
   * Displays the details of the person.
   */
    displayDetails() {
        //TODO:
    }
}
```

Task 22: BankAccount

```
/**
 * Write a JavaScript program that creates a class called BankAccount with properties for account number, account holder name, and balance. Include methods to deposit, withdraw, and transfer money between accounts. Create multiple instances of the BankAccount class and perform operations such as depositing, withdrawing, and transferring money.
 * @class
 */
class BankAccount {
    /**
   * Creates an instance of the BankAccount class.
   * @constructor
```

```
 * @param {string} accountNumber - an account number
 * @param {number} balance - an account balance
 */
constructor(accountNumber, balance) {
  //TODO:
}

/**
 * Pushes an item onto the stack.
 * @param {number} amount - The amount to be deposited.
 */
deposit(amount) {
  //TODO:
}

/**
 * Withdraws an amount from the account.
 * @param {number} amount - The amount to be withdrawn.
 */
withdraw(amount) {
  //TODO:
}

  /**
 * Withdraws an amount from the account.
 * @param {number} amount - The amount to be withdrawn.
 * @param {BankAccount} recipientAccount - The account to which the amount will be transferred.
 */
transfer(amount, recipientAccount){
  //TODO:
}

/**
 * Displays the current account balance.
 */
displayBalance() {
  //TODO:
```

```
 }
}
```

Task 23: University

```
/**
 * Write a JavaScript program that creates a class called University with properties for university name and departments. Include methods to add a department, remove a department, and display all departments. Create an instance of the University class and add and remove departments.
 * @class
 */
class University {
        /**
   * Creates an instance of the University class.
   * @constructor
   * @param {string} name - The name of the university.
   */
  constructor(name) {
    //TODO:
  }

  /**
   * Adds a department to the university.
   * @param {string} department - The name of the department to add.
   */
  addDepartment(department) {
    //TODO:
  }

  /**
   * Removes a department from the university.
   * @param {string} department - The name of the department to remove.
   */
  removeDepartment(department) {
    //TODO:
  }
```

```
  /**
   * Displays the current departments in the university.
   */
  displayDepartments() {
    //TODO:
  }
}
```

Task 24: Product, PersonalCareProduct

```
/**
 * Write a JavaScript program that creates a class called Product with properties for product ID, name, and price. Include a method to calculate the total price by multiplying the price by the quantity. Create a subclass called PersonalCareProduct that inherits from the Product class and adds an additional property for the warranty period. Override the total price calculation method to include the warranty period. Create an instance of the PersonalCareProduct class and calculate its total price.
 * @class
 */
class Product {
  /**
   * Creates an instance of the Product class.
   * @constructor
   * @param {number} id - The product ID.
   * @param {string} name - The name of the product.
   * @param {number} price - The price of the product.
   */
  constructor(id, name, price) {
    //TODO:
  }

  /**
   * Calculates the total price for a given quantity of the product.
   * @param {number} quantity - The quantity of the product.
   * @returns {number} - The total price for the given quantity.
   */
```

```
  calculateTotalPrice(quantity) {
    //TODO:
  }
}

/**
 * PersonalCareProduct Class
 * @class
 */
class PersonalCareProduct extends Product {

    /**
     * Creates an instance of the PersonalCareProduct class.
     * @param {number} id - The product ID.
     * @param {string} name - The name of the product.
     * @param {number} price - The price of the product.
     * @param {number} warrantyPeriod - The warranty period for the product.
     */
  constructor(id, name, price, warrantyPeriod) {
    //TODO:
  }

  /**
   * Calculates the total price for a given quantity of the product.
   * @param {number} quantity - The quantity of the product.
   * @returns {number} - The total price for the given quantity.
   */
  calculateTotalPrice(quantity) {
    //TODO:
  }
}
```

Task 25: Animal, Dog

```
/**
```

```
 * Write a JavaScript program that creates a class called 'Animal' with properties for species and sound. Include a method to make the animal's sound. Create a subclass called 'Dog' that inherits from the 'Animal' class and adds an additional property for color. Override the make sound method to include the dog's color. Create an instance of the 'Dog' class and make it make its sound.
 * @class
 */
class Animal {
    /**
     * Creates an instance of the Animal class.
     * @constructor
     * @param {string} species - The species of the animal.
     * @param {string} sound - The sound the animal makes.
     */
  constructor(species, sound) {
    //TODO:
  }

  /**
   * Makes the sound of the animal.
   */
  makeSound() {
    //TODO:
  }
}

/**
 * Dog Class
 * @class
 */
class Dog extends Animal {
    /**
     * Creates an instance of the Dog class.
     * @constructor
     * @param {string} species - The species of the animal.
     * @param {string} sound - The sound the animal makes.
     * @param {string} color - The color of the dog.
     */
```

```
  constructor(species, sound, color) {
    //TODO:
  }

  /**
   * Makes the sound of the dog and includes its color.
   */
  makeSound() {
    //TODO:
  }
}
```

Task 26: Employee, Manager

```
/**
 * Write a JavaScript program that creates a class called 'Employee' with properties for name and salary. Include a method to calculate annual salary. Create a subclass called 'Manager' that inherits from the 'Employee' class and adds an additional property for department. Override the annual salary calculation method to include bonuses for managers. Create two instances of the 'Manager' class and calculate their annual salary.
 * @class
 */
class Employee {
    /**
     * Creates an instance of the Employee class.
     * @constructor
     * @param {*} name
     * @param {*} salary
     */
  constructor(name, salary) {
    //TODO:
  }

  /**
   * Calculates the annual salary of the employee.
   * @returns {number} The annual salary.
   */
```

```
 calculateAnnualSalary() {
    //TODO:
  }
}

/**
 * Manager Class
 * @class
 */
class Manager extends Employee {
    /**
     * Creates an instance of the Manager class.
     * @constructor
     * @param {*} name
     * @param {*} salary
     * @param {*} department
     */
  constructor(name, salary, department) {
    //TODO:
  }

  /**
   * Calculates the annual salary of the manager, including bonuses.
   * @returns {number} The annual salary with bonuses.
   */
  calculateAnnualSalary() {
    //TODO:
  }
}
```

Task 27: Shape, Circle, Rectangle

```
/**
 * Write a JavaScript program that creates a class called 'Shape' with a method to calculate the area. Create two subclasses, 'Circle' and 'Rectangle', that inherit from the 'Shape' class and override the area calculation method. Create an instance of the 'Circle' class and calculate its area. Similarly, do the same for the 'Rectangle' class.
```

```
 * @class
 */
class Shape {
  calculateArea() {
    throw new Error("Method 'calculateArea()' must be overridden in subclasses");
  }
}

/**
 * Circle Class
 * @class
 */
class Circle extends Shape {
    /**
     * Creates an instance of the Circle class.
     * @constructor
     * @param {number} radius
     */
  constructor(radius) {
    //TODO:
  }

  /**
   * Calculates the area of the circle.
   * @returns {number} The area of the circle.
   */
  calculateArea() {
    //TODO:
  }
}

/**
 * Rectangle Class
 * @class
 */
class Rectangle extends Shape {
    /**
```

```
    * Creates an instance of the Rectangle class.
    * @constructor
    * @param {number} width
    * @param {number} height
    */
  constructor(width, height) {
    //TODO:
  }

  /**
   * Calculates the area of the rectangle.
   * @returns {number} The area of the rectangle.
   */
  calculateArea() {
    return this.width * this.height;
  }
}
```

Task 28: Node, DoublyLinkedList

```
/**
 * Write a JavaScript program to create and display Doubly Linked Lists.
 * @class
 */
class Node {
    /**
     * Creates an instance of the Node class.
     * @constructor
     * @param {*} value
     */
    constructor(value) {
        //TODO:
        this.next = null; // Initialize the pointer to the next node as null
        this.previous = null; // Initialize the pointer to the previous node as null
    }
```

```
}
/**
 * Doubly Linked List Class
 * @class
 */
class DoublyLinkedList {
    /**
     * Creates an instance of the DoublyLinkedList class.
     * @constructor
     * @param {*} value
     */
    constructor(value) {
        // Initialize the head node with the given value and no next or previous node
        this.head = {
            value: value, // Store the value of the head node
            next: null, // Pointer to the next node in the list, initially set to null
            previous: null // Pointer to the previous node in the list, initially set to null
        };
        this.length = 0; // Initialize the length of the list to 0
        this.tail = this.head; // Set the tail node to the head node initially
    }


    /**
     * Adds a new node to the end of the list.
     * @param {*} newNode
     */
    add(newNode) {
        this.tail.next = newNode; // Set the next pointer of the current tail node to the new node
        newNode.previous = this.tail; // Set the previous pointer of the new node to the current tail node
        this.tail = newNode; // Update the tail node to the new node
        this.length++; // Increment the length of the list
    }

    /**
     * Prints the values of the nodes in the list.
     */
```

```
    printList() {
        let current = this.head; // Start from the head of the list
        let result = []; // Array to store the values of the nodes
        while (current !== null) { // Iterate through the list until reaching the end
            result.push(current.value); // Push the value of the current node to the array
            current = current.next; // Move to the next node
        }
        console.log(result.join(' ')); // Log the values of the nodes separated by space
        return this; // Return the DoublyLinkedList object for chaining
    }
}
```

Task 29: Vehicle, Car

```
/**
 * Write a JavaScript program that creates a class called 'Vehicle' with properties for make, model, and year. Include a method to display vehicle details. Create a subclass called 'Car' that inherits from the 'Vehicle' class and includes an additional property for the number of doors. Override the display method to include the number of doors.
 * @class
 */
class Vehicle {
    /**
     * Creates an instance of the Vehicle class.
     * @constructor
     * @param {*} make
     * @param {*} model
     * @param {*} year
     */
  constructor(make, model, year) {
    //TODO:
  }

  /**
   * Displays the details of the vehicle.
   */
  displayDetails() {
```

```
    //TODO:
  }
}

/**
 * Car class that extends the Vehicle class.
 */
class Car extends Vehicle {
    /**
     * Creates an instance of the Car class.
     * @constructor
     * @param {*} make
     * @param {*} model
     * @param {*} year
     * @param {*} doors
     */
  constructor(make, model, year, doors) {
    //TODO:
  }

  /**
   * Displays the details of the car.
   */
  displayDetails() {
//TODO:
  }
}
```

Appendix B

Maintainability Rules (170 rules)

rule S6441: Unused methods of React components should be removed
rule S6331: Regular expressions should not contain empty groups
rule S2094: Classes should not be empty
rule S4030: Collection contents should be used
rule S888: Equality operators should not be used in "for" loop termination conditions
rule S4140: Sparse arrays should not be created with extra commas
rule S6325: Regular expression literals should be used when possible

rule S4144: Functions should not have identical implementations
rule S6326: Regular expressions should not contain multiple spaces
rule S5264: "<object>" tags should provide an alternative content
rule S6353: Regular expression quantifiers and character classes should be used concisely
rule S6594: "RegExp.exec()" should be preferred over "String.match()"
rule S6478: React components should not be nested
rule S6479: JSX list components should not use array indexes as key
rule S6477: JSX list components should have a key property
rule S4165: Assignments should not be redundant
rule S6582: Optional chaining should be preferred
rule S4043: Array-mutating methods should not be used misleadingly
rule S5257: HTML "<table>" should not be used for layout purposes
rule S6650: Renaming import, export, and destructuring assignments should not be to the same name
rule S6772: Spacing between inline elements should be explicit
rule S6770: User-defined JSX components should use Pascal case
rule S6654: __proto__ property should not be used
rule S101: Class names should comply with a naming convention
rule S6775: All "defaultProps" should have non-required PropTypes
rule S6774: React components should validate prop types
rule S6653: Use Object.hasOwn static method instead of hasOwnProperty
rule S6535: Unnecessary character escapes should be removed
rule S6657: Octal escape sequences should not be used
rule S6761: "children" and "dangerouslySetInnerHTML" should not be used together
rule S6522: Import variables should not be reassigned
rule S6643: Prototypes of builtin objects should not be modified
rule S6644: Ternary operator should not be used instead of simpler alternatives
rule S6763: "shouldComponentUpdate" should not be defined when extending "React.PureComponent"
rule S1199: Nested code blocks should not be used
rule S6647: Unnecessary constructors should be removed
rule S6645: Variables should not be initialized to undefined
rule S6767: Unused React typed props should be removed
rule S6766: JSX special characters should be escaped
rule S6790: String references should not be used
rule S6551: Objects and classes converted or coerced to strings should define a "toString()" method
rule S6793: ARIA properties in DOM elements should have valid values
rule S6791: React legacy lifecycle methods should not be used
rule S6676: Calls to ".call()" and ".apply()" methods should not be redundant
rule S6557: Ends of strings should be checked with "startsWith()" and "endsWith()"
rule S4138: "for of" should be used with Iterables
rule S108: Nested blocks of code should not be left empty
rule S107: Functions should not have too many parameters
rule S6661: Object spread syntax should be used instead of "Object.assign"
rule S6660: If statements should not be the only statement in else blocks

rule S2187: Test files should contain at least one test case
rule S2301: Methods should not contain selector parameters
rule S6666: Spread syntax should be used instead of "apply()"
rule S4123: "await" should only be used with promises
rule S6789: React's "isMounted" should not be used
rule S6788: React's "findDOMNode" should not be used
rule S2814: Variables and functions should not be redeclared
rule S1607: Tests should not be skipped without providing a reason
rule S128: Switch cases should end with an unconditional "break" statement
rule S125: Sections of code should not be commented out
rule S6079: Tests should not execute any code after "done()" is called
rule S3800: Functions should always return the same type
rule S1854: Unused assignments should be removed
rule S2703: Variables should be declared explicitly
rule S6019: Reluctant quantifiers in regular expressions should be followed by an expression that can't match the empty string
rule S6481: React Context Provider values should have stable identities
rule S6486: JSX list components keys should match up between renders
rule S6397: Character classes in regular expressions should not contain only one character
rule S6035: Single-character alternations in regular expressions should be replaced with character classes
rule S1940: Boolean checks should not be inverted
rule S3504: Variables should be declared with "let" or "const"
rule S3626: Jump statements should not be redundant
rule S3863: Imports from the same module should be merged
rule S1314: Octal values should not be used
rule S3735: "void" should not be used
rule S1439: Only "while", "do", "for" and "switch" statements should be labelled
rule S3972: Conditionals should start on new lines
rule S4619: "in" should not be used on arrays
rule S2430: Constructor names should start with an upper case letter
rule S3516: Function returns should not be invariant
rule S1219: "switch" statements should not contain non-case labels
rule S6092: Chai assertions should have only one reason to succeed
rule S7060: Module should not import itself
rule S1527: Future reserved words should not be used as identifiers
rule S2737: "catch" clauses should do more than rethrow
rule S2970: Assertions should be complete
rule S1515: Functions should not be defined inside loops
rule S1516: Multiline string literals should not be used
rule S7059: Constructors should not contain asynchronous operations
rule S1871: Two branches in a conditional structure should not have exactly the same implementation
rule S1994: "for" loop increment clauses should modify the loops' counters
rule S1874: Deprecated APIs should not be used
rule S1788: Function parameters with default values should be last

rule S2990: The global "this" object should not be used
rule S2870: "delete" should not be used on arrays
rule S1301: "if" statements should be preferred over "switch" when simpler
rule S1533: Wrapper objects should not be used for primitive types
rule S6859: Imports should not use absolute paths
rule S6852: Elements with an interactive role should support focus
rule S6851: Images should have a non-redundant alternate description
rule S6850: Heading elements should have accessible content
rule S6848: Non-interactive DOM elements should not have an interactive handler
rule S6847: Non-interactive elements shouldn't have event handlers
rule S6842: Non-interactive DOM elements should not have interactive ARIA roles
rule S6841: "tabIndex" values should be 0 or -1
rule S2486: Exceptions should not be ignored
rule S3696: Literals should not be thrown
rule S6840: DOM elements should use the "autocomplete" attribute correctly
rule S2004: Functions should not be nested too deeply
rule S6846: DOM elements should not use the "accesskey" property
rule S6845: Non-interactive DOM elements should not have the `tabIndex` property
rule S3579: Array indexes should be numeric
rule S6844: Anchor tags should not be used as buttons
rule S6843: Interactive DOM elements should not have non-interactive ARIA roles
rule S2392: Variables should be used in the blocks where they are declared
rule S6750: The return value of "ReactDOM.render" should not be used
rule S1186: Functions should not be empty
rule S6754: The return value of "useState" should be destructured and named symmetrically
rule S1068: Unused private class members should be removed
rule S6757: "this" should not be used in functional components
rule S6637: Unnecessary calls to ".bind()" should not be used
rule S6635: Constructors should not return values
rule S6756: "setState" should use a callback when referencing the previous state
rule S6509: Extra boolean casts should be removed
rule S6627: Users should not use internal APIs
rule S6748: React "children" should not be passed as prop
rule S6749: Redundant React fragments should be removed
rule S6861: Mutable variables should not be exported
rule S6746: In React "this.state" should not be mutated directly
rule S6747: JSX elements should not use unknown properties and attributes
rule S3358: Ternary operators should not be nested
rule S1481: Unused local variables and functions should be removed
rule S1128: Unnecessary imports should be removed
rule S6819: Prefer tag over ARIA role
rule S3782: Arguments to built-in functions should match documented types
rule S2692: "indexOf" checks should not be for positive numbers
rule S1121: Assignments should not be made from within sub-expressions
rule S4634: Shorthand promises should be used
rule S1125: Boolean literals should not be used in comparisons

rule S2699: Tests should include assertions
rule S6811: DOM elements with ARIA role should only have supported properties
rule S1126: Return of boolean expressions should not be wrapped into an "if-then-else" statement
rule S5843: Regular expressions should not be too complicated
rule S1479: "switch" statements should not have too many "case" clauses
rule S5958: Tests should check which exception is thrown
rule S3415: Assertion arguments should be passed in the correct order
rule S1119: Labels should not be used
rule S6807: DOM elements with ARIA roles should have the required properties
rule S1472: Function call arguments should not start on new lines
rule S2681: Multiline blocks should be enclosed in curly braces
rule S2685: "arguments.caller" and "arguments.callee" should not be used
rule S4624: Template literals should not be nested
rule S3776: Cognitive Complexity of functions should not be too high
rule S878: Comma operator should not be used
rule S5869: Character classes in regular expressions should not contain the same character twice
rule S6836: "case" and "default" clauses should not contain lexical declarations
rule S6957: Deprecated React APIs should not be used
rule S1264: A "while" loop should be used instead of a "for" loop
rule S5860: Names of regular expressions named groups should be used
rule S2234: Parameters should be passed in the correct order
rule S3686: Functions should be called consistently with or without "new"
rule S6827: Anchors should contain accessible content
rule S5973: Tests should be stable
rule S1134: Track uses of "FIXME" tags
rule S4524: "default" clauses should be last
rule S6824: No ARIA role or property for unsupported DOM elements
rule S1135: Track uses of "TODO" tags
rule S2589: Boolean expressions should not be gratuitous
rule S6822: No redundant ARIA role
rule S6821: DOM elements with ARIA roles should have a valid non-abstract role

Security Rules (13 rules)
Rule S5876: A new session should be created during user authentication
Rule S6321: Administration services access should be restricted to specific IP addresses
Rule S6317: AWS IAM policies should limit the scope of permissions given
Rule S5547: Cipher algorithms should be robust
Rule S4426: Cryptographic keys should be robust
Rule S5542: Encryption algorithms should be used with secure mode and padding scheme
Rule S2598: File uploads should be restricted
Rule S5659: JWT should be signed and verified with strong cipher algorithms
Rule S2819: Origins should be verified during cross-origin communications
Rule S4830: Server certificates should be verified during SSL/TLS connections
Rule S5527: Server hostnames should be verified during SSL/TLS connections
Rule S4423: Weak SSL/TLS protocols should not be used

Rule S2755: XML parsers should not be vulnerable to XXE attacks

Appendix C

Table 12. Kruskal Wallis Results

	H	p	epsilon_sq	k	n
claude-4-opus	5.13	0.08	0.12	3	30
gemini-2.5-pro	3.43	0.18	0.06	3	28
gpt-4.1	8.37	0.02	0.24	3	30

Table 13. Claude 4 Opus Post-hoc pairwise Results

group_a	group_b	U	p_raw	n_a	n_b	p_holm
dsal	oop	78.50	0.02	10	10	0.07
dsal	web_dev	68.50	0.11	10	10	0.22
oop	web_dev	46	0.79	10	10	0.79

Table 14. GPT 4.1 Post-hoc pairwise Results

group_a	group_b	U	p_raw	n_a	n_b	p_holm
dsal	oop	87	0.00	10	10	0.01
dsal	web_dev	74	0.04	10	10	0.09
oop	web_dev	43	0.62	10	10	0.62

Table 15. Gemini 2.5 Pro Post-hoc pairwise Results

group_a	group_b	U	p_raw	n_a	n_b	p_holm
dsal	oop	55.50	0.11	8	10	0.30
dsal	web_dev	56	0.10	8	10	0.30
oop	web_dev	55.50	0.69	10	10	0.69

Reference

1. Mucci, T., *What is a code generator?* 2024, IBM.
2. Mohsin, A., et al., *Can we trust large language models generated code? a framework for in-context learning, security patterns, and code evaluations across diverse llms.* arXiv preprint arXiv:2406.12513, 2024.

3. Vaithilingam, P., T. Zhang, and E.L. Glassman. *Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models*. in *Chi conference on human factors in computing systems extended abstracts*. 2022.
4. Madaan, A., et al., *Self-refine: Iterative refinement with self-feedback.* Advances in Neural Information Processing Systems, 2023. **36**: p. 46534-46594.
5. Liu, R., et al., *On Iterative Evaluation and Enhancement of Code Quality Using GPT-4o.* arXiv preprint arXiv:2502.07399, 2025.
6. Pearce, H., et al., *Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions.* Communications of the ACM, 2025. **68**(2): p. 96-105.
7. Jiang, J., et al., *A survey on large language models for code generation.* arXiv preprint arXiv:2406.00515, 2024.
8. Barbosa, E.A., et al., *Enforcing exception handling policies with a domain-specific language.* IEEE Transactions on Software Engineering, 2015. **42**(6): p. 559-584.
9. Ye, S., et al., *Prompt Alchemy: Automatic Prompt Refinement for Enhancing Code Generation.* arXiv preprint arXiv:2503.11085, 2025.
10. Bi, Z., et al., *Iterative refinement of project-level code context for precise code generation with compiler feedback.* arXiv preprint arXiv:2403.16792, 2024.
11. Chen, Y., et al., *Iterative prompt refinement for mining gene relationships from ChatGPT.* bioRxiv, 2023.
12. Khojah, R., et al., *The Impact of Prompt Programming on Function-Level Code Generation.* IEEE Transactions on Software Engineering, 2025.
13. Dong, Y., et al., *Codescore: Evaluating code generation by learning code execution.* ACM Transactions on Software Engineering and Methodology, 2025. **34**(3): p. 1-22.
14. Straubinger, P., et al. *Mutation Testing via Iterative Large Language Model-Driven Scientific Debugging*. in *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2025. IEEE.
15. Nunez, A., et al., *Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing.* arXiv preprint arXiv:2409.10737, 2024.
16. Wei, J., et al., *Finetuned language models are zero-shot learners.* arXiv preprint arXiv:2109.01652, 2021.
17. Kojima, T., et al., *Large language models are zero-shot reasoners.* Advances in neural information processing systems, 2022. **35**: p. 22199-22213.
18. Wang, X., et al., *Self-consistency improves chain of thought reasoning in language models.* arXiv preprint arXiv:2203.11171, 2022.
19. Satyapriya Krishna, Chirag Agarwal, and H. Lakkaraju, *Understanding the Effects of Iterative Prompting on Truthfulness.* arXiv:2402.06625v1, 2024.
20. Boukouchi, Y., et al., *Comparative study of software quality models.* IJCSI International Journal of Computer Science Issues, 2013: p. 309-314.
21. Standardization, I.O.f., ***Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Quality model overview and usage*** 2024, ISO.
22. Zheng, J., et al., *Beyond Correctness: Benchmarking Multi-dimensional Code Generation for Large Language Models.* arXiv preprint arXiv:2407.11470, 2024.
23. Tosi, D., *Studying the quality of source code generated by different ai generative engines: An empirical evaluation.* Future Internet, 2024. **16**(6): p. 188.
24. Coignion, T., C. Quinton, and R. Rouvoy. *A performance study of llm-generated code on leetcode*. in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 2024.
25. Peng, J., et al., *CWEval: Outcome-driven Evaluation on Functionality and Security of LLM Code Generation.* arXiv preprint arXiv:2501.08200, 2025.
26. Wang, J., et al., *Enhancing Large Language Models for Secure Code Generation: A Dataset-driven Study on Vulnerability Mitigation.* arXiv preprint arXiv:2310.16263, 2023.

27. Kharma, M., et al., *Security and Quality in LLM-Generated Code: A Multi-Language, Multi-Model Analysis.* arXiv preprint arXiv:2502.01853, 2025.
28. Dillmann, M., J. Siebert, and A. Trendowicz, *Evaluation of large language models for assessing code maintainability.* arXiv preprint arXiv:2401.12714, 2024.
29. Developers, P. *PMD: Source Code Analyzer*. 2024; Available from: https://pmd.github.io/.
30. SonarSource. *SonarQube: Continuous Inspection Tool*. 2024; Available from: https://www.sonarsource.com/products/sonarqube/.
31. Wadhwa, N., et al., *Core: Resolving code quality issues using llms.* Proceedings of the ACM on Software Engineering, 2024. **1**(FSE): p. 789-811.
32. Liu, Y., et al., *Refining chatgpt-generated code: Characterizing and mitigating code quality issues.* ACM Transactions on Software Engineering and Methodology, 2024. **33**(5): p. 1-26.
33. Blyth, S., et al., *Static Analysis as a Feedback Loop: Enhancing LLM-Generated Code Beyond Correctness.* arXiv preprint arXiv:2508.14419, 2025.
34. GeeksForGeeks. *Top 10 Programming Languages For 2025*. 2025; Available from: https://www.geeksforgeeks.org/blogs/top-programming-languages-of-the-future/.
35. Hou, W. and Z. Ji, *Comparing large language models and human programmers for generating programming code.* Advanced Science, 2025. **12**(8): p. 2412279.
36. Sobo, A., et al., *Evaluating LLMs for code generation in HRI: A comparative study of ChatGPT, gemini, and claude.* Applied Artificial Intelligence, 2025. **39**(1): p. 2439610.
37. ApX. *Top AI Models: Best LLMs for Coding*. 2025; Available from: https://apxml.com/leaderboards/coding-llms.
38. Router, O. *Open Router Ranking*. 2025; Available from: https://openrouter.ai/rankings.
39. Team, O. *Introducing GPT-4.1 in the API*. 2025; Available from: https://openai.com/index/gpt-4-1/.
40. Team, C. *Claude Opus 4*. 2025; Available from: https://www.anthropic.com/claude/opus.
41. Team, G., *Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities.* 2025.
42. Park, J., et al., *JEST: N+1-Version Differential Testing of Both JavaScript Engines and Specification*, in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021.
43. Della Porta, A., S. Lambiase, and F. Palomba, *Do Prompt Patterns Affect Code Quality? A First Empirical Assessment of ChatGPT-Generated Code.* arXiv preprint arXiv:2504.13656, 2025.
44. Li, K., et al., *Measuring and controlling instruction (in) stability in language model dialogs, 2024.* URL https://arxiv. org/abs/2402.10962.
45. Zhang, Y., et al., *The law of knowledge overshadowing: Towards understanding, predicting, and preventing llm hallucination.* arXiv preprint arXiv:2502.16143, 2025.
46. Zhuo, J., et al., *ProSA: Assessing and understanding the prompt sensitivity of LLMs.* arXiv preprint arXiv:2410.12405, 2024.
47. Jimenez, C.E., et al., *Swe-bench: Can language models resolve real-world github issues?, 2024.* URL https://arxiv. org/abs/2310.06770, 2023. **7**.
48. Liu, T., C. Xu, and J. McAuley, *Repobench: Benchmarking repository-level code auto-completion systems.* arXiv preprint arXiv:2306.03091, 2023.
49. Chen, M., et al., *Evaluating large language models trained on code.* arXiv preprint arXiv:2107.03374, 2021.