Towards LLM-Driven Code Generation: The Impact of Process Models and Non-Functional Requirements on Software Development

Feng Lin

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Science (Master of Applied Science Software Engineering) at

Concordia University

Montréal, Québec, Canada

May, 2025

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Feng Lin**

Entitled: **Towards LLM-Driven Code Generation: The Impact of Process Models and Non-Functional Requirements on Software Development**

and submitted in partial fulfillment of the requirements for the degree of

Master of Science (Master of Applied Science Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Yang Wang

_____ Examiner
Dr. Emad Shihab

_____ Examiner
Dr. Yang Wang

_____ Supervisor
Dr. Tse-Hsun (Peter) Chen

Approved by _____
Joey Paquet, Chair
Department of Computer Science and Software Engineering

_____ 2025

_____
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Towards LLM-Driven Code Generation: The Impact of Process Models and Non-Functional Requirements on Software Development

Feng Lin

Research on LLM-based code generation is growing but often overlooks the impact of Software Engineering (SE) knowledge, such as software process models and non-functional requirements (NFRs). This thesis explores how integrating SE practices with LLMs can help bridge this gap.

We first propose *FlowGen* to explore the impact of software process models on code generation. We assign LLM agents to different development roles and study three models: $FlowGen_{Waterfall}$, $FlowGen_{TDD}$, and $FlowGen_{Scrum}$. We evaluate *FlowGen* using GPT-3.5, comparing it against several baselines on four code benchmarks (i.e., HumanEval, HumanEval-ET, MBPP, and MBPP-ET). Our results show that $FlowGen_{Scrum}$ outperforms the other process models, achieving a 15% improvement in Pass@1 over *RawGPT* on average. Integrating a state-of-the-art technique (i.e., *CodeT*) further boosts Pass@1 scores. Our findings also show that development activities impact code quality differently, and *FlowGen* enhances result stability across LLM versions and temperature settings.

Secondly, we investigate how variations in developer behavior affect how LLMs address NFRs (e.g., expressing the same NFR using different wording). Robust LLMs should generate consistent code despite such variations. We propose *RoboNFR* to evaluate LLM robustness in NFR-aware code generation across four key dimensions—code design, performance, readability, and reliability—using three methodologies: prompt variation, regression testing, and diverse workflows. Our experiments show that *RoboNFR* effectively reveals robustness issues in tested LLMs. Overall, across the three methodologies, incorporating NFRs tends to reduce Pass@1 scores while improving NFR-specific metrics, but also increases the standard deviation in both correctness and quality.

This thesis highlights the significant influence of software process models and NFRs, emphasizing the need for future work to incorporate such SE knowledge in the era of LLMs.

Acknowledgments

I would like to express my sincere gratitude towards my supervisor Dr. Tse-Hsun (Peter) Chen for his invaluable guidance and continuous support. His brilliance taught me how to think and approach problems in a brand new way. I am highly appreciative of this incredible opportunity.

Additionally, I am beyond grateful to my parents for their unconditional love and unrelenting support towards my education. Their hard work and dedication inspire me everyday and I would not be where I am today without their sacrifices.

Finally, I would like to thank my friends, especially those in the Software PErformance, Analysis, and Reliability (SPEAR) Lab at Concordia University. Their encouragement and support have guided me through every stage of my graduate studies. Their kindness and care have been a constant source of strength and motivation.

Thank you all for your immense contributions to this thesis.

Related Publications

My contributions to the following chapters and related publications of the thesis include: researching and drafting initial ideas; studying background knowledge and related work; designing and implementing the proposed frameworks; conducting experiments and analyzing results; and writing and refining the manuscript.

My co-authors supported me by helping to refine the initial ideas, identifying missing related work, providing feedback on early drafts, and assisting in polishing the writing.

Earlier versions of the work in the thesis were published as listed below:

- **Feng, Lin**, Dong Jae Kim, Tse-Husn (Peter)Chen. "SOEN-101: Code Generation by Emulating Software Process Models Using Large Language Model Agents" (Chapter 2). *arXiv preprint* arXiv:2403.15852 (2024). Accepted to the 47th International Conference on Software Engineering (ICSE 2025), Research Track, to appear.

- **Feng Lin**, Dong Jae Kim, Zhenhao Li, Jinqiu Yang, Tse-Hsun (Peter)Chen. "RobuNFR: Evaluating the Robustness of Large Language Models on Non-Functional Requirements Aware Code Generation" (Chapter 3). *arXiv preprint* arXiv:2503.22851 (2025).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Recent research has extensively explored how Large Language Models (LLMs) can enhance code generation, particularly in addressing complex coding challenges and improving code correctness (Fan et al., 2023; J. Liu et al., 2024; Q. Zhang et al., 2024; Z. Zhang et al., 2024; Zheng et al., 2024). Some LLM-based services—such as ChatGPT (OpenAI, 2023), GitHub Copilot (Copilot, 2024a), and Cursor (Cursor, 2024)—have become increasingly integrated into modern development workflows. These tools are now widely adopted to assist developers in software development, particularly code generation tasks.

Current research has focused on enhancing the code generation capabilities of LLMs, particularly in terms of both functional and non-functional requirements (NFRs). For example, some studies rely on prompt-based techniques: Huang, Bu, Qing, and Cui (2024) introduce Chain-of-Thought prompting to improve code correctness through step-by-step reasoning, while Waghjale, Veerendranath, Wang, and Fried (2024) explore how prompt engineering can ensure correctness while also boosting code execution speed. Other studies adopt agent-based approaches. For instance, Dong, Jiang, Jin, and Li (2023) propose a multi-agent system involving communication between a developer, tester, and analyst to improve code correctness, and Nunez, Islam, Jha, and Najafirad (2024) employ agents to enhance code security. Overall, these works primarily focus on prompt-based or agent-based techniques to improve the accuracy and quality of generated code.

Although these papers provide valuable insights, most existing studies emphasize syntactic correctness or benchmark scores, while rarely exploring the integration of LLMs into structured software engineering practices. For example, real-world development teams often follow software process models such as Waterfall or Scrum. The impact of deploying LLMs within such teams and their development activities remains largely unexplored. Similarly, real-world teams care not only about code correctness but also may use LLMs to address NFRs. Practical aspects such as prompt selection and the effectiveness of LLMs across different NFR dimensions are still underexplored and deserve further study.

Some studies have attempted to incorporate software engineering knowledge into code generation research. For example, Hong et al. (2024) introduce MetaGPT to explore how LLMs can be integrated into the Waterfall model. However, the reason behind choosing Waterfall as the workflow remains unclear, and whether better-suited workflows exist is still an open question. Furthermore, their study focuses on how the Waterfall workflow affects code correctness, while largely ignoring code quality metrics. Similarly, important details related to LLM usage—such as temperature settings, model version selection, and prompt design—are also not discussed.

Hence, motivated by prior studies and the growing importance of applying LLMs in real-world software development scenarios, this thesis presents a set of investigations aimed at providing insights for both LLM users and developers. We conduct our study from two key perspectives: the integration of LLMs into **Software Process Models** and the **robustness of LLMs in NFR-aware code generation**. Based on these perspectives, we propose a series of approaches to investigate how these factors influence LLM-generated code and to offer practical insights for future applications.

1.2 Thesis Overview and Contributions

The main research in this thesis is organized into two chapters, and this section offers an overview of both, highlighting their contributions and providing a brief summary of each chapter.

Chapter 2: SOEN-101: Code Generation by Emulating Software Process Models Using Large Language Model Agents

In this chapter, we investigate how LLM code generation capabilities vary when deployed within different software process models. We introduce *FlowGen*, a framework designed to evaluate models such as $FlowGen_{Waterfall}$, $FlowGen_{TDD}$, and $FlowGen_{Scrum}$ by assigning LLM agents to distinct software development roles. We compare the code generated across these workflows and analyze how specific development activities influence the outcomes. Furthermore, we examine how variations in temperature settings and model versions affect the results of these workflows. In addition, we incorporate some State-Of-The-Art (SOTA) techniques with the studied workflows to demonstrate how they can be combined to achieve better results.

Overall, *FlowGen* provides a practical framework for future studies to easily explore how workflow configurations impact LLM-generated code. Additionally, our findings offer guidance such as workflow selection, development activity prioritization, and temperature settings that should be considered when integrating LLMs into different development scenarios.

Chapter 3: Robustness of Large Language Models on Non-Functional Requirements Aware Code Generation

In this chapter, we examine the robustness of LLMs in NFR-aware code generation, meaning we evaluate how the generated code changes when users interact with LLMs in different ways to address NFRs. To support this study, we present *RoboNFR*, which examines four NFR dimensions—code design, readability, reliability, and performance—using three evaluation methodologies: prompt variation, regression testing, and diverse workflows. Our findings reveal that different LLMs exhibit varying degrees of change in their code generation capabilities under these conditions, highlighting the robustness challenges of LLMs in NFR-aware code generation.

Overall, we show that existing LLMs face robustness issues in NFR-aware code generation. *RoboNFR* provides a practical framework for monitoring the robustness of deployed LLMs, offering continuous quality assurance for LLM-based products. It offers insights for LLM users and developers to select suitable prompts, apply robust model updates, and design workflows to guard

against unexpected output changes in their LLM-based products.

1.3 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 presents our study on software process models, while Chapter 3 explores NFR-aware code generation using LLMs. Each section includes subsections on methodology, evaluation, discussion, and related topics. In Chapter 4, we summarize our findings, discuss key insights, and outline potential directions for future work.

Chapter 2

SOEN-101: Code Generation by Emulating Software Process Models Using Large Language Model Agents

In this chapter, we explore a key aspect of software engineering knowledge: what happens when LLMs are integrated into real-world software process models.

2.1 Abstract

Software process models are essential to facilitate collaboration and communication among software teams to solve complex development tasks. Inspired by these software engineering practices, we present *FlowGen*– a code generation framework that emulates software process models based on multiple Large Language Model (LLM) agents. We emulate three process models, $FlowGen_{Waterfall}$, $FlowGen_{TDD}$, and $FlowGen_{Scrum}$, by assigning LLM agents to embody roles (i.e., requirement engineer, architect, developer, tester, and scrum master) that correspond to everyday development activities and organize their communication patterns. The agents work collaboratively using chain-of-thought and prompt composition with continuous self-refinement to improve the code quality. We use GPT3.5 as our underlying LLM and several baselines (*RawGPT*, *CodeT*, *Reflexion*) to evaluate code generation on four benchmarks: HumanEval, HumanEval-ET, MBPP, and MBPP-ET. Our

findings show that *FlowGen*$_{Scrum}$ excels compared to other process models, achieving a Pass@1 of 75.2, 65.5, 82.5, and 56.7 in HumanEval, HumanEval-ET, MBPP, and MBPP-ET, respectively (an average of 15% improvement over *RawGPT*). Compared with other state-of-the-art techniques, *FlowGen*$_{Scrum}$ achieves a higher Pass@1 in MBPP compared to *CodeT*, with both outperforming *Reflexion*. Notably, integrating *CodeT* into *FlowGen*$_{Scrum}$ resulted in statistically significant improvements, achieving the highest Pass@1 scores. Our analysis also reveals that the development activities impacted code smell and exception handling differently, with design and code review adding more exception handling and reducing code smells. Finally, *FlowGen* models maintain stable Pass@1 scores across GPT3.5 versions and temperature values, highlighting the effectiveness of software process models in enhancing the quality and stability of LLM-generated code.

2.2 Introduction

The recent surge of Large Language Models (LLMs) has sparked a transformative phase in programming and software engineering. With tools like ChatGPT (OpenAI, 2023) or LLaMA (Touvron et al., 2023), researchers have demonstrated the potential of LLMs in generating commit messages (Y. Zhang et al., 2024), resolving merge conflicts (Shen, Yang, Pan, & Zhou, 2023), generating tests (Schäfer, Nadi, Eghbali, & Tip, 2023; Xie, Chen, Zhi, Deng, & Yin, 2023; Yuan, Lou, et al., 2023), method renaming (AlOmar, Venkatakrishnan, Mkaouer, Newman, & Ouni, 2024), and even facilitating log analytics (L. Ma et al., 2024; Z. Ma, Chen, Kim, Chen, & Wang, 2024).

Among all development activities, code generation has received much attention due to its potential to reduce development costs. As LLMs are becoming increasingly integral to software development, various techniques have emerged in LLM-based code generation. For example, prompting techniques like few-shot learning (Kumar & Talukdar, 2021; Yuan, Liu, et al., 2023) have been shown to improve code generation results. In particular, few-shot learning coupled with few-shot sampling (Kang, Yoon, & Yoo, 2023; Z. Ma et al., 2024) or information retrieval augmented technique (J. Chen, Lin, Han, & Sun, 2023; Nashid, Sintaha, & Mesbah, 2023) have been shown to improve code generation. Moreover, one can integrate personalization in the prompt, instructing LLMs to be domain experts in a specific field, which can further improve LLM responses (Shao,

Li, Dai, & Qiu, 2023; White, Hays, Fu, Spencer-Smith, & Schmidt, 2023). Such personalization techniques highlight the potential of using multiple LLMs working together to assist in complex software development activities.

Given the complexity of software development, LLM agents stand out among various LLM techniques. Agents are LLM instances that can be customized to carry out specific tasks that replicate human workflow (Dong, Jiang, et al., 2023; Hong et al., 2023). Recently, multi-agent systems have achieved significant progress in solving complex problems in software development by emulating development roles (Dong, Jiang, et al., 2023; Hong et al., 2023; Qian et al., 2023). MetaGPT, introduced by Hong et al. (2023), integrated development workflow using standard operating procedures by assigning specific roles (e.g., a designer or a developer) to LLM agents. Dong, Jiang, et al. (2023) developed self-collaboration, which assigns LLM agents to work as distinct "*experts*" for sub-tasks in software development. Qian et al. (2023) proposed an end-to-end framework for software development through self-communication among the agents.

Despite the promising applications of LLMs in automating software engineering tasks, it is pivotal to recognize that software development is a collaborative and multi-faceted endeavor. In practice, developers and stakeholders work together, following certain software process models like *Waterfall*, *Test-Driven-Development (TDD)*, and *Scrum*. Even though there is a common community agreement on the pros and cons of each process model (Fowler, 2005), the impact of adopting these process models for LLM code generation tasks remains unknown. In particular, will emulating different process models impact the generated code quality in different aspects, such as reliability, code smell, and functional correctness?

While some research has explored integrating multi-agents within LLM frameworks (Qian et al., 2023; Y. Xu et al., 2023; Yang, Yue, & He, 2023), their research focus diverges from the influence of the software process model on code generations for several reasons: 1) Y. Xu et al. (2023) do not adhere to specific process models, and 2) both Dong, Jiang, et al. (2023) and Qian et al. (2023) focus solely on *Waterfall-like* models, neglecting *TDD* and *Scrum*, which may have different impact on code generations. Importantly, none of the aforementioned studies conduct a fine-grain analysis of how different development activities affect code quality metrics, such as code smell and reliability, other than the Pass@1 score. Our study takes further steps to analyze the impacts of different agents

within the process models on code generation and their influence on other code quality attributes.

This research presents a novel multi-agent LLM-based code generation framework named *FlowGen*. *FlowGen* integrates diverse prompt engineering techniques, including chain-of-thought (J. Li, Li, Li, & Jin, 2023; Wei et al., 2022), prompt composition (P. Liu, Yuan, Fu, Jiang, et al., 2023; Yuan, Lou, et al., 2023), and self-refinement (Madaan et al., 2024), with a focus on emulating the flow of development activities in various software process models. Specifically, we implemented three popular process models into *FlowGen*: $FlowGen_{Waterfall}$, $FlowGen_{TDD}$, and $FlowGen_{Scrum}$. Each process model emulates a real-world development team involving several LLM agents whose roles (i.e., requirement engineer, architect, developer, tester, and scrum master) correspond to common software development activities. The agents work collaboratively to produce software artifacts and help other agents review and improve the artifacts in every activity.

We evaluate *FlowGen* on four popular code generation benchmarks: *HumanEval* (M. Chen et al., 2021), *HumanEval-ET* (Dong, Ding, et al., 2023), *MBPP* (Austin et al., 2021), and *MBPP-ET* (J. Liu, Xia, Wang, & Zhang, 2023). We apply zero-shot learning to avoid biases in selecting few-shot samples (J. Xu et al., 2022). To compare, we also apply zero-shot learning on GPT-3.5 as our baseline (*RawGPT*). We repeat our experiments five times to account for variability in LLM's responses and report the average value and standard deviation. To study code quality, in addition to Pass@1, we run static code checkers to detect the prevalence of code smells in the generated code. Our evaluation shows that $FlowGen_{Scrum}$'s generated code achieves the highest accuracy (Pass@1 is 75.2 for HumanEval, 65.5 for HumanEval-ET, 82.5 for MBPP, and 56.7 for MBPP-ET), surpassing *RawGPT*'s Pass@1 by 5.2% to 31.5%. While *FlowGen*, in general, is more stable than *RawGPT*, $FlowGen_{Scrum}$ exhibits the most stable results with an average standard deviation of only 1.3% across all benchmarks.

Additionally, we compare $FlowGen_{Scrum}$ with state-of-the-art techniques: *CodeT* (B. Chen et al., 2023) and *Reflexion* (Shinn, Cassano, Gopinath, Narasimhan, & Yao, 2024). Both $FlowGen_{Scrum}$ and *CodeT* outperform *Reflexion* significantly in terms of Pass@1 across all benchmarks, with $FlowGen_{Scrum}$ achieving a higher Pass@1 than *CodeT* in MBPP. Furthermore, the integration of *CodeT* into $FlowGen_{Scrum}$ demonstrates the highest Pass@1, highlighting the potential of integrating other techniques with *FlowGen* for improved code generation.

We further study the impact of each development activity on code quality. We find that removing the testing activity in the process model results in a significant decrease in Pass@1 accuracy (17.0% to 56.1%). Eliminating the testing activity also leads to a substantial increase in error and warning code smell densities. We also find that the design and code review activities reduce refactor and warning code smells, and improve reliability by adding more exception handling code. Nevertheless, *FlowGen* consistently outperforms *RawGPT* by reducing code smells and enhancing exception handling. Finally, we find that the GPT model version plays a significant role in the quality of generated code, and *FlowGen* helps ensure stability across different versions of LLMs and temperature values.

We summarize the main contributions as follows:

(1) **Originality**: We introduce a multi-agent framework called *FlowGen*, incorporating software process models from real-world development practice. We integrate agents acting as requirement engineers, architects, developers, testers, and scrum masters, and study how their interaction improves code generation and code quality.

(2) **Technique**: We integrate prompt engineering techniques like chain-of-thought, prompt composition, and self-refinement to facilitate interactions among the agents. We implement three recognized process models: $FlowGen_{Waterfall}$, $FlowGen_{TDD}$, and $FlowGen_{Scrum}$, but the technique can be easily extended to emulate other process models or development practices (e.g., DevOps).

(3) **Evaluation**: We conduct a fine-grained evaluation on the quality of the generated code using four popular code generation benchmarks: *Humaneval* (M. Chen et al., 2021), *Humaneval-ET* (Dong, Ding, et al., 2023), *MBPP* (Austin et al., 2021), *MBPP-ET* (J. Liu et al., 2023), comparing agent interactions and their effect on both accuracy (Pass@1) and other code quality metrics (e.g., smells). We manually checked the generated code and discussed the reasons for test failures. Finally, we examined how model versions and temperature settings affect code generation stability.

(4) **Data Availability**: To encourage future research in this area and facilitate replication, We made our data and code publicly available online (Anonymous, 2024).

Organization. Section 2.3 discusses background and related work. Section 2.4 provides the details of *FlowGen*. Section 2.5 evaluates our *FlowGen*. Section 2.6 provides a discussion on future work. Section 2.7 discusses threats to validity. Section 2.8 concludes the study.

2.3 Background & Related Works

In this section, we discuss the background of software process models and LLM agents. We also discuss related work on LLM-based code-generation.

2.3.1 Background

Software Development Process. Software development processes encompass methodologies and practices that development teams use to plan, design, implement, test, and maintain software. The primary goal of a software process is to assist the development teams in producing high-quality software. Generally, different software process models involve the same set of development activities, such as requirement, design, implementation, and testing, but differ in how the activities are organized. Because of the variation, each software process model has its strengths and weaknesses based on the project type, teams, and experience (Fowler, 2005).

In particular, three well-known and widely adopted software process models were created over the years: *Waterfall*, *Test-Driven-Development (TDD)*, and *Scrum*. *Waterfall* (Bassil, 2012) is often used in safety-critical systems where development teams must adhere to a linear path, and each software development activity builds upon the previous one. *TDD* and *Scrum* are both variants of the agile development model. Compared to *Waterfall*, agile process models focus more on iterative and incremental development and adapting to change. *TDD* (Maximilien & Williams, 2003) emphasize writing tests before writing the actual code to improve software design and quality. *Scrum* highlights the importance of collaboration and communication in software development. *Scrum* prescribes for teams to break work into time-boxed iterations called sprints. During these sprints, teams focus on achieving specific goals (e.g., user stories), ensuring a continuous discussion among teams to handle any unexpected risks throughout the development process.

LLM Agents. LLM agents are artificial intelligence systems that utilize LLM as their core computational engines to understand questions and generate human-like responses. LLM agents can refine their responses based on feedback, learn from new information, and even interact with other AI agents to collaboratively solve complex tasks (Hong et al., 2023; Park et al., 2023; Qian et al., 2023; Y. Xu et al., 2023). Through prompting, agents can be assigned different roles (e.g., a developer or a tester) and provide more domain-specific responses that can help improve the answer (Hong et al., 2023; Shao et al., 2023; White et al., 2023).

One vital advantage of agents is that they can be implemented to interact with external tools. When an agent is reasoning the steps to answer a question, it can match the question/response with corresponding external tools or APIs to construct or refine the response. For instance, an LLM agent that represents a data analysis engineer can apply logical reasoning to generate corresponding SQL query statements, invoke the database API to get the necessary data, and then answer questions based on the returned result. When multiple LLM agents are involved, they can collaborate and communicate with each other. Such communication is essential for coordinating tasks, sharing insights, and making collective decisions. Hence, defining how the agents communicate can help optimize the system's overall performance (Xi et al., 2023), allowing agents to undertake complex projects by dividing tasks according to their domain-specific skills or knowledge.

The software development process plays a crucial role in software development, fundamentally involving communication among various development roles. Given the demonstrated capability of Large Language Model (LLM) agents to mimic domain experts in specific fields (Hong et al., 2023; Qian et al., 2023; White et al., 2023), this study leverages LLM agents to represent diverse development roles and conduct their associated duties. Our research establishes a collaborative team of LLM agents designed to emulate these process models and roles, aiming to enhance code generation.

2.3.2 Related Works

Code generation is a thriving field of research because of its potential to reduce development costs. In particular, *prompt-based* and *agent-based* code generation techniques are two of the most prevalent directions.

Prompt-based Code Generation. Prompt-based code generation employs a range of techniques to refine prompts, ultimately leading to the generation of expected code. For example, J. Li et al. (2023) propose using structured prompts containing code information (e.g., branch and loop structures) to improve the generated code. Nashid et al. (2023) retrieval code demos similar to the given task and include them in the prompt to improve code generation. Ruiz, Grishina, Hort, and Moonen (2024) use translation techniques for program repair, where buggy programs are first translated into natural language or other programming languages. The translated code is used as a prompt to generate new/fixed code with the same feature. Schäfer et al. (2023) iteratively refine prompts based on feedback received from interpreters or test execution results. Kang et al. (2023) provide specific instructions, test method signature, and bug report as part of the prompt for generating test code to reproduce bugs. Xie et al. (2023) parse the code to identify the focal method and related code context, which are given in the prompt for test code generation. Yuan, Lou, et al. (2023) apply a prompt composition technique by first asking an LLM to provide a high-level description of a method, and then the description is used as part of the prompt to enhance test code generation. B. Chen et al. (2023) introduced *CodeT*, a framework that employs self-generated tests to evaluate the quality of generated code. Shinn et al. (2024) presented *Reflexion*, which utilizes an evaluator LLM to provide feedback for enhancing future decision-making processes.

Agent-based Code Generation. Agent-based code generation emphasizes on the importance of role definition and communication among multiple LLM agents. Some approaches incorporate external tools as agents. For example, Huang, Bu, Zhang, Luck, and Cui (2023) introduce the test executor agent, employing a Python interpreter to provide test logs for LLMs. Zhong, Wang, and Shang (2024) introduces a debugger agent that utilizes a static analysis tool to build control flow graph information, guiding LLMs in locating bugs. Meanwhile, other studies (Dong, Jiang, et al., 2023; Hong et al., 2023; Qian et al., 2023) task LLMs as agents by emulating diverse human roles, including analysts, engineers, testers, project managers, chief technology officers (CTOs), etc. Nevertheless, these studies miss key roles in the development activities (e.g., only has analysts, coders, and testers (Dong, Jiang, et al., 2023)) or focus more on the business side of the roles (e.g., employ CTO and CEO) (Qian et al., 2023). In our work, we try to follow the Waterfall model that is proposed in the software engineering literature and create agents that correspond to every

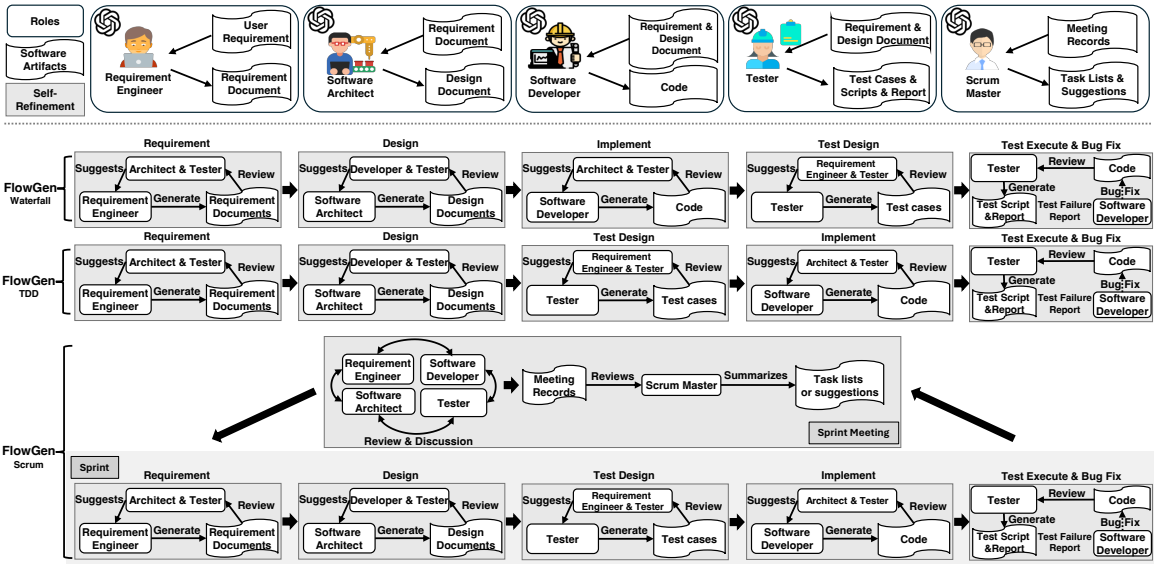

Figure 2.1: An overview of $FlowGen_{Waterfall}$, $FlowGen_{TDD}$, and $FlowGen_{Scrum}$.

development activity. These approaches follow the *Waterfall* model to communicate among these roles, with variation in the prompts and roles, ultimately improving code generation.

In comparison, our research leverages LLM agents to emulate multiple software development process models, while prior research focuses only on the *Waterfall* model (Dong, Jiang, et al., 2023; Hong et al., 2023; Qian et al., 2023). We implement several prompting techniques, but more importantly, we emphasize on how various process models and the associated development activities affect the generated code. Different from prior works which only study functional correctness, we study several additional dimensions of code quality, including code design, code smell, convention issues, and reliability. We also explore why the generated code fails the tests and the sensitivity of the results across LLM model versions and temperature values.

2.4 Methodology

We propose *FlowGen*, an agent-based code generation technique based on emulating different software processes. Figure 2.1 shows the overview of *FlowGen*: (1) define the roles and their responsibilities; (2) use LLM agents to represent these roles; and (3) complete the interactions among these agents according to the software process models. In each development activity, we implement chain-of-thought and self-refinement to improve the quality of the generated artifacts.

In particular, we study and compare three software process models: *Waterfall*, *TDD*, and *Scrum*. Nevertheless, *FlowGen* can be easily adapted to different process models. We use zero-shot learning in all our experiments to avoid biases in selecting data samples. Below, we discuss *FlowGen* in detail.

2.4.1 Using LLM Agents to Represent Different Development Roles in Software Process Models

In *FlowGen*, we create LLM agents who are responsible for the main development activities: requirement, design, implementation, and testing. Hence, to emulate a software process model, we reorganize the communication and interaction among different agents. The benefit of such a design is that it maximizes the extensibility and reusability of the agents, and *FlowGen* can be easily adapted to different process models. We implement four agents whose role corresponds to the common development activities: *Requirement Engineer*, *Architect*, *Developer*, and *Tester*. For *Scrum*, we introduce an additional role – *Scrum Master*.

We designed these roles to use the same prompt template across different process models (with different terms such as user stories v.s. requirement) to investigate the effectiveness of process models on code generation. The exact words that we used for the prompts can be found online (Anonymous, 2024). The role-specific details of our prompt are:

```
{
  "Role": "You are a [role] responsible for [task]",
  "Instruction": "According to the Context please [role-specific instruction]",
  "Context": "[context]"
}
```

In this prompt template (inspired by MetaGPT (Hong et al., 2023) and Self-Collaboration (Dong, Jiang, et al., 2023)), `role` refers to one of the roles (e.g., Requirement Engineer) that corresponds to the development activity, and `task` describes the duties for the `role` (e.g., analyze and create requirement documents). `Instruction` leverages chain-of-thought reasoning (J. Li et al., 2023; Wei et al., 2022) and refers to role-specific instruction listed in steps, such as 1) analyzing the requirement and 2) writing the requirement documentation. Finally, `Context` contains the

programming question, the agent conversation history, or the agent-generated artifacts. `Context` includes all necessary information that helps the agents to make a next-step decision based on the current conversation and generated results.

Table 2.1 shows the `tasks`, `instructions`, and `contexts` for every development role. In general, every role takes the output from the prior development activities as input (i.e., `context`). For example, an architect writes a design document based on the requirement document generated by a requirement engineer. We design developers and testers to have multiple tasks. Developers are responsible for writing code and fixing/improving the code based on suggestions. We design testers using a prompt composition technique, which is shown to improve the LLM-generated result (P. Liu et al., 2023; Yuan, Lou, et al., 2023). First, testers design a test. Then, testers write and execute the tests based on the design. On average, *FlowGen* generates four tests for each problem before the review meeting and six after the meeting. It is important to note that oracles are kept aside and never used in the code-generation process. Finally, testers generate a test failure report.

Developers receive the test failure report to fix the code. In addition to the tasks described in Table 2.1, all roles have one common task, which is to provide feedback to other roles for further improvement (e.g., for code review).

2.4.2 Communications Among Agents

One of the most important aspects of LLM agents is how the agents communicate. A recent survey paper (Xi et al., 2023) shows that one common communication pattern is sequential, i.e., ordered, where one agent communicates to the next in a fixed order. Another pattern is disordered, where multiple agents participate in the conversation. Each agent gets the context separately and outputs the response in a shared buffer. Then, the responses can be summarized and used in the next decision-making process. Based on the software process models and the two aforementioned communication patterns, we implement three interaction models for the agents: $FlowGen_{Waterfall}$, $FlowGen_{TDD}$, and $FlowGen_{Scrum}$ (Figure 2.1). The details of our multi-agent communication history are available online (Anonymous, 2024).

FlowGen$_{Waterfall}$

$FlowGen_{Waterfall}$ follows the *Waterfall* model and implements an ordered communication among

Table 2.1: Tasks, instructions, and corresponding contexts that are used for constructing the prompts for the development roles.

Role	Task	Instruction	Context
Requirement Engineer	Analyze and generate requirement documentation from the context.	1) Analyze the requirement; and 2) Write a requirement document.	Programming problem description.
Architect	Design the overall structure and high-level components of the software.	1) Read the context documents; and 2) Write the design document. The design should be high-level and focus on guiding the developer in writing code.	Requirement document.
Developer	Write code in Python that meets the requirements.	1) Read the context documents; and 2) Write the code. Ensure that the code you write is efficient, readable, and follows best practices.	Requirement and design documents.
	Fix the code so that it meets the requirements.	1) Read the test failure reports and code suggestions from the context; and 2) Rewrite the code.	Original code, test failure report, and suggestions for improvement.
Tester	Design tests to ensure the software satisfies feature needs and quality.	1) Read context documents; and 2) Design test cases.	Requirement and design documents.
	Write a Python test script using the unittest framework.	1) Read the context documents; 2) Write a Python test script; and 3) Follow the input and output given by the requirement.	Test case design and requirement documents.
	Write a test failure report.	1) Read the test execution result; and 2) Analyze and generate a test failure report.	Test execution result.
Scrum Master	Summarize and break down the discussion into a task list for the scrum team.	1) Read and understand the context; and 2) Define the tasks for development roles.	Meeting discussion.

the agents. Given a programming problem, the problem goes through the requirement analysis, design, implementation, and testing. One thing to note is that the test result from our generated tests is redirected to the developer agent in our implementation of $FlowGen_{Waterfall}$ so developers can fix and improve the code.

$FlowGen_{TDD}$

In the design of $FlowGen_{TDD}$, we follow the ordered communication pattern and organize the development activities so that testing happens after design and before implementation. Once the tests are written, the developer agent considers the test design when implementing the code. When

the implementation is finished, we execute the tests. If a test fails, the developer agent is asked to examine the code and resolve the issue.

FlowGen$_{Scrum}$

Compared to *Waterfall* and *TDD*, *Scrum* involves one additional role, the *Scrum Master*. There are also additional *Sprint meetings* among the agents. Note that, different from *FlowGen$_{Waterfall}$*, we use the agile terminologies in the prompt (e.g., we use user story instead of requirement document) when implementing *FlowGen$_{Scrum}$*. We follow a disordered communication pattern in the design of *FlowGen$_{Scrum}$*, because, in sprint meetings, every development role can provide their opinion (e.g., to simulate the planning poker process). Every development role, except the *Scrum Master*, reads the common context (e.g., description of the programming problem) from a common buffer. Then, every role provides a discussion comment and is saved back in the buffer. Therefore, every role is aware of all the comments. Then, the *Scrum Master* summarizes the entire discussion and derives a list of user stories for each development role. During the sprint, similarly to *Waterfall* and *TDD*, the four development roles carry out the development activities in sequence. At the end of the sprint, the agents will start another sprint meeting to discuss the next steps, such as releasing the code or needing to fix the code because of test failures.

Self-Refinement

We implement self-refinement (Madaan et al., 2024), which tries to refine the LLM-generated result through iterative feedback, to further improve the generated artifacts from every development activity. In all three variations of *FlowGen*, we assign other agents to review the generated artifacts for every development activity and provide improvement suggestions. We assign the agents from both the downstream activity and the tester to examine the generated artifacts and provide suggestions. The suggestions are then considered for the re-generation of the artifacts. We include the tester in every development activity to emulate the DevTestOps practice (Testsigma, n.d.), where testers are involved in all development activities and provide feedback on the quality aspects. For example, once a requirement engineer generates a requirement document, both the architect and tester would read the document and provide suggestions for improvement. Then, the requirement engineer will re-generate the requirement document based on the previously generated document and suggestions. At the development and testing activity, the tester will generate a test failure report

if any of the LLM-generated tests fail or if the code cannot be executed (e.g., due to syntax error). The test failure report is then given to the developer for bug fixing. We repeat the process t times to self-refine the generated code. In our implementation, we currently set t=3. If the code still cannot pass the test, we repeat the entire software development process.

2.4.3 Implementation and Experiment Settings

Environment. We use GPT3.5 (version gpt-3.5-turbo-1106) as our underlying LLM due to its popularity and wide usage in code generation research (Dong, Jiang, et al., 2023; J. Li et al., 2023; Shinn et al., 2024). We leverage OpenAI's APIs (version 0.28.1) to interact with GPT. We send prompts using JSON format (White et al., 2023) and send all the conversation history as part of the prompt (Hong et al., 2023). We set the temperature value to 0.8 and explore the effect of the temperature value in RQ3. We implemented *FlowGen* using *Python 3.9*.

Benchmark Datasets. We follow prior studies (J. Li et al., 2023; Zhong et al., 2024; Zhou, Yan, Shlapentokh-Rothman, Wang, & Wang, 2023) and evaluate the code generation result using four benchmarks: HumanEval, HumanEval-ET, MBPP (Mostly Basic Python Programming), and MBPP-ET. These benchmarks contain both the programming problems and tests for evaluation. Given a programming problem, we consider that a generated code snippet is correct if it can pass all the provided tests. HumanEval (M. Chen et al., 2021) has 164 programming problems, and MBPP (Austin et al., 2021) has 427 programming problems (we use the sanitized version released by the original authors) and three test cases for each problem. We also use the dataset published by Dong et al. (Dong, Ding, et al., 2023), where they use the same problems as HumanEval and MBPP but offer stronger evaluation test cases (around 100 test cases for each problem, called HumanEval-ET and MBPP-ET). All these benchmarks use Python as the programming language. Each programming problem contains the INPUT pairs <method signature, method description, invoke examples> and expect the code as OUTPUT.

Evaluation Metric. To evaluate the quality of the generated code, we use the Pass@K metric (M. Chen et al., 2021; Dong, Jiang, et al., 2023). Pass@K evaluates the first K generated code's functional accuracy (i.e., whether the generated code can pass all the test cases). In this work, we set K=1 to evaluate if the first generated code can pass all the provided test cases. A Pass@1 of

Table 2.2: Average and standard deviation of the Pass@1 accuracy across five runs, with the best Pass@1 marked in bold. The numbers in the parentheses show the percentage difference compared to *RawGPT*. Statistically significant differences are marked with a "*".

	HumanEval	HumanEval-ET	MBPP	MBPP-ET
RawGPT	64.4±3.7	49.8±3.0	77.5±0.8	53.9±0.7
$FlowGen_{Waterfall}$	69.5±2.3 (+7.9%)*	59.4±2.5 (+19.2%)*	76.3±0.9 (-1.5%)	51.1±1.7 (-5.2%)*
$FlowGen_{TDD}$	69.8±2.2 (+8.4%)*	60.0±2.1 (+20.5%)*	76.8±0.9 (-1.0%)	52.8±0.7 (-2.1%)*
$FlowGen_{Scrum}$	**75.2±1.1 (+16.8%)***	**65.5±1.9 (+31.5%)***	**82.5±0.6 (+6.5%)***	**56.7±1.4 (+5.2%)***

100 means 100% of the generated code can pass all the tests in the first attempt. We use Pass@1 because it is a stricter criterion, reflecting situations where developers do not have the groundtruth for automatically evaluating multiple attempts.

2.5 Results

We evaluate *FlowGen* with four research questions (RQs).

RQ1: What is the code generation accuracy of *FlowGen*?

Motivation. In this RQ, we emulate the three process models using LLM agents and compare their results on code generations. Such results may provide invaluable evidence for future researchers seeking to optimize process models for code generation within their specific business domain.

Approach. As a baseline for comparison, we directly give the programming problems to ChatGPT (which we refer to as *RawGPT*). Although prior works (Kang et al., 2023; Le & Zhang, 2023; J. Li et al., 2023; Nashid et al., 2023) show that few-shot learning can improve the results from LLMs, they can be biased on how the few-shot samples are selected (J. Xu et al., 2022). Hence, we use zero-shot learning in our experiment. To control for randomness in the experiment, we ensure all these experiments use the same temperature value (t=0.8) and the same model version (*gpt-3.5-turbo-1106*). Finally, we repeat each *FlowGen* five times and report the average Pass@1 and standard deviation across the runs. We also conduct a student's t-test to study if *FlowGen*'s results are statistically significantly different from *RawGPT*.

Result. ***$FlowGen_{Scrum}$ shows a consistent improvement over RawGPT, achieving 5.2% to 31.5%***

improvement in Pass@1. Table 2.2 shows the Pass@1 accuracy of *FlowGen* across different process models on the benchmark datasets studied. As shown in the Table 2.2, for HumanEval and HumanEval-ET, all of the studied process models have 7.9% to even 31.5% improvement in Pass@1 compared to *RawGPT*, and the improvements are all statistically significant ($p <= 0.05$). For MBPP and MBPP-ET, $FlowGen_{Scrum}$ also has statistically significant improvements of 5.2% to 6.5%, even though we see a slight decrease when adopting $FlowGen_{Waterfall}$ and $FlowGen_{TDD}$ to MBPP and MBPP-ET.

Despite slight variations in code generation responses form LLM across executions, **we find stable standard deviations of Pass@1, ranging from 0.6% to 3.7% across all process models and benchmarks**. In particular, $FlowGen_{Scrum}$ has the lowest standard deviation (0.6% to 1.9%, an average of 1.2%), while *RawGPT* has the highest standard deviation (0.5% to 3.7%, an average of 2%). Following $FlowGen_{Scrum}$, $FlowGen_{Waterfall}$ has the second highest standard deviation, with $FlowGen_{TDD}$ is ranking third. In conclusion, although the models generally have consistent Pass@1 across runs, $FlowGen_{Scrum}$ consistently produces the most stable results.

There are potential issues in the tests provided by the benchmarks, which may hinder the Pass@1 of FlowGen. Table 2.3 provides a breakdown of failure types from the Python Interpreter (Python, 2005) across various process models and benchmarks. For example, `IndexError` happens when the generated code does not handle an out-of-bound index, causing an exception to be thrown. While we repeat our experiment 5 times, the standard deviation across runs is low; hence, we represent test failure from only one of the runs. Aligned with the findings from Table 2.2, $FlowGen_{Scrum}$ has the lowest `AssertionError` compared to other models (i.e., higher pass rate). We also notice that `SyntaxError` is more evident in *RawGPT*, as expected due to the absence of a code review and testing process. However, there are still higher test failures in $FlowGen_{Waterfall}$ and $FlowGen_{TDD}$ caused by increased occurrences of `ValueError`, `TypeError`, `IndexError` and `NameError`, for MBPP and MBPP-ET as seen in Table 2.3. Upon manual investigation of prevalent test failures types, we discover that $FlowGen_{Waterfall}$ and $FlowGen_{TDD}$ **introduce various validations and enforce programming naming conventions in the generated code, *which may help improve code quality but cause tests to fail***. For example, Listing 2.1 depicts the provided tests and the generated code for MBPP-582, of which the objective is to *"Write a function to*

Table 2.3: *FlowGen* test failure categorization. Failure types are generated from Python Interpreter (Python, 2005). Darker red indicates higher percentages of the failure categories in the generated code across the models. Percentages are calculated by the ratio of specific failure types to the total number of failed tests across different process models.

		Failure Categories								
Benchmarks	Model	Assertion	Syntax	Name	Type	Index	Value	Recursion	Attribute	Total
HumanEval	RawGPT	36 (24%)	18 (100%)	11 (58%)	2 (17%)	2 (50%)	1 (12%)	0 (0%)	0 (0%)	70 (33%)
	Waterfall	39 (26%)	0 (0%)	3 (16%)	3 (25%)	0 (0%)	1 (12%)	0 (0%)	0 (0%)	46 (22%)
	TDD	39 (26%)	0 (0%)	4 (21%)	3 (25%)	1 (25%)	5 (62%)	0 (0%)	0 (0%)	52 (24%)
	Scrum	38 (25%)	0 (0%)	1 (5%)	4 (33%)	1 (25%)	1 (12%)	0 (0%)	0 (0%)	45 (21%)
HumanEval-ET	RawGPT	39 (21%)	9 (82%)	13 (43%)	1 (25%)	4 (57%)	2 (18%)	0 (0%)	0 (0%)	68 (27%)
	Waterfall	49 (26%)	1 (9%)	7 (23%)	0 (0%)	1 (14%)	3 (27%)	0 (0%)	0 (0%)	61 (24%)
	TDD	50 (26%)	1 (9%)	6 (20%)	2 (50%)	1 (14%)	4 (36%)	0 (0%)	0 (0%)	64 (25%)
	Scrum	52 (27%)	0 (0%)	4 (13%)	1 (25%)	1 (14%)	2 (18%)	0 (0%)	0 (0%)	60 (24%)
MBPP	RawGPT	66 (23%)	7 (70%)	7 (41%)	7 (25%)	2 (67%)	0 (0%)	1 (100%)	1 (50%)	91 (26%)
	Waterfall	76 (27%)	1 (10%)	5 (29%)	6 (21%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	88 (25%)
	TDD	86 (30%)	1 (10%)	5 (29%)	8 (29%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	100 (29%)
	Scrum	58 (20%)	1 (10%)	0 (0%)	7 (25%)	1 (33%)	0 (0%)	0 (0%)	1 (50%)	68 (20%)
MBPP-ET	RawGPT	133 (24%)	8 (57%)	22 (26%)	26 (24%)	2 (50%)	3 (8%)	1 (100%)	1 (100%)	196 (24%)
	Waterfall	148 (27%)	2 (14%)	22 (26%)	27 (25%)	0 (0%)	15 (39%)	0 (0%)	0 (0%)	214 (27%)
	TDD	147 (27%)	2 (14%)	23 (27%)	28 (26%)	1 (25%)	10 (26%)	0 (0%)	0 (0%)	211 (26%)
	Scrum	123 (22%)	2 (14%)	18 (21%)	28 (26%)	1 (25%)	10 (26%)	0 (0%)	0 (0%)	182 (23%)

check if a dictionary is empty or not". While *RawGPT* passed the provided tests, $FlowGen_{Waterfall}$ and $FlowGen_{TDD}$ failed. This failure is because the generated code contains *strict input validation* to check that the input should be of type *dict*. However, the MBPP-582 provided test uses an input of the type *set*, which fails the validation, causing a `TypeError` exception. Moreover, $FlowGen_{Waterfall}$ and $FlowGen_{TDD}$ enforce the common naming convention format and more meaningful function name (e.g., `my_dict` v.s. `is_dict_empty`), both of which causes `NameError` exception due to *wrong function declaration*, causing test failure. More interestingly, we find that such code standardization may also be misled by the requirement provided by the benchmark itself. For example, the MBPP-582 requirement specifies expected input as `dict`, yet provides a `set` type as the input to the test. The LLM code generation indeed captures this correct requirement by validating that input must be of type `dict`. Such inconsistency in the benchmark may reduce the

Pass@1.

```
# MBPP-582: check if a dictionary is empty
# MBPP Test Case
def Test():
    assert my_dict({10})==False # {10} is a set not a dict
# rawGPT's answer
def my_dict(dict1):
    return len(dict1) == 0
# Waterfall/TDD's answer, the input type must be a dict
def is_dict_empty(input_dict): # function name is renamed from my_dict
    if not isinstance(input_dict, dict):
        raise TypeError("Input is not a dictionary")
    return True if not input_dict else False
```

Listing 2.1: MBPP-582 Test Failure due to *Strict Input Validation* and *Wrong Function Name*.

In MBPP-794 (Listing 2.2), test cases provided by MBPP-ET change the return value from a `boolean` (as is the case in MBPP) to the string "`match!`". Moreover, in MBPP-797, MBPP-ET's test capitalized the last word in uppercase (`range` v.s. "`R`"`ange`). Such non-standard evaluation leads to unfair test results (leads to failure), which may bias the experimental results for MBPP-ET. Such bias suggests that the decrease in Pass@1 rates for MBPP-ET is not solely due to an increase in the number of provided tests.

```
# Example1: Changed return type from boolean to string
assert text_starta_endb("aabbbb") # MBPP 794
assert text_starta_endb("aabbbb")==('match!') # MBPP-ET 794
# Example2: Capitalized the last character in function name
assert sum_in_range(2,5) == 8 # MBPP-797
assert sum_in_Range(2,5) == 8 # MBPP-ET-797
```

Listing 2.2: MBPP-794 & MBPP-797 Test Failure due to Irregularity in Test Cases.

Fixing the issues largely improve *FlowGen*'s Pass@1: 16 to 28 improvement in HumanEval, 29 to 35 in HumanEval-ET, 15 to 21 in MBPP, and 28 to 42 in MBPP-ET. Namely, *FlowGen*'s Pass@1 can achieve over 90 to 95 across all benchmarks. Even though we also observe improvement in *RawGPT*'s Pass@1, the three *FlowGen* models had greater improvement. On average, the three models have a Pass@1 that is 14%, 8%, 1.4%, and 5% better than *RawGPT* in HumanEval,

HumanEval-ET, MBPP, and MBPP-ET, respectively.

The results underscore the deficiencies in the benchmarks, suggesting that the current Pass@1 score of *FlowGen* could represent a lower bound. These preliminary findings highlight the potential of *FlowGen* and suggest that future research should improve the benchmarks by incorporating input checking or consistent naming convention into the tests and subsequently re-evaluate existing code generation techniques.

FlowGen$_{Scrum}$ achieves the best results, with a Pass@1 that is 5.2% to 31.5% better than *RawGPT*. *FlowGen*$_{Scrum}$ also has the most stable results (average standard decision of 1.3% across all benchmarks) among all models. Notably, while *FlowGen*$_{Waterfall}$ and *FlowGen*$_{TDD}$ enhance code quality, such improvements may result in test failures.

RQ2: How do different development activities impact the quality of the generated code?

Motivation. As observed in RQ1, various process models can indeed affect the functional correctness (Pass@1) of the generated code. However, it is equally crucial to understand code quality issues such as code smells and the impact of a development activity on the generated code. This understanding is essential for assessing whether the generated code adheres to industry best practices. Moreover, such insight may offer valuable opportunities for enhancing the design, readability, and maintainability in auto-generated code.

Approach. To study the impact of each development activity on code quality, we remove each activity separately and re-execute *FlowGen*. For example, we first remove the requirement activity in *FlowGen*$_{Waterfall}$ and execute *FlowGen*$_{Waterfall}$. Then, we add the requirement activity back and remove the design activity. We repeat the same process for every development activity. Note that we cannot remove the coding activity since our goal is code generation. Hence, we removed code review at the end of the coding activity.

Code quality considers numerous facets beyond mere functional correctness (Yamashita & Moonen, 2012; Yetiştiren, Özsoy, Ayerdem, & Tüzün, 2023). Other factors, such as code smells, maintainability, and readability, are also related to code quality. Hence, to gain a comprehensive understanding of how code quality changes, we 1) apply a static code analyzer to detect code smells

Table 2.4: Pass@1 and *Error/Warning/Convention/Refactor/Handled-Exception* density (per 10 lines of code) in the full *FlowGen* (with all the development activities) and after removing a development activity. A ***lower*** error/warning/convention/refactor is preferred, and a ***higher*** handled-exception is preferred. Darker red indicates a larger decrease in percentages, while darker green indicates a larger increase in percentages.

Model	Dev. Activities	HumanEval						MBPP					
		Pass@1	Error	Warning	Convention	Refactor	Handled Exception	Pass@1	Error	Warning	Convention	Refactor	Handled Exception
RawGPT	–	64.4	0.25	0.19	0.39	0.30	0.00	77.47	0.22	0.20	1.18	0.30	0.01
FlowGen$_{Waterfall}$	full	69.5	0.01	0.12	0.24	0.21	0.37	76.35	0.03	0.12	0.47	0.23	0.67
	rm-requirement	-1.2 (1.7%)	0.0 (0.0%)	-0.01 (8.3%)	-0.02 (8.3%)	-0.01 (4.8%)	-0.06 (16.2%)	+0.7 (0.9%)	-0.02 (66.7%)	-0.02 (16.7%)	0.0 (0.0%)	+0.02 (8.7%)	-0.09 (13.4%)
	rm-design	-1.2 (1.7%)	+0.01 (100.0%)	+0.02 (16.7%)	+0.02 (8.3%)	0.0 (0.0%)	-0.15 (40.5%)	-1.64 (2.1%)	-0.01 (33.3%)	0.0 (0.0%)	+0.02 (4.3%)	+0.01 (4.3%)	-0.2 (29.9%)
	rm-codeReview	-2.4 (3.5%)	0.0 (0.0%)	0.0 (0.0%)	-0.03 (12.5%)	+0.02 (9.5%)	-0.09 (24.3%)	+0.46 (0.6%)	-0.02 (66.7%)	+0.07 (58.3%)	-0.02 (4.3%)	-0.02 (8.7%)	-0.17 (25.4%)
	rm-test	-39.0 (56.1%)	+0.17 (1700.0%)	+0.01 (8.3%)	+0.07 (29.2%)	+0.1 (47.6%)	+0.1 (27.0%)	-23.7 (31.0%)	+0.09 (300.0%)	+0.05 (41.7%)	+0.36 (76.6%)	+0.01 (4.3%)	-0.01 (1.5%)
FlowGen$_{TDD}$	full	69.8	0.01	0.08	0.33	0.27	0.33	76.77	0.04	0.13	0.71	0.28	0.62
	rm-requirement	-2.9 (4.2%)	+0.01 (100.0%)	+0.01 (12.5%)	0.0 (0.0%)	-0.03 (11.1%)	-0.1 (30.3%)	+1.92 (2.5%)	-0.02 (50.0%)	+0.02 (15.4%)	+0.03 (4.2%)	0.0 (0.0%)	-0.27 (43.5%)
	rm-design	-2.9 (4.2%)	0.0 (0.0%)	+0.02 (25.0%)	-0.06 (18.2%)	+0.03 (11.1%)	-0.13 (39.4%)	+1.22 (1.6%)	-0.02 (50.0%)	-0.03 (23.1%)	-0.03 (4.2%)	+0.04 (14.3%)	-0.24 (38.7%)
	rm-codeReview	-0.5 (0.7%)	-0.01 (100.0%)	+0.02 (25.0%)	-0.04 (12.1%)	+0.03 (11.1%)	-0.09 (27.3%)	+0.98 (1.3%)	-0.03 (75.0%)	+0.01 (7.7%)	-0.05 (7.0%)	+0.1 (35.7%)	-0.09 (14.5%)
	rm-test	-11.9 (17.0%)	+0.07 (700.0%)	+0.04 (50.0%)	-0.02 (6.1%)	-0.05 (18.5%)	-0.1 (30.3%)	-17.3 (22.5%)	+0.11 (275.0%)	+0.14 (107.7%)	+0.16 (22.5%)	-0.01 (3.6%)	+0.16 (25.8%)
FlowGen$_{Scrum}$	full	75.2	0.00	0.13	0.21	0.24	0.15	82.48	0.02	0.17	0.51	0.23	0.44
	rm-requirement	+1.0 (1.3%)	0.0 (0.0%)	0.0 (0.0%)	+0.05 (23.8%)	-0.01 (4.2%)	+0.09 (60.0%)	-1.92 (2.3%)	+0.01 (50.0%)	0.0 (0.0%)	+0.01 (2.0%)	-0.01 (4.3%)	+0.04 (9.1%)
	rm-design	+1.0 (1.3%)	0.0 (0.0%)	-0.02 (15.4%)	-0.03 (14.3%)	0.0 (0.0%)	-0.03 (20.0%)	+0.89 (1.1%)	-0.01 (50.0%)	-0.04 (23.5%)	+0.11 (21.6%)	+0.03 (13.0%)	-0.19 (43.2%)
	rm-codeReview	-2.0 (2.7%)	0.0 (0.0%)	0.0 (0.0%)	-0.03 (14.3%)	0.0 (0.0%)	-0.03 (20.0%)	+0.19 (0.2%)	0.0 (0.0%)	-0.01 (5.9%)	+0.01 (2.0%)	+0.06 (26.1%)	-0.1 (22.7%)
	rm-test	-14.2 (18.9%)	+0.03 (588.2%)	+0.06 (46.2%)	0.0 (0.0%)	-0.03 (12.5%)	+0.07 (46.7%)	-26.5 (32.1%)	+0.13 (650.0%)	+0.1 (58.8%)	+0.41 (80.4%)	-0.03 (13.0%)	+0.14 (31.8%)
	rm-sprintMeeting	-1.6 (2.1%)	0.0 (0.0%)	-0.01 (7.7%)	-0.02 (9.5%)	-0.03 (12.5%)	+0.1 (66.7%)	-3.09 (3.7%)	0.0 (0.0%)	-0.02 (11.8%)	+0.05 (9.8%)	-0.01 (4.3%)	+0.21 (47.7%)

in the generated code and 2) study code reliability by analyzing the exception handling code. To study code smell, design, and readability, we apply Pylint 3.0.4 (Dasgupta & Hooshangi, 2017; Team, n.d.) (a Python static code analyzer) on the generated code. Pylint classifies the detected code smells into different categories such as *error*, *warning*, *convention*, and *refactor*.

We study how the number of detected code smells in each category changes when removing an activity. Since the generated code may have different lengths, we report the density of the code smells in each category. We calculate the code smell density as the total number of code smell instances in a category (e.g., *error*) divided by the total lines of code. To study reliability, we calculate the density of handled exceptions (total number of handled exceptions divided by the total lines of code) since exceptions are one of the most important mechanisms to detect and recover from errors (Fetzer, Felber, & Hogstedt, 2004). For better visualization, we present the density results as per every 10 lines of code. We also ensure reliability of our results by repeating all of the aforementioned approach three times.

Result. ***Testing has the largest impact on the functional correctness of the code, while other development activities only have small impacts. Removing testing causes Pass@1 to decrease by 17.0% to 56.1%.*** Table 2.4 presents changes in Pass@1 and the densities of code smell and

handled exceptions. We show the results for HumanEval and MBPP because they share the same programming problems and generated code with the other two benchmarks. Among all development activities, testing has the largest impact on Pass@1, where removing testing causes a large decrease in Pass@1 (17.0% to 56.1% decrease). The finding implies that LLM's generated tests are effective in improving the functional correctness of code. In both benchmarks, removing sprint meetings in $FlowGen_{Scrum}$ also causes Pass@1 to drop. However, removing other activities only has a small and inconsistent effect on Pass@1. For example, in HumanEval, removing requirement, design, and code review generally causes Pass@1 to decrease (except for $FlowGen_{Scrum}$), but removing these activities improves Pass@1 in MBPP. In other words, most development activities do not significantly contribute to the functional correctness of the generated code.

As shown in Table 2.4, eliminating test activities significantly boosts *error* and *warning* smell densities by an average of 702.2% and 52.2%, respectively. Omitting design raises refactor smell density by an average of 7.1%, and skipping code review leads to a 14.2% average increase in *warning* density. However, removing other development activities shows either a small or inconsistent impact. We also find some differences in the artifacts generated by different roles. For example, although both roles generate documents, requirement engineers specify the acceptance criteria, while architects address time/space complexity. In short, the findings show that **adding design, testing, and code review can help reduce the density of code smell in the generated code**.

Having design and code review activities significantly improves reliability by increasing the density of handled exceptions, while other development activities only have small or no impacts. Removing design and code review activities separately causes the handled exception density to decrease from 20.0% to 43.2% and 14.5% to 27.3%, respectively. Namely, these two activities add exception handling in the generated code, which may help improve reliability. Removing other activities shows a mixed relationship with the density of the handled exception. For example, removing testing in $FlowGen_{TDD}$ causes an increase of handled exception density by 25.8% in MBPP (i.e., testing removes exception handling code) while causing a decrease of 30.3% in HumanEval (i.e., testing adds exception handling code). While the effect of each development may be related to the nature of the benchmarks, our findings show that, in both benchmarks, **adding design and code review activities can help improve code reliability by handling more exceptions in the**

generated code.

FlowGen shows consistent improvement over RawGPT in the quality of the generated code: decreasing the density of error/warning/convention/refactor code smells (6.7% to 96.0%) while significantly increasing handled exception density. The code generated by *RawGPT* has higher *error/warning/convention/refactor* code smell densities than that of $FlowGen_{Waterfall}$, $FlowGen_{TDD}$, and $FlowGen_{Scrum}$. This finding shows all three models improve the quality of the generated code to different degrees. Specifically, compared to *RawGPT*, *FlowGen* decreases the *error* code smell density by 81.8% to 96.0%, *warning* density by 15.0% to 57.9%, *convention* density by 15.4% to 60.2%, and *refactor* density by 6.7% to 30.0%. Meanwhile, *RawGPT* has fewer handled exceptions than *FlowGen*. As Table 2.4 shows, in both HumanEval and MBPP, *RawGPT* has almost zero handled exception, while $FlowGen_{Waterfall}$ generates the most handled-exception (0.37 and 0.67 handled exceptions per every 10 LOC in the two benchmarks), $FlowGen_{TDD}$ ranks second (0.33 and 0.62), and then $FlowGen_{Scrum}$ (0.15 and 0.44). In short, *FlowGen* improves the quality of the generated code by reducing code smells while adding more exception-handling code.

Compared to *RawGPT*, *FlowGen* remarkably improves the quality of the generated code by reducing code smells and adding more exception handling. Testing has the most significant impact on Pass@1 and code smells among all development activities, while having design and code review greatly improve the exception-handling ability.

RQ3: How stable is the *FlowGen* generated code?

Motivation. In LLM, the stability of generated responses can be influenced by several parameters: 1) *temperature*, affecting the randomness in the generated responses, and 2) *model versions*, which may introduce variability due to changes in optimization and fine-tuning (L. Chen, Zaharia, & Zou, 2023a). Understanding and improving the stability of LLMs is crucial for enhancing their trustworthiness, thereby facilitating their adoption in practice. Therefore, in this research question (RQ), we investigate the stability of our *FlowGen* in Pass@1 across four benchmarks, considering various temperature values and model versions.

Approach. We evaluate the Pass@1 of *FlowGen* across four versions of GPT3.5: *turbo-0301*, *turbo-0613*, *turbo-1106*, and *turbo-0125*. The latest version is *turbo-0125* (published in January 2024),

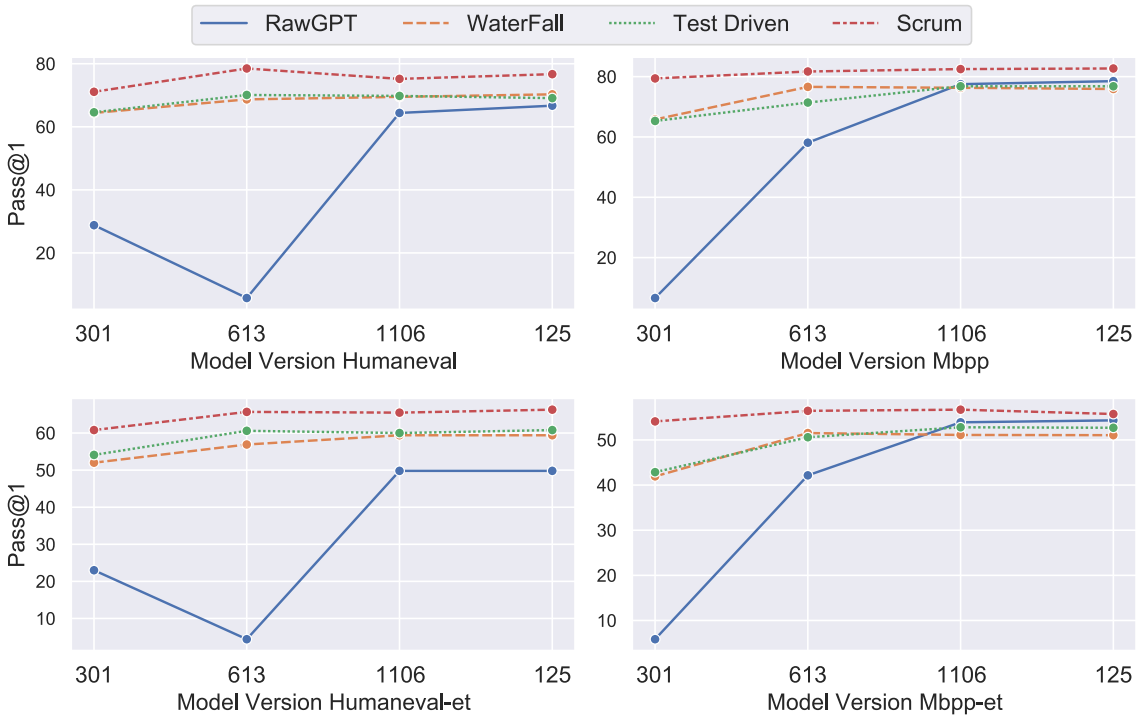

Figure 2.2: Pass@1 across GPT3.5 versions.

and the earlier version is *turbo-0301* (published in March 2023). To avoid the effect of the model version when we vary the temperature, we use the same model version (*turbo-1106*, the version that we used in prior RQs) to study the effect of temperature values. We set the temperature to 0.2, 0.4, 0.6, and 0.8 in our experiment. We execute *RawGPT* and the three variants of *FlowGen* three times under each configuration and report the average Pass@1.

Result. ***RawGPT has extremely low Pass@1 in some versions of GPT3.5, while FlowGen has stable results across all versions. FlowGen may help ensure the stability of the generated code even when the underlying LLM regresses.*** Figure 2.2 shows the Pass@1 for *RawGPT* and the three *FlowGen* across GPT3.5 versions. In earlier versions of GPT3.5 (*0301* and *0613*), *RawGPT* has very low Pass@1 on all benchmarks (e.g., 20 to 30 in HumanEval and HumanEval-ET). In *0301*, MBPP's Pass@1 is even lower with a value around 5. **The findings show that model version may have a significant impact on the generated code**. However, we see that, after adopting our agent-based techniques, all three variants of *FlowGen* achieve similar Pass@1 across GPT3.5 versions. The results indicate that *FlowGen* can generate similar-quality code even if we have an underperformed baseline model.

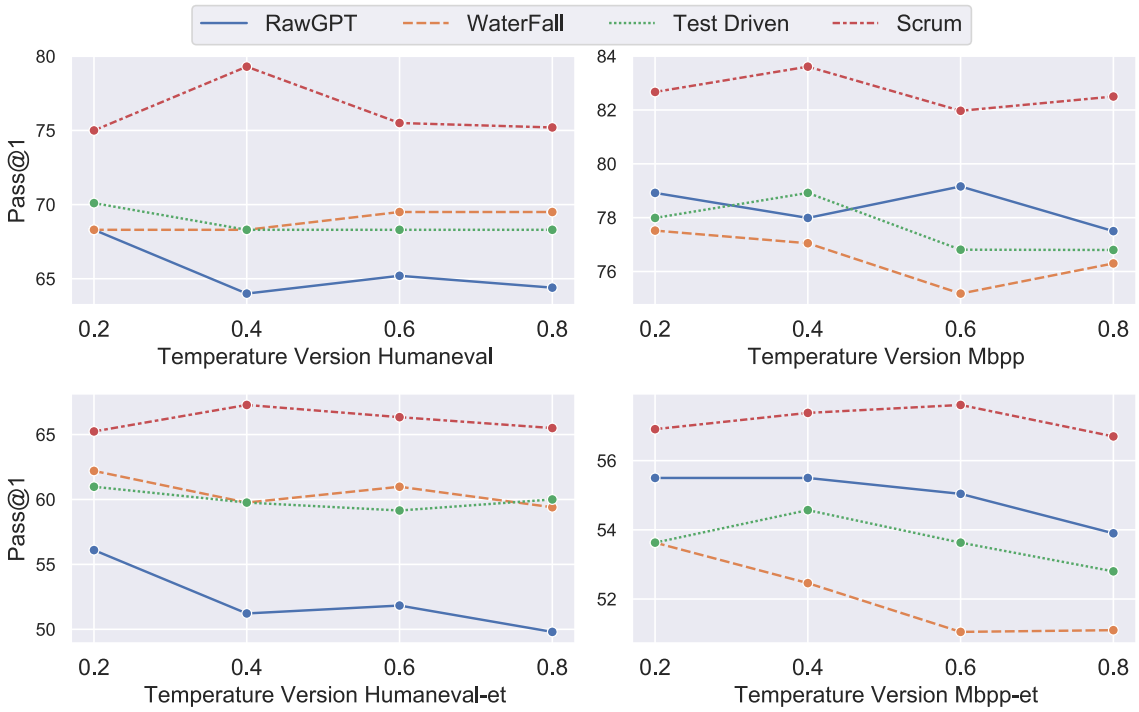

Figure 2.3: Pass@1 across temperature values.

All techniques have a relatively similar Pass@1 when the temperature value changes. Figure 2.3 shows the Pass@1 for all the techniques when the temperature value changes. There is a slight downward trend for *RawGPT* when t increases, but the changes are not significant (Pass@1 is decreased by 2 to 5). For *FlowGen*, and especially $FlowGen_{Scrum}$, we see similar Pass@1 regardless of the temperature value. Although we see a slight increase in the Pass@1 of $FlowGen_{Scrum}$ when t=0.4 (2 to 5 higher compared to when t=0.8 across the benchmarks), the difference is small, and the Pass@1 is almost the same when t is either the lowest (0.2) or largest value (0.8). In short, although temperature values may have an impact on the generated code, the effect is relatively small for *FlowGen*.

FlowGen generates stable results across GPT versions, while we see large fluctuations (14 times difference) in *RawGPT*'s Pass@1. Pass@1 is generally consistent across all models when the temperature value changes.

RQ4: How does *FlowGen* compare with other techniques?

Motivation. *FlowGen* is designed to organize agents to emulate process models and can be combined with other code generation techniques. However, it is crucial to evaluate its performance relative to these techniques to assess the effectiveness in emulating software process models for code generation. While many code generation results use the same benchmarks, our evaluation results cannot be directly compared with other agent-based code generation works (B. Chen et al., 2023; Hong et al., 2023; Huang, Bu, & Cui, 2023; Huang, Bu, Zhang, et al., 2023; Shinn et al., 2024) due to missing information on model versions, temperature values, post-processing steps, specific prompts, or the selection of few-shot samples. Therefore, in this RQ, we compare $FlowGen_{Scrum}$ with other LLM-based baselines under the same environment settings. Moreover, we evaluate an integrated version of $FlowGen_{Scrum}$ to showcase how existing prompting techniques can be combined with *FlowGen*.

Approach. We compare against two state-of-the-art techniques: *CodeT* (B. Chen et al., 2023) and *Reflexion* (Shinn et al., 2024). *CodeT* employs self-generated tests to evaluate the quality of generated code, which is similar to the testing phase of *FlowGen*. *Reflexion* is an agent-based technique that achieves state-of-the-arts Pass@1 on the benchmarks. We apply these two using their released code, replacing the LLM version and temperature with those used by $FlowGen_{Scrum}$, and repeat the experiment five times.

Result. ***While $FlowGen_{Scrum}$ and CodeT achieve similar results, with $FlowGen_{Scrum}$ having a higher Pass@1 in MBPP, they both have higher Pass@1 than Reflexion on all benchmarks (statistically significant).*** Table 2.5 shows the Pass@1 of the techniques. $FlowGen_{Scrum}$ has a statistically significantly higher Pass@1 than *CodeT* in MBPP and similar Pass@1 in other benchmarks (no statistically significant difference). Both techniques achieve higher Pass@1 than *Reflexion* (statistically significant). *Reflexion*'s MBPP results are even worse than *RawGPT*. This observation aligns with the original study (Shinn et al., 2024) where reported similar performance degradation.

Integrating CodeT to $FlowGen_{Scrum}$ further brings statistically significant improvements of Pass@1 by up to 5%. *CodeT* is a general technique where it repeats the code generation and selects the code that passes the most self-generated tests. Hence, as a pilot study, we made the developer agent repeats the implementation activity multiple times to produce several versions of the code (i.e.,

Table 2.5: Average and standard deviation of Pass@1 across five runs, with the best Pass@1 marked in bold.

	HumanEval	HumanEval-ET	MBPP	MBPP-ET
Reflexion	71.3±1.5	55.7±2.8	71.7±0.8	52.0±0.7
CodeT	75.7±0.4	66.9±0.4	79.9±1.3	56.7±0.9
$FlowGen_{Scrum}$	75.2±1.1	65.5±1.9	82.5±0.6	56.7±1.4
$FlowGen_{Scrum+Test}$	**79.3±1.6**	**67.7±1.1**	**83.8±0.6**	**58.7±1.3**

$FlowGen_{Scrum+Test}$). The developer agent then generates multiple test assertions to identify the version with the highest pass rate. The selected code is subsequently submitted to the next stage: the testing activity. Our findings indicate that $FlowGen_{Scrum+Test}$ outperforms $FlowGen_{Scrum}$ and *CodeT*, achieving an average Pass@1 score of 83.8, 58.3, 79.3, and 67.7 on HumanEval, HumanEval-ET, MBPP, and MBPP-ET, respectively. This provides statistically significant improvement over both $FlowGen_{Scrum}$ and *CodeT*.

Our finding highlights the potential of *FlowGen* in boosting the performance of other code generation techniques (and vice versa). Future studies can refine *FlowGen* to incorporate enhancement to each activity for further improvement.

To support these efforts, we have made our code publicly available (Anonymous, 2024) to facilitate further adoption and allow researchers to experiment with different software process models.

Both $FlowGen_{Scrum}$ and *CodeT* outperform *Reflexion* in Pass@1 across all benchmarks, with $FlowGen_{Scrum}$ and *CodeT* demonstrating similar results. The incorporation of *CodeT* into $FlowGen_{Scrum}$ further enhances performance, achieving the highest Pass@1 scores, highlighting the potential of *FlowGen* for code generation tasks.

2.6 Discussion & Future Works

Role of Human Developers in *FlowGen*. Although software process models were originally designed for human-centric development rather than for LLMs, our empirical findings suggest that certain elements of these processes can contribute to better code quality. Every activity in the process model also has different impacts on the generated artifacts. Future research should examine the incorporation of human developers into various phases of the code generation process. Specifically,

humans can play critical roles in the following stages: ***(1) Pre-Execution of FlowGen***: different process models exhibit varying quality (e.g., smell and accuracy). Humans are instrumental in selecting the most appropriate model for a given task. Humans are also essential in providing initial requirements and design specifications. ***(2) During FlowGen Execution***: Humans can oversee review meetings and assist in reviewing/improving generated artifacts. For example, humans can validate the generated requirements with product managers, or verify the quality of the generated code by manual inspection and debugging. The improvement in each activity can also impact the subsequent activities and, hence, affect the final artifacts. ***(3) Post-Execution of FlowGen***: Following the code-generation phase, humans can either accept the generated artifact or request further refinements, offering additional requirements as needed to better meet project goals.

Quality of Code Generation Benchmarks We manually validated all coding problems that failed the tests. We found that the majority of quality issues within the benchmark were in MBPP and MBPP-ET (e.g., bad naming convention or inconsistent test definition). These issues may contribute to reduced Pass@1 scores due to factors beyond logic in the code. It is also important to acknowledge that other benchmarks might present unique challenges that could similarly affect Pass@1 evaluations. Hence, a crucial research direction is to conduct a thorough evaluation of benchmarks for more diverse and accurate evaluations of code generation approaches.

2.7 Threats to Validity

Internal validity. Due to the generative nature of LLM, the responses may change across runs. Variables such as temperature and LLM model version can also impact the generated code. We set the temperate value to be larger than 0 because we want LLMs to be more creative. To mitigate the threat, we execute the LLMs multiple times. As found in RQ1, the standard deviation of the results is small, so the generated results should be consistent. In RQ3, we conducted the experiments using different temperature and versions. The temperature only has a small effect on Pass@1, and versions have a large impact on *RawGPT*. In RQ2, we study the impact of removing every activity. However, having multiple activities may have a tandem effect that further improves code quality. Future studies are needed to study the effects of different combinations of development activities in

code generation.

External validity. We conduct our study using state-of-the-art benchmarks. However, as we discussed, there exist some issues in the provided tests. Moreover, the programming problems are mostly algorithmic, so the findings may not generalize to other. Future studies should consider applying *FlowGen* on different programming tasks. We use GPT3.5 as our underlying LLM. Although one can easily replace the LLM in our experiment, the findings may be different. Future studies on how the results of *FlowGen* change when using different LLMs.

Construct validity. We implement an agent system that follows various software development processes. However, there are many variations of the same process model, and some variations may give better results. Future studies should explore how changing the process models affect the code generation ability. One limitation is the correctness of the generated tests. However, we found that the generated tests still contribute to improving the quality of the generated code. Similar findings are reported on CodeT (B. Chen et al., 2023), where generating tests helps improve code correctness. However, future studies should focus on further improving the generated tests by using traditional software engineering techniques to estimate the oracles or select higher-quality tests (e.g., mutation testing).

2.8 Conclusion

In this research, we emulate various roles in software development, using LLM agents, and structuring their interactions according to established process models. We introduce *FlowGen*, a framework that implements three renowned process models: $FlowGen_{Waterfall}$, $FlowGen_{TDD}$, and $FlowGen_{Scrum}$. We evaluated how these models affect code generation in terms of correctness and code quality on four benchmarks: HumanEval, HumanEval-ET, MBPP, and MBPP-ET. Our findings show that $FlowGen_{Scrum}$ notably enhances Pass@1 by an average of 15% over *RawGPT*, while maintaining the lowest standard deviation (averaging 1.2%). Moreover, we find that development activities such as design and code review significantly reduce code smells and increase the presence of handled exceptions. This indicates that *FlowGen* not only boosts code correctness but also reduces code smells and improves reliability. Compared with other state-of-the-art techniques,

FlowGen$_{Scrum}$ and *CodeT* achieved similar results, with both outperforming *Reflexion*. Integrating *CodeT* into *FlowGen*$_{Scrum}$ further resulted in statistically significant improvements, achieving the highest Pass@1 scores. These insights pave the way for future research to develop innovative development models tailored for LLM integration in software development processes.

In this study, we introduced *FlowGen*, a framework designed to emulate software process models using Large Language Model (LLM) agents, each representing roles such as requirement engineers, architects, developers, and testers. We implemented three variations of *FlowGen*: *FlowGen*$_{Waterfall}$, *FlowGen*$_{TDD}$, and *FlowGen*$_{Scrum}$. Our evaluation across four benchmarks–HumanEval, HumanEval-ET, MBPP, and MBPP-ET–demonstrated the superior performance of *FlowGen*$_{Scrum}$, achieving up to 31.5% improvement in Pass@1 over *RawGPT*.

Our results showed that incorporating software process models into LLM-based code generation significantly enhances code correctness, reduces smells, and improves exception handling. FlowGenScrum consistently outperformed other models, achieving the highest Pass@1 and the lowest standard deviation, indicating more stable and reliable code generation.

Additionally, our comparative analysis with state-of-the-art techniques revealed that FlowGenScrum and CodeT achieved similar results, both outperforming Reflexion. Notably, integrating CodeT into FlowGenScrum resulted in statistically significant improvements, achieving the highest Pass@1. This highlights the robustness and potential of combining structured software development practices with LLM capabilities.

Future research should focus on refining these models, incorporating more sophisticated interactions, and expanding the scope of evaluation to include a broader range of development tasks and environments. By doing so, we can better understand the capabilities of LLMs in software engineering and improve their integration into practical development workflows.

Chapter 3

RobuNFR: Evaluating the Robustness of Large Language Models on Non-Functional Requirements Aware Code Generation

In Chapter 2, we observed that software engineering knowledge influences LLM code generation. For instance, different versions of *GPT-3.5-turbo* exhibited notable differences in Pass@1 scores. However, their impact on NFRs remains unclear. In this chapter, we further explore NFR-aware code generation, with a focus on evaluating the robustness of LLM.

3.1 Abstract

When using LLMs to address Non-Functional Requirements (NFRs), developers may behave differently (e.g., expressing the same NFR in different words). Robust LLMs should output consistent results across these variations; however, this aspect remains underexplored. We propose *RoboNFR* for evaluating the robustness of LLMs in NFR-aware code generation across four NFR dimensions—design, readability, reliability, and performance—using three methodologies: prompt

variation, regression testing, and diverse workflows. Our experiments show that *RoboNFR* reveals robustness issues in the tested LLMs when considering NFRs in code generation. Specifically, under prompt variation, including NFRs leads to a decrease in Pass@1 by up to 39% and an increase in the standard deviation from 0.48 to 2.48 compared to the baseline without NFRs (i.e., Function-Only). While incorporating NFRs generally improves overall NFR metrics, it also results in higher prompt sensitivity. In regression settings, some LLMs exhibit differences across versions, with improvements in one aspect (e.g., reduced code smells) often accompanied by regressions in another (e.g., decreased correctness), revealing inconsistencies that challenge their robustness. When varying workflows, the tested LLMs show significantly different NFR-aware code generation capabilities between two workflows: (1) integrating NFRs and functional requirements into the initial prompt and (2) enhancing Function-Only-generated code with the same NFR.

3.2 Introduction

Code quality is a critical aspect of software development, alongside code correctness, and requires developers to consider Non-Functional Requirements (NFRs) throughout the development process. With the rise of Large Language Models (LLMs) (Q. Zhang et al., 2024), many LLM-based services, such as ChatGPT (OpenAI, 2023), GitHub Copilot (Copilot, 2024a), and Cursor (Cursor, 2024), have become increasingly integrated into modern development workflows. These services are now widely adopted for addressing code generation tasks related to NFRs, either by generating new code with NFR considerations in mind or by enhancing existing code to meet those requirements.

In real-world scenarios, users employ LLMs to address NFRs in various ways. For example, different users might use distinct prompts to express the same NFRs, and these variations can lead to different outcomes, depending on the robustness of the LLM's NFR-aware code generation capabilities. Figure 3.1 illustrates this scenario: for the same coding problem, the outcomes vary depending on whether performance is considered, and further differences emerge when different prompts are used to address performance requirements. Even when generated by the same LLM, code generated without explicit performance requirements relies on exhaustive iteration, leading to

{Task} *Model: gpt-3.5-turbo-0125*
Data: HumanEval-83

```
def starts_one_ends(n):
    """
    Given a positive integer n, return the count of the
numbers of n-digit positive integers that start or end with 1.
    >>> starts_one_ends(1)
    1
    >>> starts_one_ends(2)
    18
    """
```

Function-Only

Finish the following code generation task: {Task}

```
1  def starts_one_ends(n):
2      count = 0
3      for i in range(10**(n-1), 10**n):
4          if str(i)[0] == '1' or str(i)[-1] == '1':
5              count += 1
6      return count
```

Test (TimeOut)

```
7  assert starts_one_ends(2) == 18
8  assert starts_one_ends(18) == 180000000000000000
```

Performance - Prompt 1

Optimize for performance while solving the following code generation task: {Task}

```
1  def starts_one_ends(n):
2      if n == 1:
3          return 2
4      else:
5          return 2 * 9 * 10 ** (n - 2) + 10 ** (n - 1)
```

Test (AssertionError)

```
6  assert starts_one_ends(2) == 18
7  assert starts_one_ends(18) == 180000000000000000
```

Performance - Prompt 2

Keep performance in mind while solving the following code generation task: {Task}

```
1  def starts_one_ends(n):
2      if n == 1:
3          return 1
4      else:
5          count = 2 * 9 * 10 ** (n - 2)
6          return count
```

Test (Pass)

```
7  assert starts_one_ends(2) == 18
8  assert starts_one_ends(18) == 180000000000000000
```

Figure 3.1: Simplified examples of Generated Code: One Without Performance Considerations, and Two With Performance Considerations Using Different Prompts.

a *Timeout* for large loop sizes. Moreover, even when performance is considered, *Prompt 1* misleads the LLM into generating code that improves speed but raises an *AssertionError*, whereas *Prompt 2* achieves both correct functionality and enhanced performance. This motivates the necessity of systematically evaluating LLMs' NFR-aware code generation capabilities—such as by examining how prompt variations affect the resulting code.

In this research, we introduce *RoboNFR*, an automated framework that evaluates the robustness of LLMs in code generation while incorporating NFRs (i.e., NFR-aware code generation). *RoboNFR* examines four commonly targeted dimensions of NFRs (Rasheed et al., 2024) and considers three methodologies for evaluating LLMs in NFR-aware code generation. Specifically, *RoboNFR* examines NFR dimensions—including *Code Design*, *Reliability*, *Readability*, and *Performance*—using associated metrics to evaluate the generated code. While considering these NFRs, *RoboNFR* introduces the following methodology: 1) ***Prompt Variations***—to examine how generated code correctness and NFR quality change when users apply different prompts targeting specific NFR dimensions; 2) ***Regression Testing***—to investigate how an LLM's NFR-aware code generation capabilities evolve after updates to the same model; and 3) ***NFR-Aware Code Generation Workflows***—to explore how different workflows for applying LLMs to address NFRs affect the resulting code.

Through these NFR dimensions and methodologies, *RoboNFR* is able to reveal potential robustness issues in LLMs when generating code with NFR awareness.

We apply *RoboNFR* to three popular LLM families (Minaee et al., 2024; W. X. Zhao et al., 2025) in our study: *GPT-3.5-turbo* and *GPT-4o* from OpenAI, and *Claude-3.5* from Anthropic. Using four widely adopted coding benchmarks (Huang, Bu, Zhang, et al., 2023; Lin, Kim, Tse-Husn, & Chen, 2024)—*HumanEval*, *HumanEval-ET*, *MBPP*, and *MBPP-ET*—we investigate the robustness of each model using *RoboNFR*'s evaluation methodology. *RoboNFR* evaluates the robustness of LLMs' NFR-aware code generation capabilities by analyzing both functional correctness and NFR-related metrics (i.e., code smells, readability, exception handling, and execution time), along with their average values and STandard DEViations (STDEV). Our experimental results demonstrate that *RoboNFR* effectively uncovers robustness issues in the tested LLMs through each methodology. While code generation without considering NFRs (i.e., *RawGPT*) achieves stable Pass@1 scores (with an average STDEV of 0.48) across prompt variations, using different prompts to guide LLMs in addressing NFRs can reduce Pass@1 scores by up to 39% compared to *RawGPT*. It also results in a significantly higher STDEV of 2.48, indicating greater variation and reduced robustness. Incorporating NFRs in the prompt generally helps improve NFR quality metrics. However, NFR metrics, especially related to readability and exception-handling, are more sensitive to prompt variation, where they have higher STDEV across prompts.

We found that different versions of the LLM may introduce some trade-offs between code correctness and NFR metrics. For example, the newer version of *GPT-4o* reduces code correctness in favor of improved performance (shorter execution time). However, not all LLMs exhibit clear trade-off patterns. For instance, *GPT-3.5-turbo* shows completely opposite trends across the two benchmarks, *HumanEval* and *MBPP* compared to *GPT-4o*. This inconsistency may make it difficult for users to understand the model's NFR-aware code generation capability or for developers to identify areas for improvement, highlighting potential robustness issues in LLMs. We also found robustness issues when changing the interaction workflows with the LLMs. Integrating NFRs and functional requirements into a single prompt (i.e., *NFR-Integrated*) generally produces code with higher correctness compared to asking the LLM to enhance *RawGPT*-generated code with the same NFRs (i.e., *NFR-Enhanced*). Nevertheless, *NFR-Enhanced* performs better in reducing code smells

and improving readability, while *NFR-Integrated* is more effective at adding exception-handling statements and optimizing execution time. Overall, these findings suggest that robustness issues exist in the NFR-aware code generation capabilities of LLMs, highlighting the need for carefully selecting suitable prompts, model versions, and well-designed workflows during development.

We summarize our contributions as follows:

- We propose *RoboNFR*, a novel framework for systematically evaluating the robustness of LLMs in addressing Non-Functional Requirements (NFRs) during code generation.
- We reveals potential robustness issues within popular LLM families (e.g., *GPT* and *Claude*). Their ability to generate NFR-aware code is significantly affected by prompt variations, model updates, and different workflows, impacting both code correctness and NFR metrics.
- Our comprehensive experiments emphasize the importance of establishing continuous quality assurance when integrating LLMs into real-world development. Frameworks like *RoboNFR* can be used to monitor the robustness of deployed LLMs in real-world scenarios, preventing unexpected changes in LLM-based product behavior.
- We provide the replication package and data to support reproduction and future studies (A. Anonymous, 2024).

Research Organization. Section 3.3 reviews related work. Section 3.4 describes how *RoboNFR* works. Section 3.5 presents our experiment with *RoboNFR*. Section 3.6 discusses the key findings and their implications. Section 3.7 outlines potential threats to validity. Finally, Section 3.8 concludes the study.

3.3 Related Work

Non-Functional Requirements (NFRs) in Coding. Prior studies proposed approaches for examining or refining existing source code to meet the NFRs. Pereira dos Reis, Brito e Abreu, de Figueiredo Carneiro, and Anslow (2022) summarized a series of studies on detecting and visualizing code smells that negatively impact code design. Vitale, Piantadosi, Scalabrino, and Oliveto (2023) trained models to improve the readability of given code snippets and Z. Li et al. (2023) identified readability issues from logging code. J. Zhang et al. (2020) proposed an automated approach

to generate exception handling code based on existing source code to improve the overall software reliability. Biringa and Kul (2023) proposed a program analysis framework that provides continuous feedback on the performance impact of pending code updates, while Q. Zhao, Chabbi, and Liu (2023) developed tools to help developers identify and resolve performance inefficiencies. All these studies highlight the importance of NFRs in real-world software development; however, they are limited to study only specific types of NFR tasks and a narrow set of metrics. In contrast, our work investigates how code quality changes across four NFR dimensions using multiple metrics, offering a more comprehensive aspects for studying code quality.

Exploring NFRs in Code Generation with LLMs. Several studies have explored the potential of LLMs in addressing NFRs to improve code quality across various dimensions. For instance, Wu et al. (2024) integrated LLMs with traditional code smell detection tools to automatically reduce code smells. Y. Xu et al. (2025) leveraged LLMs to enhance code readability through automated refactoring. Han, Kim, Yoo, Lee, and won Hwang (2024) incorporated software requirements from textual descriptions to enable NFR-aware code generation, improving aspects such as reliability. Gao, Gao, Gu, and Lyu (2024) utilized LLMs to optimize the execution efficiency of source code. Unlike prior studies that focus on specific NFR-related tasks or refining existing code, our work emphasizes the inherent capabilities of LLMs in generating NFR-aware code. In addition, recent research has pointed out that addressing NFRs may negatively impact the functional correctness of generated code. For example, Singhal, Aggarwal, Awasthi, Natarajan, and Kanade (2024) proposed a new benchmark to evaluate LLM in NFR-aware code generation and found that current models often struggle with such tasks. Waghjale et al. (2024) studied how to improve code execution speed while preserving functional correctness. While these works focus on identifying or addressing NFR-related issues in LLM-generated code, our study examines how the NFR-aware code generation capabilities of LLMs vary across different usage scenarios and exploring the associated robustness challenges.

Studying the Robustness of LLMs in Code Generation. Wang et al. (2022), J. Chen, Li, Hu, and Xia (2024), and Shirafuji et al. (2023) explored robustness by perturbating different components in the prompts (e.g., problem descriptions, docstrings) with diverse patterns. L. Chen, Zaharia, and

Zou (2023b) and Lin et al. (2024) reported that ChatGPT's performance on code generation can change substantially between different versions of the same model. Mishra et al. (2024) examined how robustness varies across various models and model sizes. These studies primarily focused on the functional correctness of the generated code. Given the critical role of NFRs in software development, our study addresses the importance of exploring the impact of incorporating NFR considerations into various coding workflows for LLM-based code generation. We also study the stability on the functional and NFR code quality across semantically equivalent prompts and model versions. 2

3.4 Methodology

In this research, we propose *RoboNFR*, an automated framework that evaluates the robustness of LLMs when addressing Non-Functional Requirements (NFRs) in code generation. (i.e., *NFR-aware code generation*). In this section, we provide details of *RoboNFR*, including the four NFR dimensions along with their associated metrics, as well as the three complementary methodologies it incorporates.

3.4.1 Overview of *RoboNFR*

NFRs, such as maintainability and readability, are critical aspects of code quality. However, existing studies on code generation often overlook NFRs and only focus on functional correctness metrics. For example, Pass@1 (L. Chen et al., 2023b; M. Chen et al., 2021) is commonly used to assess whether the code generated by the LLM passes all test cases on its first attempt. However, without considering NFRs, the generated code might be only functionally correct but lack reliability, readability, or efficiency. Such neglect can lead to significant maintenance challenges and impact software quality (Chung & do Prado Leite, 2009). When users leverage LLMs to address NFRs, they may do so in different ways—for example, by using varied prompts to express the same NFRs, by asking LLM to generate NFR-aware code, or by asking the LLM to improve existing code while considering NFRs. These different usage patterns can lead to different results (J. Chen et al., 2024; Shirafuji et al., 2023; Wang et al., 2022). A robust LLM should produce consistent outputs when

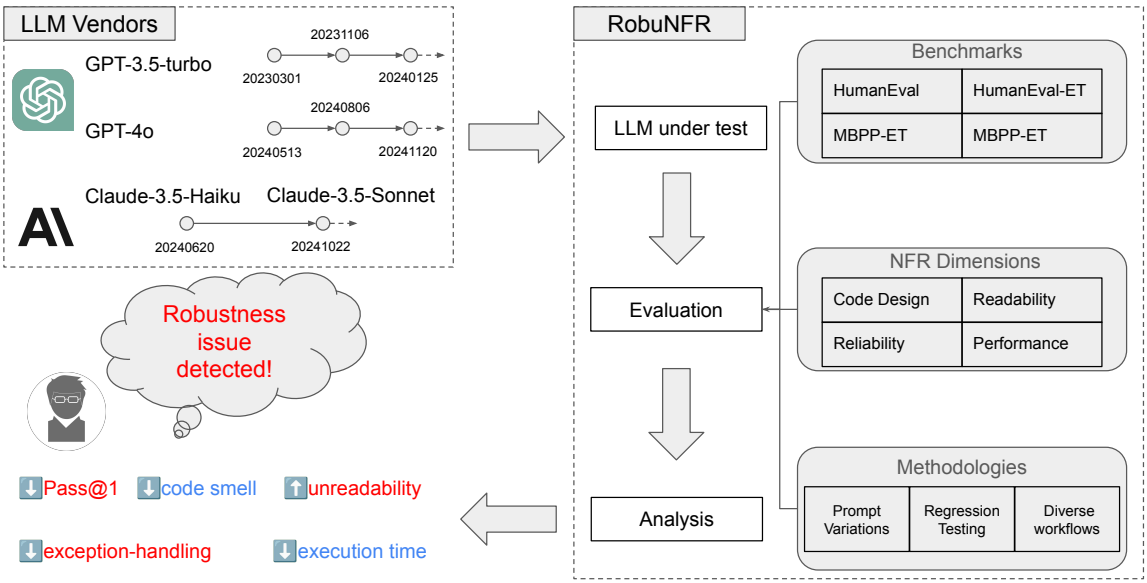

Figure 3.2: Overview of *RoboNFR*. *RoboNFR* leverages three methodologies to evaluate the code generation capabilities of LLM across NFR dimensions using various code benchmarks, aiming to reveal potential robustness issues in the LLM under test.

addressing the same NFRs, regardless of how the request is phrased or applied. However, measuring the robustness of LLMs in this context remains a challenging task. Hence, *RoboNFR* is designed to study the capability and robustness of LLMs in addressing NFRs during the code generation process.

As shown in Figure 3.2, for the given LLM, *RoboNFR* integrates existing coding benchmarks (e.g., HumanEval) with a set of specified NFR dimensions (e.g., Code Design) and employs three distinct evaluation approaches to generate comprehensive evaluation results. Specifically, our framework adopts three evaluation methodologies: 1) ***Prompt Variations***, assessing whether the functional and non-functional quality of LLM-generated code varies when using different prompts that convey the same NFRs; 2) ***Regression Testing***, evaluating differences in the NFR-aware code generation capabilities of an LLM after updates to the same model; and 3) ***NFR-Aware Code Generation Workflows***, analyzing the impact on how LLMs generate NFR-aware code when NFRs are addressed either through a one-shot prompt or sequentially. During the evaluation process, *RoboNFR* provides a comprehensive set of metrics to assess both the functional correctness and code quality of the generated code that addresses NFRs. These metrics capture changes in LLM-generated code during evaluation, thereby highlighting potential robustness issues in NFR-aware code generation.

3.4.2 Studied NFR Dimensions and Metrics

In addition to functional correctness, *RoboNFR* evaluates four NFR dimensions that contribute to a code's maintainability, reliability, and efficiency. Below, we describe functional correctness and each NFR dimension, along with the corresponding metric used for evaluation.

Functional Correctness refers to whether the generated code satisfies the intended functionality. We use Pass@1 (M. Chen et al., 2021) to determine whether the code passes all test cases on its first attempt.

Code design refers to the structural and architectural quality of code, where bad designs can significantly hinder maintainability and scalability (Fowler, 2018; Walter & Alkhaeir, 2016). We use the *Refactor* checker of Pylint (PyCQA, 2024b) to detect code smells. It includes predefined static code checkers to detect various code smells. We calculate and report the code smell density as the number of detected smells per 10 Lines Of Code (LOC) since the generated code may have different lengths.

Reliability is the code's ability to handle unexpected inputs and ensure stable execution under various scenarios (e.g., exception handling) (Pham, 2000; J. Zhang et al., 2020). In particular, we measure whether the generated code includes exception-handling mechanisms, such as try-catch blocks, to gain insight into how well the code anticipates and manages potential errors. We calculate exception density as the number of exception-handling statements per 10 lines of code, as this metric highlights the extent of error-handling logic (De Padua & Shang, 2017).

Readability is how easily code can be understood and modified. Readable code should follow coding style guidelines and conventions to ease understanding and collaboration (Piantadosi, Fierro, Scalabrino, Serebrenik, & Oliveto, 2020). Similar to code design, we use use the *Convention* checker of Pylint (PyCQA, 2024a) to detect issues like inconsistent naming, incorrect indentation, and missing comments. We also report the density of readability issues per 10 lines of code.

Performance assesses the efficiency of code, where performance issues (e.g., slower execution) can cause higher operational costs and reduce user satisfaction (Malik, Hemmati, & Hassan, 2013). We measure the execution time in milliseconds for all tests associated with each coding problem. To minimize measurement fluctuations, we run each test case five times and calculate the mean.

Table 3.1: LLM generated prompt templates to consider non-functional requirements in code generation.

Error-handle	Code Smell	Readability	Performance
Incorporate various *error handling* techniques	Investigate various strategies to handle *code smell*	Evaluate different coding practices for *readability*	Optimize for *performance*
Implement multiple *exception handling* strategies	Minimize *code smell*	Investigate various techniques to enhance *readability*	Focus on enhancing *performance*
Apply different *error handling* mechanisms	Eliminate *code smell*	Improve the code *readability*	Ensure the code *runs efficiently*
Investigate different methods of *managing exceptions*	Identify and address different *code smells*	Ensure the code is *readable*	Prioritize *runtime optimization*
Integrate diverse *error handling* approaches	Apply best practices to reduce *code smell*	Apply coding practices that enhance *readability*	Keep *performance* in mind while solving
Utilize multiple *error management* techniques	Mitigate *code smell*	Focus on *readability*	Aim for *high-performance* execution
Experiment with various ways to *handle exceptions*	Tackle different *code smell* issues	Enhance the *readability* of the code	Reduce *computational overhead*
Combine different *error handling* practices	Implement techniques to prevent *code smell*	Implement strategies to make the code more *readable*	Emphasize *speed and efficiency*
Evaluate multiple *exception management* strategies	Resolve *code smell* problems	Optimize the code for better *readability*	Ensure minimal *resource consumption*
Develop a range of *error handling* solutions	Optimize code to avoid *code smell*	Adopt coding practices for improved *readability*	Maximize *performance* in your solution

3.4.3 Evaluation Methodology 1: Prompt Variations

Prior research (J. Chen et al., 2024; Shirafuji et al., 2023; Wang et al., 2022) suggests that variations in prompt templates, even when preserving semantic context, can generate significantly different code. Hence, we repeat the code generation process using different but semantically equivalent prompts. To mitigate potential biases introduced by manually altering the prompts, we leverage GPT-4o-mini to generate various prompts for each dimension of NFRs while preserving the same semantics. This approach allows us to evaluate the stability of the results by measuring variations across different prompt templates, thereby clarifying how changes in prompts influence the code generated by LLMs. In other words, if the results remain stable, it indicates that the LLM is robust to variations in prompts.

Constructing Diverse NFR-Aware Prompts. Table 3.1 shows the semantically equivalent prompt templates generated for each dimension of NFRs. Initially, we manually crafted a seed prompt with the structure: "*Consider [NFR] and complete the following code*", where "*[NFR]*" corresponds to specific non-functional requirements, such as code design or readability. We then provided the seed prompt to ChatGPT to generate 10 semantically equivalent prompts for the experiment. We incorporate all 10 prompt variants to assess the robustness of the LLM against semantic preserving changes in the prompt. In total, we execute NFR-aware code generation 40 times (10 variations per NFR) for each workflow and each LLM version. Although our experiments are conducted on existing code generation benchmarks (e.g., HumanEval and MBPP), *RoboNFR* is highly adaptable. Future studies can easily tailor its process to accommodate new NFRs and additional benchmarks.

3.4.4 Evaluation Methodology 2: Regression Testing

Previous research has shown that robustness issues exist in LLMs during version updates (L. Chen et al., 2023b; Lin et al., 2024). Although developers may claim that an LLM's code generation ability remains consistent after an update, the actual results often vary depending on the coding benchmark used. This discrepancy becomes even more pronounced in NFR-aware code generation tasks, where robustness issues may be more deeply concealed.

To address this challenge, *RoboNFR* leverages the concept of regression testing to monitor robustness issues, specifically in NFR-aware code generation tasks. In particular, *RoboNFR* incorporates several key components of regression testing:

Fixed Test Suite and Metrics: *RoboNFR* evaluates both the older and newer versions using the same test suites (e.g., code generation benchmark) and metrics, guaranteeing a fair comparison.

Baseline Establishment: We define both the *RawGPT* (i.e., requirements contain only functional features without any additional NFRs) and the older model version as baselines, enabling a direct comparison of their behaviors.

Impact Analysis: By maintaining consistent evaluation criteria, *RoboNFR* tracks trends in NFR-aware code generation and quantifies the extent of change.

RoboNFR employs regression testing to reliably identify robustness issues during model updates. In other words, if a newer version of an LLM demonstrates consistent NFR-aware code generation capabilities across all regression test cases, it can be considered robust with respect to version changes.

3.4.5 Evaluation Methodology 3: NFR-Aware Code Generation Workflows

In addition to generating functional code, modern code generation tools, such as Cursor (Cursor, 2024) and GitHub Copilot (Copilot, 2024a), provide two typical workflows for NFR-aware code generation (Copilot, 2024b). 1) ***NFR-integrated code generation*** involves developers providing both the functional and non-functional requirements in one prompt to generate the complete code in one shot. 2) ***NFR-enhanced code refinement*** refers to the process by which developers

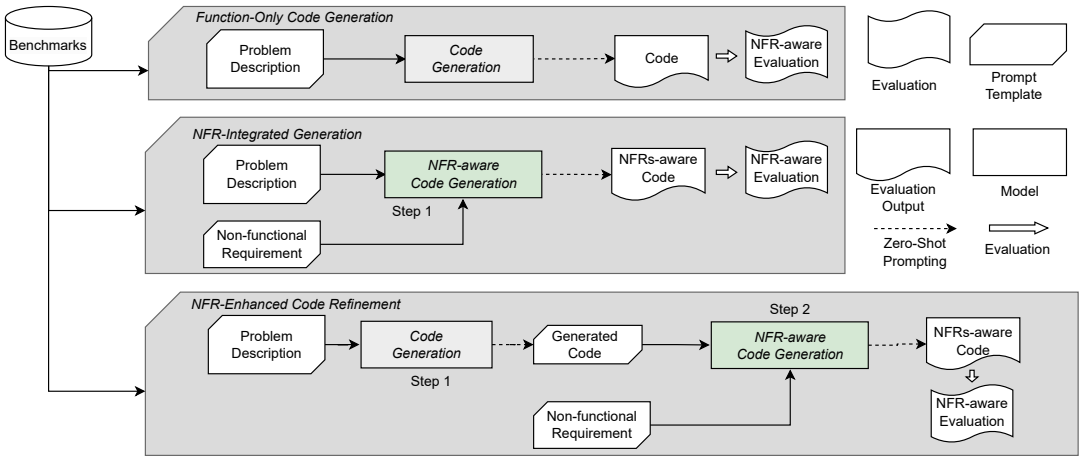

Figure 3.3: *RoboNFR* defines three workflows as part of its NFR-aware code generation evaluation methodology. These workflows include Function-Only code generation, NFR-Integrated code generation, and NFR-Enhanced code refinement. We compare the functional and non-functional quality of the generated code across these workflows.

➢Function-Only Code Generation
Complete the following code.
Input: '{Problem Description}'
Response: '{Code}'

➢NFR-Integrated Code Generation
Given the problem description, generate code by considering {NFR}."
Input: '{Problem Description}'
Response: '{NFR-aware Code}'

➢NFR-Enhanced Code Refinement
Step 1 - Existing code to be refined -> '{Code}'
Step 2 - Refine the code with NFR
Given the following code, your goal is to improve its {NFR}."
Input: '{Code}'
Response: '{NFR-aware Code}'

Figure 3.4: A simplified example of a prompt template for NFR-Aware code generation workflows.

use an LLM to refine existing code, thereby enhancing code quality and better aligning it with specific requirements (White, Hays, Fu, Spencer-Smith, & Schmidt, 2024). Figure 3.3 provides an overview of these workflows. The baseline workflow considers only the functional requirement (i.e., *Functional-Only Code Generation*, denoted as *Functional*, and two NFR-aware code generation workflows (i.e., *NFR-Integrated* and *NFR-Enhanced*).

While both workflows provide the instruction to generate code that satisfies specific requirements, the final output may be different, as the ways of interacting with LLMs may significantly affect the generated results (Lee, Jang, Jang, Lee, & Yu, 2024). Therefore, in *RoboNFR*, we consider both of these two workflows to incorporate the four NFRs into the code. In the remainder of the research, we denote NFR-integrated code generation as *NFR-Integrated* and NFR-aware code enhancement as *NFR-Enhanced* for conciseness. To analyze the results, we compare the functional and non-functional code quality metrics across the code generated by three distinct workflows, examining the impact of NFR-aware code generation on overall code quality.

Figure 3.4 shows the prompt templates for each workflow. *Functional* only contains the functional requirement in the prompt. *NFR-Integrated* incorporates NFRs directly into the prompt template. For example, when considering reliability, the prompt asks the LLM to generate code that meets the functional requirements and optimize reliability in a single prompt. *NFR-Enhanced* adopts a two-step process. It leverages the code generated by *Functional*, and it sends a separate prompt asking the LLM to enhance the code by addressing a specific NFR. Both *NFR-Integrated* and *NFR-Enhanced* use the same NFR-aware prompt templates outlined in Table 3.1.

3.5 Evaluation

Although *RoboNFR* is not limited to specific benchmarks or LLMs under evaluation, in our study, we selected several popular models and coding benchmarks to demonstrate how *RoboNFR* can reveal robustness issues in the NFR-aware code generation capabilities of LLMs.

Studied LLMs. We selected two widely recognized LLM families mentioned in recent LLM-focused surveys (Minaee et al., 2024; W. X. Zhao et al., 2025). Specifically, we evaluated *GPT-3.5-turbo* and *GPT-4o* from OpenAI, as well as *Claude-3.5-Sonnet* and *Claude-3.5-Haiku* from Anthropic, as these models allow for a comparison of different LLMs within the same family, sharing similar architectures and development processes. To support regression testing, we included two released versions for each LLM (e.g., *gpt-3.5-turbo-1106* and *gpt-3.5-turbo-0125* for *GPT-3.5-turbo*). All interactions with the models were conducted via vendor-provided APIs. To minimize output variance and ensure deterministic responses, we set the temperature parameter to 0.

Benchmark Datasets. *RoboNFR* can be applied to any code generation benchmark across different programming languages. In our study, we selected four commonly used datasets: HumanEval, HumanEval-ET, MBPP (Mostly Basic Python Programming), and MBPP-ET. These benchmarks are widely adopted in code generation research (Huang, Bu, Zhang, et al., 2023; Lin et al., 2024) and include test cases for evaluating the functional correctness of generated code. HumanEval (M. Chen et al., 2021) comprises 164 programming problems, while MBPP (Austin et al., 2021) includes 427 programming problems (we used the sanitized version provided by the original authors). Furthermore, HumanEval-ET and MBPP-ET, published by Dong, Ding, et al. (2023), use the same problems as HumanEval and MBPP but offer more test cases with approximately 100 test cases for each problem.

Environment. Our experiments were conducted on a Mac Mini (Apple M4, 10 cores, 16GB RAM), using Python 3.9.19 to implement *RoboNFR* and the evaluation scripts. The OpenAI API library used was version 1.14.3, and the Claude API library used was version 0.39.0. For detecting code smells and readability issues, we used Pylint version 3.2.5. We have made our framework code and evaluation data publicly available to support future research (A. Anonymous, 2024).

RQ1: How do variations in prompts affect the robustness of LLMs in NFR-aware code generation?

Motivation. Different users may express the same NFRs using diverse prompts when instructing an LLM to generate code. This research question explores how variations in prompt wording influence the LLMs in producing code that adheres to NFRs and functional correctness.

Approach. We adopt Evaluation Methodology 1 (i.e., Prompt Variations) as discussed in Section 3.4.3. We compute the Pass@1 and NFR metrics for generating four types of NFR-aware code—design, reliability, readability, and performance—using different prompts. Pass@1 measures the functional correctness of the code by evaluating whether the first generated solution successfully passes all the provided tests (M. Chen et al., 2021). A Pass@1 of 100 means the first generated code can pass all the tests in 100% of the coding problems in a benchmark. Additionally, we compare the changes in average (AVG) and standard deviation (STDEV) relative to our baseline (*RawGPT*), where code is generated considering only functional requirements. We then extend our study across various

models and four benchmarks, as discussed in Section 3.5.

Results. ***Integrating NFRs almost always reduces Pass@1 scores across all models and benchmarks by up to 39%. Notably, some models appear more susceptible to the challenges posed by NFR dimensions.*** Table 3.2 presents the results for the baseline (*RawGPT*) and all four NFR dimensions using different prompts that convey the same meaning. The Pass@1 scores for the NFR dimensions are mostly lower than that of the baseline, particularly in the Reliability dimension, where the Claude-3.5-Haiku model shows a 39% reduction in Pass@1. Additionally, we observed that the average Pass@1 score dropped by 9.76% between the Claude-3.5-Sonnet and Claude-3.5-Haiku models, compared to a 3.45% drop between GPT-3.5-turbo and GPT-4o. This type of comparison—between two generations of the Claude family and two generations of the GPT family—highlights potential directions for improving model architectures and training processes. Our results show that models from the Claude family tend to be more adversely affected by the additional challenges introduced by the NFRs.

Compared to the baseline (RawGPT), all models exhibit consistently higher STDEV values in Pass@1 when handling NFR dimensions, indicating increased robustness issues. As shown in Table 3.2, the STDEV for *RawGPT*'s Pass@1 is generally lower than that for NFR dimensions across all models and benchmarks. For example, the STDEV for *RawGPT*'s Pass@1 is between 0.00 to 1.16 across all the models and benchmarks, while the STDEV, when considering Code Design, is between 0.77 to 5.47. This suggests that when users employ different wordings in prompts to convey the same meaning in NFR dimensions, the Pass@1 of the LLMs varies more than in *RawGPT* dimensions. ***Additionally, our analysis indicates that different NFR dimensions exhibit varying levels of robustness issues.*** In particular, the Reliability dimension presents the largest variance in certain models. For instance, as shown in Table 3.2, GPT-3.5-turbo reaches a peak STDEV of 13.93 across all four NFR dimensions, while Claude-3.5-Haiku achieves a peak STDEV of 4.21. These results suggest that certain prompts can significantly impact code correctness for these models, indicating that less robust models may be more sensitive to variations in Pass@1 scores under the reliability dimension.

Incorporating NFRs into prompts generally improves code quality by reducing code smells and enhancing readability. As shown in Table 3.3, incorporating NFRs generally reduces code

Table 3.2: The Pass@1 column represents the Pass@1 scores, along with their STDEV across 10 semantically equivalent prompts. Δ indicates the percentage difference in Pass@1 of the same model version between the NFR-aware results and the Function-Only result.

NFR Dimension	Model	HumanEval			HumanEval-ET			MBPP			MBPP-ET		
		Pass@1	STDEV	Δ(%)	Pass@1	STDEV	Δ(%)	Pass@1	STDEV	Δ(%)	Pass@1	STDEV	Δ(%)
Function-Only	GPT-3.5-turbo	72.50	0.73	-	64.33	1.05	-	67.82	0.48	-	47.21	0.39	-
	GPT-4o	90.55	1.16	-	80.18	0.96	-	74.43	0.55	-	53.37	0.34	-
	Claude-3.5-Sonnet	89.39	0.33	-	78.54	0.51	-	75.97	0.27	-	54.94	0.13	-
	Claude-3.5-Haiku	86.22	0.33	-	75.61	0.43	-	72.37	0.00	-	52.93	0.00	-
Code Design	GPT-3.5-turbo	72.87	1.82	0.51	64.51	2.15	0.28	67.68	1.40	↓ 0.21	47.61	1.08	0.85
	GPT-4o	89.33	0.77	↓ 1.35	79.63	1.08	↓ 0.69	73.37	1.12	↓ 1.42	52.58	1.10	↓ 1.48
	Claude-3.5-Sonnet	84.02	1.85	↓ 6.01	72.56	1.93	↓ 7.61	70.87	2.58	↓ 6.71	49.79	2.23	↓ 9.37
	Claude-3.5-Haiku	80.73	2.42	↓ 6.37	70.12	2.62	↓ 7.26	64.73	5.47	↓ 10.56	45.67	3.93	↓ 13.72
Readability	GPT-3.5-turbo	73.17	2.80	0.92	64.33	1.91	0.00	68.76	1.51	1.39	48.41	1.19	2.54
	GPT-4o	91.40	1.64	0.94	80.98	1.60	1.00	75.04	0.87	0.82	53.63	0.89	0.49
	Claude-3.5-Sonnet	86.46	1.80	↓ 3.28	75.85	1.81	↓ 3.43	73.35	2.64	↓ 3.45	51.66	1.52	↓ 5.97
	Claude-3.5-Haiku	84.02	3.41	↓ 2.55	73.78	3.26	↓ 2.42	61.55	7.79	↓ 14.95	44.31	5.41	↓ 16.29
Reliability	GPT-3.5-turbo	68.29	3.50	↓ 5.81	59.09	3.62	↓ 8.15	42.93	13.93	↓ 36.70	29.46	9.60	↓ 37.6
	GPT-4o	88.29	1.25	↓ 2.50	76.22	1.52	↓ 4.94	71.59	0.88	↓ 3.82	50.02	0.70	↓ 6.28
	Claude-3.5-Sonnet	81.83	2.64	↓ 8.46	70.12	2.96	↓ 10.72	69.32	1.81	↓ 8.75	46.79	1.96	↓ 14.83
	Claude-3.5-Haiku	73.05	2.26	↓ 15.27	62.20	2.02	↓ 17.74	47.45	4.21	↓ 34.43	32.04	2.68	↓ 39.47
Performance	GPT-3.5-turbo	70.79	3.45	↓ 2.36	61.83	2.93	↓ 3.89	66.63	1.92	↓ 1.75	47.82	1.47	1.29
	GPT-4o	89.33	2.12	↓ 1.35	80.18	1.61	0.00	74.07	0.72	↓ 0.48	53.56	0.80	0.36
	Claude-3.5-Sonnet	83.29	1.53	↓ 6.82	74.02	1.40	↓ 5.76	72.04	1.76	↓ 5.17	51.43	2.08	↓ 6.39
	Claude-3.5-Haiku	81.32	1.98	↓ 4.81	71.95	0.96	↓ 4.84	70.73	2.36	↓ 2.27	49.41	2.39	↓ 6.65

smells—achieving an average reduction of 34.93% compared to *RawGPT* —and enhance readability, with an average improvement of 24.32% across all models and benchmarks. We also observed that the extent of these improvements varies across models. ***Claude models consistently generating code with fewer smells and better readability compared to GPT models.*** For instance, in the Code Design dimension, Claude-3.5-Sonnet and Claude-3.5-Haiku achieved code smell density values of 0.02 and 0.01, compared to GPT-3.5-turbo and GPT-4o at 0.22 and 0.06. Although all models improve over the baseline, Claude models show a more pronounced reduction in code smell—a trend consistent across all NFR dimensions and benchmarks, especially in Readability dimension. However, differences in the Reliability and Performance dimension are less distinct.

Incorporating NFRs into prompts results in higher STDEV values for the code smell, unreadability, and exception-handling metrics, suggesting that different prompts introduce greater variability in the NFR-aware code generation capabilities of LLMs. Table 3.3 shows that the

Table 3.3: Columns *code smell density*, *unreadability density*, *exception-handling density*, and *execution time (millisecond)* represent the NFR metrics (Section 3.4.2). Each metric includes standard deviations and $\Delta\%$, which indicates the percentage difference between NFR-aware results and *RawGPT* results.

NFR Dimension	Model	HumanEval				MBPP			
		code smell(Δ%)	unreadability(Δ%)	exception-handling(Δ%)	execution time(Δ%)	code smell(Δ%)	unreadability(Δ%)	exception-handling(Δ%)	execution time(Δ%)
Function-Only	GPT-3.5-turbo	0.31±0.01	3.42±0.04	0.036±0.003	112.63±48.09	0.27±0.01	3.44±0.03	0.011±0.000	43.18±7.96
	GPT-4o	0.12±0.01	2.62±0.03	0.026±0.005	75.92±4.23	0.12±0.00	3.18±0.02	0.130±0.005	37.23±1.37
	Claude-3.5-Sonnet	0.10±0.01	2.03±0.01	0.037±0.003	57.06±2.93	0.08±0.00	2.67±0.01	0.069±0.002	40.92±3.68
	Claude-3.5-Haiku	0.06±0.00	2.60±0.02	0.022±0.000	70.31±13.75	0.03±0.00	2.69±0.00	0.041±0.000	34.40±0.30
Code Design	GPT-3.5-turbo	0.22±0.03 (↓29.0)	2.68±0.20 (↓21.6)	0.051±0.038 (41.7)	92.93±32.03 (↓17.49)	0.28±0.03 (3.7)	4.68±0.48 (36.0)	0.117±0.106 (963.6)	58.78±11.12 (36.13)
	GPT-4o	0.06±0.00 (↓50.0)	1.27±0.15 (↓51.5)	0.091±0.018 (250.0)	81.22±6.32 (6.98)	0.05±0.00 (↓58.3)	1.79±0.08 (↓43.7)	0.363±0.029 (179.2)	38.30±3.03 (2.87)
	Claude-3.5-Sonnet	0.02±0.01 (↓80.0)	0.79±0.10 (↓61.1)	0.148±0.064 (300.0)	47.27±15.87 (↓17.16)	0.03±0.01 (↓62.5)	1.66±0.18 (↓37.8)	0.454±0.142 (558.0)	36.42±0.18 (↓11.00)
	Claude-3.5-Haiku	0.02±0.01 (↓66.7)	1.54±0.50 (↓40.8)	0.176±0.118 (700.0)	55.95±1.23 (↓20.42)	0.01±0.01 (↓66.7)	1.95±0.64 (↓27.5)	0.445±0.171 (985.4)	35.18±1.83 (2.27)
Readability	GPT-3.5-turbo	0.18±0.03 (↓41.9)	2.47±0.24 (↓27.8)	0.016±0.005 (↓55.6)	120.41±49.29 (6.91)	0.24±0.02 (↓11.1)	4.14±0.49 (20.3)	0.013±0.006 (18.2)	49.45±10.22 (14.52)
	GPT-4o	0.06±0.01 (↓50.0)	1.25±0.07 (↓52.3)	0.029±0.007 (11.5)	84.49±5.31 (11.29)	0.07±0.01 (↓41.7)	1.86±0.19 (↓41.5)	0.107±0.033 (↓17.7)	36.39±2.33 (↓2.26)
	Claude-3.5-Sonnet	0.04±0.02 (↓60.0)	0.97±0.20 (↓52.2)	0.084±0.042 (127.0)	44.04±13.54 (↓22.82)	0.05±0.02 (↓37.5)	1.54±0.24 (↓42.3)	0.261±0.097 (278.3)	39.04±8.94 (↓4.59)
	Claude-3.5-Haiku	0.02±0.01 (↓66.7)	1.38±0.20 (↓46.9)	0.088±0.040 (300.0)	61.56±10.61 (↓12.44)	0.02±0.01 (↓33.3)	1.61±0.10 (↓40.1)	0.226±0.086 (451.2)	35.33±1.56 (2.70)
Reliability	GPT-3.5-turbo	0.34±0.10 (9.7)	2.81±0.40 (↓17.8)	1.342±0.247 (3627.8)	117.86±41.17 (4.64)	0.36±0.07 (33.3)	3.25±0.62 (↓5.5)	1.601±0.222 (14454.5)	40.96±0.67 (↓5.14)
	GPT-4o	0.10±0.04 (↓16.7)	1.45±0.16 (↓44.7)	0.942±0.157 (3523.1)	90.48±4.98 (19.18)	0.17±0.08 (41.7)	2.61±0.19 (↓17.9)	1.584±0.204 (1118.5)	35.09±0.27 (↓5.75)
	Claude-3.5-Sonnet	0.05±0.01 (↓50.0)	1.07±0.05 (↓47.3)	1.177±0.152 (3081.1)	53.22±7.62 (↓6.73)	0.05±0.01 (↓37.5)	1.98±0.45 (↓25.8)	1.354±0.107 (1862.3)	43.66±15.62 (6.70)
	Claude-3.5-Haiku	0.03±0.01 (↓50.0)	1.76±0.16 (↓32.3)	1.006±0.079 (4472.7)	81.29±41.04 (15.62)	0.01±0.00 (↓66.7)	1.48±0.20 (↓45.0)	1.115±0.075 (2619.5)	34.69±0.43 (↓0.84)
Performance	GPT-3.5-turbo	0.27±0.04 (↓12.9)	3.21±0.22 (↓6.1)	0.016±0.005 (↓55.6)	63.48±33.85 (↓43.64)	0.28±0.06 (3.7)	5.99±0.30 (74.1)	0.011±0.002 (0.0)	51.61±10.93 (19.52)
	GPT-4o	0.08±0.00 (↓33.3)	1.64±0.13 (↓37.4)	0.027±0.010 (3.8)	74.34±1.00 (↓2.08)	0.12±0.02 (0.0)	3.26±0.18 (2.5)	0.103±0.026 (↓20.8)	34.50±0.43 (↓7.33)
	Claude-3.5-Sonnet	0.05±0.02 (↓50.0)	1.67±0.11 (↓17.7)	0.028±0.005 (↓24.3)	35.20±1.72 (↓38.31)	0.05±0.00 (↓37.5)	2.51±0.12 (↓6.0)	0.096±0.047 (39.1)	34.62±0.64 (↓15.40)
	Claude-3.5-Haiku	0.02±0.01 (↓66.7)	2.33±0.08 (↓10.4)	0.033±0.008 (50.0)	98.39±24.43 (39.94)	0.02±0.00 (↓33.3)	2.42±0.11 (↓10.0)	0.070±0.027 (70.7)	37.19±2.53 (8.11)

STDEV values for *RawGPT* across the three NFR metrics (i.e., code smell, unreadability, and exception handling) are generally lower compared to the STDEV values observed in all four NFR dimensions across all models and benchmarks. For instance, when analyzing exception-handling density in HumanEval, GPT-3.5-turbo's STDEV in *RawGPT* is 0.003, which increases to 0.247 for the generated code under the Reliability dimension. Specifically, this substantial increase suggests that varying prompts significantly impact the exception-handling density of the generated code, indicating a robustness issue in LLMs when different prompts are used in the Reliability dimension. However, such a trend was not evident in the execution time metrics. For example, in the HumanEval, Claude-3.5-Haiku has a STDEV of 13.75 in the *RawGPT*, which increases to 24.43 in the Performance dimension—indicating that the generated code execution time varies more when different prompts are used to improve code performance. In contrast, Claude-3.5-Sonnet shows a STDEV of 2.93 in the *RawGPT* setting and a lower value of 1.72 in the Performance dimension, suggesting that this LLM's generated code performance is less affected by prompt variations. This finding suggests that the impact of prompt variations on code performance differs across models, highlighting the importance of using *RoboNFR* to evaluate robustness issues in different LLMs.

Summary of RQ1: Integrating NFRs reduces Pass@1 scores by up to 39% and increases output variability, revealing robustness issues across all models. While incorporating NFRs generally helps reduce code smells and improve readability, it also leads to higher STDEV values, indicating that code quality varies significantly with prompt variations.

RQ2: Are LLMs robust to model updates in their NFR-Aware code generation capabilities?

Motivation. When relying on LLMs for NFR-related tasks, users often need to choose which model version to use. This is especially relevant when using the default version, which automatically switches to the latest one. In such cases, users may experience unexpected changes in outcomes due to model updates. Therefore, this study investigates how updates to LLM versions influence their ability to generate NFR-aware code.

Approach. We adopt Evaluation Methodology 2 (i.e., Regression Testing) as discussed in Section 3.4.4 to compare the results of functional correctness and NFR metrics among different versions of LLMs. We compute the Pass@1 and NFR metrics for four types of NFRs—code design, reliability, readability, and performance—generated by different versions of the same LLM model. We then compare the results within each model. Our study covers three LLM models, and for each, we compare the two most recent versions (at the time of the experiments). Specifically, we used *gpt-3.5-turbo-1106* and *gpt-3.5-turbo-0125* for GPT-3.5-turbo, *gpt-4o-2024-05-13* and *gpt-4o-2024-08-06* for GPT-4o, and *claude-3-5-sonnet-20240620* and *claude-3-5-haiku-20241022* for Claude-3.5. We selected these two versions from the Claude family because they share the same underlying LLM architecture and similar code generation capabilities (Anthropic, 2024), and they were the only ones available to us.

Results. ***When model version updates occur, trade-offs emerge between code correctness and NFR metrics.*** As illustrated in Table 3.4, improvements in one area—such as a reduction in code smell density—are often offset by declines in other metrics, like Pass@1 or ET-Pass@1. For instance, in GPT-3.5-turbo on HumanEval under the *RawGPT*, the pass rate drops by 5.18% and 3.74%, while the code smell metric improves significantly by 18.42%. This pattern suggests that tuning to enhance certain qualitative aspects may come at the expense of Pass@1, and vice versa.

Table 3.4: This table compares various metrics for the same LLM model across different versions. The Pass@1(Δ%) column shows the Pass@1 score for the older version along with the percentage change relative to the newer version, while the ET-Pass@1(Δ%) column presents the same metric for the ET-version dataset. Additionally, the NFR metrics—including code smell density, unreadability density, exception-handling density, and execution time (in milliseconds)—report the older version's results with standard deviations and the corresponding percentage differences compared to the newer version.

Model	Dataset	NFR Dimension	Pass@1(Δ%)	ET-Pass@1(Δ%)	code smell(Δ%)	unreadability(Δ%)	exception-handling(Δ%)	execution time(Δ%)
GPT-3.5 (*20231106*) vs (*20240125*)	HumanEval	Function-Only	76.46±0.77 ↓5.18	66.83±0.51 ↓3.74	0.38±0.01 ↓18.42	2.77±0.04 ↑23.47	0.011±0.003 ↑227.27	110.78±46.55 ↑1.67
		Code Design	72.44±2.71 ↑0.59	64.63±2.80 ↓0.19	0.25±0.01 ↓12.00	1.79±0.10 ↑49.72	0.055±0.058 ↓7.27	97.55±49.53 ↓4.74
		Readability	73.29±3.56 ↓0.16	64.82±2.88 ↓0.76	0.21±0.04 ↓14.29	1.58±0.09 ↑56.33	0.015±0.007 ↑6.67	97.58±46.15 ↑23.39
		Reliability	65.73±4.29 ↑3.89	57.62±4.40 ↑2.55	0.40±0.10 ↓15.00	1.92±0.23 ↑46.35	1.362±0.311 ↓1.47	117.04±54.46 ↑0.70
		Performance	72.26±1.58 ↓2.03	63.54±2.13 ↓2.69	0.32±0.06 ↓15.63	2.41±0.18 ↑33.11	0.014±0.003 ↑14.29	62.87±3.01 ↑0.97
	MBPP	Function-Only	63.47±0.55 ↑6.85	44.75±0.64 ↑5.50	0.32±0.01 ↓15.63	3.64±0.02 ↓5.49	0.006±0.000 ↑83.33	48.84±1.33 ↓11.59
		Code Design	66.53±1.05 ↑1.73	46.49±0.85 ↑2.41	0.35±0.04 ↓20.00	3.71±0.18 ↑26.17	0.126±0.118 ↓7.14	51.66±11.29 ↑13.78
		Readability	66.93±2.38 ↑2.73	47.26±1.60 ↑2.43	0.30±0.03 ↓20.00	3.42±0.28 ↑21.05	0.012±0.005 ↑8.33	52.02±5.54 ↓4.94
		Reliability	45.11±11.71 ↓4.83	30.80±8.21 ↓4.35	0.45±0.12 ↓20.00	2.72±0.54 ↑19.49	1.785±0.212 ↓10.31	42.01±3.51 ↓2.50
		Performance	65.95±2.16 ↑1.03	47.14±1.41 ↑1.44	0.32±0.05 ↓12.50	5.18±0.22 ↑15.64	0.011±0.003 (0.00)	47.05±6.29 ↑9.70
GPT-4o (*20240513*) vs (*20240806*)	HumanEval	Function-Only	92.56±0.85 ↓2.17	81.52±1.00 ↓1.64	0.13±0.01 ↓7.69	2.50±0.04 ↑4.80	0.040±0.002 ↓35.00	77.40±13.78 ↓1.91
		Code Design	90.73±1.49 ↓1.54	80.12±1.93 ↓0.61	0.06±0.01 (0.00)	1.35±0.17 ↓5.93	0.095±0.027 ↓4.21	81.57±6.25 ↓0.43
		Readability	92.74±1.30 ↓1.44	81.89±1.52 ↓1.11	0.07±0.01 ↓14.29	1.38±0.15 ↓9.42	0.038±0.012 ↓23.68	81.52±9.24 ↑3.64
		Reliability	89.09±1.92 ↓0.90	76.46±2.23 ↓0.31	0.10±0.03 (0.00)	1.66±0.19 ↓12.65	0.910±0.136 ↑3.52	87.69±1.44 ↑3.18
		Performance	90.18±1.56 ↓0.94	80.73±1.63 ↓0.68	0.07±0.01 ↑14.29	1.38±0.11 ↑18.84	0.023±0.008 ↑17.39	79.44±8.68 ↓6.42
	MBPP	Function-Only	75.34±0.58 ↓1.21	53.91±0.49 ↓1.00	0.10±0.00 ↑20.00	2.72±0.03 ↑16.91	0.129±0.007 ↑0.78	34.69±0.21 ↑7.32
		Code Design	73.79±0.74 ↓0.57	52.95±1.04 ↓0.70	0.06±0.01 ↓16.67	2.23±0.12 ↓19.73	0.363±0.049 (0.00)	38.64±4.47 ↓0.88
		Readability	73.72±1.32 ↑1.79	52.67±0.74 ↑1.82	0.08±0.01 ↓12.50	2.26±0.10 ↓17.70	0.122±0.031 ↓12.30	37.39±2.32 ↓2.67
		Reliability	71.59±0.83 (0.00)	50.35±1.01 ↓0.66	0.18±0.08 ↓5.56	2.75±0.15 ↓5.09	1.588±0.192 ↓0.25	35.44±1.81 ↓0.99
		Performance	73.54±0.47 ↑0.72	52.95±0.67 ↑1.15	0.13±0.02 ↓7.69	3.31±0.19 ↓1.51	0.107±0.039 ↓3.74	36.25±2.48 ↓4.83
Claude-3.5 (*20240620*) vs (*20241022*)	HumanEval	Function-Only	89.39±0.33 ↓3.55	78.54±0.51 ↓3.73	0.10±0.01 ↓40.00	2.03±0.01 ↑28.03	0.037±0.003 ↓40.54	57.06±2.93 ↑23.24
		Code Design	84.02±1.85 ↓3.92	72.56±1.93 ↓3.36	0.02±0.01 (0.00)	0.79±0.10 ↑94.94	0.148±0.064 ↑18.92	47.27±15.87 ↑18.37
		Readability	86.46±1.80 ↓2.82	75.85±1.81 ↓2.73	0.04±0.02 ↓50.00	0.97±0.20 ↑42.27	0.084±0.042 ↑4.76	44.04±13.54 ↑39.82
		Reliability	88.29±1.25 ↓17.26	76.22±1.52 ↓18.39	0.05±0.01 ↓40.00	1.07±0.05 ↑64.49	1.177±0.152 ↓14.53	53.22±7.62 ↑52.75
		Performance	83.29±1.53 ↓2.37	74.02±1.40 ↓2.80	0.05±0.02 ↓60.00	1.67±0.11 ↑39.52	0.028±0.005 ↑17.86	35.20±1.72 ↑179.50
	MBPP	Function-Only	75.97±0.27 ↓4.74	54.94±0.13 ↓3.66	0.08±0.00 ↓62.50	2.67±0.01 ↑0.75	0.069±0.002 ↓40.58	40.92±3.68 ↓15.93
		Code Design	70.87±2.58 ↓8.66	49.79±2.23 ↓8.27	0.03±0.01 ↓66.67	1.66±0.18 ↑17.47	0.454±0.142 ↓1.98	36.42±0.18 ↓3.40
		Readability	73.35±2.64 ↓16.09	51.66±1.52 ↓14.23	0.05±0.02 ↓60.00	1.54±0.24 ↑4.55	0.261±0.097 ↓13.41	39.04±8.94 ↓9.50
		Reliability	71.59±0.88 ↓33.72	50.02±0.70 ↓35.95	0.05±0.01 ↓80.00	1.98±0.45 ↓25.25	1.354±0.107 ↓17.64	43.66±15.62 ↓20.54
		Performance	72.04±1.76 ↓1.82	51.43±2.08 ↓3.93	0.05±0.00 ↓60.00	2.51±0.12 ↓3.59	0.096±0.047 ↓27.08	34.62±0.64 ↑7.42

Interestingly, not every model clearly exhibits trade-off patterns within the same metric groups. ***Some models appear to be less robust, such as GPT-3.5-turbo, which exhibits inconsistent trade-off patterns across benchmarks, indicating greater robustness issues in these LLMs.*** As shown in Table 3.4, the impact of certain model updates is clearly not uniform across NFR dimensions or datasets. For instance, the metric groups Pass@1 and unreadability for newer GPT-3.5-turbo

show inconsistent trade-off patterns. Specifically, on the MBPP dataset, the newer GPT-3.5-turbo shows an improvement in *RawGPT*, with a 6.85% increase in Pass@1 scores and a 5.49% decrease in unreadability density, suggesting improved correctness and code quality compared to its older version. In contrast, on the HumanEval dataset, the newer GPT-3.5-turbo shows a decline in *RawGPT*, with a 5.18% decrease in Pass@1 and a 23.47% increase in unreadability density, indicating that it performs worse in both aspects compared to the older one. This inconsistent trend across datasets highlights potential robustness issues in GPT-3.5-turbo updates, making it more difficult to accurately assess the LLM's NFR-aware code generation capabilities. Notably, some models, like GPT-4o, appear to be more robust, as their trade-off patterns are more consistent. For the same metric group—Pass@1 and unreadability—the newer GPT-4o consistently achieves lower Pass@1 scores and higher unreadability density, as shown in Table 3.4. Although these results indicate that the newer GPT-4o performs worse than its older version, they clearly highlight how its NFR-aware code generation abilities change with model updates. This insight can help users make informed decisions about its usage and assist LLM developers in debugging.

Additionally, GPT-4o exhibits smaller percentage changes overall, indicating more stable NFR-aware code generation capabilities when updated to a new LLM version. For example, when examining Pass@1 for GPT-4o across all NFR dimensions, the percentage change ranges from 0.00% to 2.17%, compared to GPT-3.5, which ranges from 0.59% to 6.85%, and Claude-3.5, which ranges from 1.82% to 33.72%. This indicates that when GPT-4o is updated, it experiences only minor variations on both the HumanEval and MBPP datasets. Compared to GPT-3.5 and Claude-3.5, the results suggests that GPT-4o is more robust to version updates among the models we tested.

As discussed above, GPT-4o appears to be the most robust among the three models, which is evident when examining only the results from the Reliability dimension. This means that ***the Reliability dimension is the most effective in identifying robustness issues in LLMs' NFR-aware code generation capabilities when the model is updated.*** As shown in Table 3.4, Claude-3.5 exhibited significant changes, with a 17.26% decrease in Pass@1 on HumanEval and a 33.72% decrease on MBPP, while GPT-3.5 also showed percentage changes of 3.89% on HumanEval and 4.83% on MBPP. In contrast, GPT-4o displayed minimal changes, with only a 0.90% change on HumanEval

and no change (0.00%) on MBPP. These smaller percentage changes indicate that when the model is updated, GPT-4o is more robust compared to the other two models, aligning with the findings discussed above. A potential reason is that the Reliability dimension requires LLMs to add exception handling, which typically alters the code structure more and introduces additional statements compared to other NFR dimensions.

Overall, these findings highlight that different LLM models exhibit varying changes in NFR-aware code generation capabilities when updated. This underscores the importance of using robustness evaluation frameworks like *RoboNFR* to identify potential issues in LLMs, whether for user adoption or developer debugging.

Summary of RQ2: When LLMs are updated, they introduce trade-offs between code correctness and NFR metrics. Some models are less robust—such as GPT-3.5-turbo exhibits inconsistent trade-offs after an update—while GPT-4o appears more robust, as it maintains more consistent trade-offs and exhibits smaller percentage changes across all metrics. Among the NFR dimensions, the Reliability dimension serves as a strong indicator for monitoring LLM robustness in NFR-aware code generation, as it represents the degree of change between newer and older versions of the same model.

RQ3: Do robustness issues exist in LLMs' NFR-aware code generation when different workflows are applied?

Motivation. As discussed in Section 3.4.5, users primarily rely on *NFR-Integrated* or *NFR-Enhanced* when using LLMs for NFR dimensions. Therefore, this research question investigates the NFR-aware code generation capabilities of the two workflows, comparing them based on both code correctness (measured by Pass@1) and NFRs metrics.

Approach. We adopt Evaluation Methodology 3 (i.e., NFR-Aware Code Generation Workflows) as discussed in Section 3.4.5. For this experiment, we use the same models as in RQ2 (see Section 3.5), namely GPT3.5-1106, GPT3.5-0125, GPT4o-0513, GPT4o-0806, Claude3.5-0620, and Claude3.5-1022. We also use the 10 prompts discussed in RQ1 (see Section 3.5) to compute the average Pass@1 and the standard deviation (STDEV). Finally, we average these values to obtain the final comparison.

Table 3.5: Pass@1 AVG: This column shows the average Pass@1 score computed across all experimental models. Δ: The Δ symbol indicates the Pass@1 difference between the results of various NFR dimensions and the baseline (*RawGPT*) results.

Workflow	NFR Dimension	HumanEval		HumanEval-ET		MBPP		MBPP-ET	
		Pass@1 AVG	Δ(%)	Pass@1 AVG	Δ(%)	Pass@1 AVG	Δ(%)	Pass@1 AVG	Δ(%)
-	*RawGPT*	84.61±0.70	0	74.50±0.74	0	71.57±0.41	0	51.19±0.33	0
NFR-Integrated	Code Design	81.69±1.84	↓3.46	71.93±2.09	↓3.45	69.50±2.06	↓2.89	49.18±1.71	↓3.91
	Readability	83.51±2.42	↓1.30	73.61±2.16	↓1.20	69.89±2.75	↓2.34	49.66±1.89	↓2.99
	Reliability	77.71±2.64	↓8.15	66.95±2.79	↓10.13	58.00±5.56	↓18.96	39.91±4.03	↓22.03
	Performance	81.32±1.98	↓3.89	72.04±1.78	↓3.30	70.49±1.57	↓1.50	50.39±1.47	↓1.56
NFR-Enhanced	Code Design	72.43±7.54	↓14.40	64.02±6.84	↓14.07	53.48±12.12	↓25.27	37.73±8.64	↓26.28
	Readability	76.05±5.96	↓10.12	66.87±5.29	↓10.25	57.41±8.08	↓19.78	40.67±5.51	↓20.55
	Reliability	72.75±4.54	↓14.02	62.09±4.36	↓16.66	61.04±4.08	↓14.71	42.38±2.93	↓17.21
	Performance	77.02±2.70	↓8.97	68.23±2.49	↓8.41	68.53±2.01	↓4.24	48.98±1.67	↓4.30

Results. ***NFR-Integrated almost always achieves better Pass@1 than NFR-Enhanced.*** In Table 3.5, our finding shows that a two-step approach has a negative impact on Pass@1, and the difference can be over 20% (e.g., between *NFR-Integrated* and *NFR-Enhanced* for Code Design in MBPP), depending on the specific NFR and dataset. For code design and readability, the decrease is even more notable in *NFR-Enhanced* (10% to over 20% compared to *RawGPT*) compared to *NFR-Integrated* (1.3% to 3.91% over *RawGPT*). In contrast, even though exception handling (i.e., Reliability) has the largest decrease in *NFR-Integrated*, the difference with *NFR-Enhanced* is smaller. Performance has relatively more stable results between *NFR-Integrated* and *NFR-Enhanced*. As *NFR-Integrated* shows an average 2.56% decrease compared to *RawGPT* in the performance dimension, and *NFR-Enhanced* shows a 6.48% decrease, the gap between them is only 3.92%—much smaller than in other NFR dimensions (e.g., Code Design shows a 16.58% gap between *NFR-Integrated* and *NFR-Enhanced*).

Our findings show that the one-step approach may allow the LLM to balance the objectives better, and ***generative models may perform worse at Pass@1 on a two-step code enhancement, especially if the NFR is more related to re-structuring the code (i.e., code design and readability)***.

Incorporating NFRs reduces the capability of LLMs in stably generating functionally correct code, resulting in more variable Pass@1. *NFR-Integrated* and *NFR-Enhanced* consistently exhibit

Table 3.6: Columns *code smell density*, *unreadability density*, *exception-handling density*, and *execution time (millisecond)* represent the NFR metrics (Section 3.4.2). Each metric includes standard deviations and Δ%, which indicates the percentage difference between NFR-aware results and *RawGPT* results.

Dataset	Workflow	NFR Dimension	code smell(Δ%)	unreadability(Δ%)	exception-handling(Δ%)	execution time(Δ%)
HumanEval	-	Function-Only	0.18±0.01	2.66±0.03	0.029±0.003	84.02±21.55
	NFR-Integrated	Code Design	0.10±0.01 (↓ 44.4)	1.57±0.20 (↓ 41.0)	0.103±0.054 (↑ 255.2)	76.08±18.54 (↓ 9.45)
		Readability	0.10±0.02 (↓ 44.4)	1.51±0.16 (↓ 43.2)	0.045±0.019 (↑ 55.2)	81.60±22.35 (↓ 2.88)
		Reliability	0.17±0.05 (↓ 5.6)	1.78±0.20 (↓ 33.1)	1.123±0.180 (↑ 3772.4)	91.27±25.12 (↑ 8.63)
		Performance	0.14±0.02 (↓ 22.2)	2.11±0.14 (↓ 20.7)	0.024±0.006 (↓ 17.2)	68.95±12.11 (↓ 17.94)
	NFR-Enhanced	Code Design	0.06±0.02 (↓ 66.7)	1.23±0.20 (↓ 53.8)	0.090±0.039 (↑ 210.3)	73.03±26.91 (↓ 13.08)
		Readability	0.07±0.02 (↓ 61.1)	1.24±0.15 (↓ 53.4)	0.056±0.013 (↑ 93.1)	75.14±24.52 (↓ 10.57)
		Reliability	0.09±0.02 (↓ 50.0)	1.41±0.16 (↓ 47.0)	0.855±0.150 (↑ 2848.3)	81.50±27.27 (↓ 3.00)
		Performance	0.10±0.02 (↓ 44.4)	1.62±0.10 (↓ 39.1)	0.035±0.011 (↑ 20.7)	78.41±18.92 (↓ 6.68)
MBPP	-	Function-Only	0.15±0.01	3.06±0.02	0.064±0.002	39.88±2.47
	NFR-Integrated	Code Design	0.13±0.02 (↓ 13.3)	2.67±0.28 (↓ 12.7)	0.311±0.103 (↑ 385.9)	43.16±5.32 (↑ 8.22)
		Readability	0.12±0.02 (↓ 20.0)	2.47±0.23 (↓ 19.3)	0.124±0.043 (↑ 93.8)	41.60±5.15 (↑ 4.31)
		Reliability	0.20±0.06 (↑ 33.3)	2.47±0.36 (↓ 19.3)	1.504±0.169 (↑ 2250.0)	38.64±3.72 (↓ 3.11)
		Performance	0.15±0.02 (↓ 0.0)	3.78±0.19 (↑ 23.5)	0.066±0.024 (↑ 3.1)	40.21±3.88 (↑ 0.83)
	NFR-Enhanced	Code Design	0.05±0.02 (↓ 66.7)	2.18±0.46 (↓ 28.8)	0.224±0.059 (↑ 250.0)	43.02±5.56 (↑ 7.87)
		Readability	0.07±0.02 (↓ 53.3)	2.36±0.23 (↓ 22.9)	0.152±0.031 (↑ 137.5)	44.15±6.64 (↑ 10.71)
		Reliability	0.13±0.05 (↓ 13.3)	2.23±0.24 (↓ 27.1)	1.327±0.182 (↑ 1973.4)	41.12±3.79 (↑ 3.11)
		Performance	0.13±0.02 (↓ 13.3)	3.14±0.19 (↑ 2.6)	0.089±0.036 (↑ 39.1)	41.25±2.32 (↑ 3.44)

higher standard deviations (STDEV) of Pass@1 across all benchmarks compared to *RawGPT*. For example, in HumanEval, the STDEV for Pass@1 ranges from 1.84 to 2.64 for *NFR-Integrated* and 2.70 to 7.54 for *NFR-Enhanced*, both much higher than the STDEV of 0.70 for *RawGPT*. Moreover, we find that *NFR-Enhanced* exhibits higher variability in Pass@1 than *NFR-Integrated*, ***which aligns with our earlier finding that LLMs are better at generating functionally correct code in one-step approach***.

Unlike Pass@1, NFR-Enhanced leads to a larger improvement in certain non-functional code quality than NFR-Integrated. While *NFR-Integrated* outperforms *NFR-Enhanced* in Pass@1, *NFR-Enhanced* excels in improving NFR metrics. For code smell density, *NFR-Integrated* achieves a reduction of 13.3% and 44.4% on HumanEval and MBPP, respectively, whereas *NFR-Enhanced*

reduces by 66.7% for both datasets. Similarly, for readability, *NFR-Integrated* improves by 19.3%–43.2%, while *NFR-Enhanced* achieves 22.9%–53.4% enhancements. Interestingly, an inverse pattern emerges for reliability, where *NFR-Integrated* outperforms *NFR-Enhanced* with improvements of 2250.0%–3772.4% for HumanEval and MBPP, compared to *NFR-Enhanced*'s 1973.4%–2848.3%. A similar trend is observed for the performance metric, with *NFR-Integrated* reducing execution time by 17.94% compared to *NFR-Enhanced*'s 6.68% in HumanEval, but no statistically significant difference in MBPP (t-test's p-value > 0.05).

Our findings suggest that the two NFR-aware code generation workflows have varying benefits depending on the NFRs. While *NFR-Enhanced* is more effective for improving readability and reducing code designs, *NFR-Integrated* may be better suited for addressing runtime-related requirements like exception handling and performance.

On average, NFR-Integrated and NFR-Enhanced share similar levels of stability in the NFR metrics. *RawGPT* has the lowest STDEV across all NFR metrics, partly because of its lack of consideration of NFRs. In comparison, *NFR-Integrated* and *NFR-Enhanced* have larger STDEVs, but the values are often stable. For example, code smell density has an STDEV of 0.01–0.02, and unreadability density has an STDEV of 0.15–0.23 for both NFR-aware workflows.

Overall, as revealed by *RoboNFR*, even minor differences in workflows—such as executing a process like *NFR-Integrated* versus *NFR-Enhanced*—can lead to different outcomes. This highlights the importance of future research in designing clear workflows for deploying LLMs and evaluating their capabilities in NFR-aware code generation.

Summary of RQ3: When incorporating NFRs, *NFR-Integrated* consistently generates functionally correct code more frequently than *NFR-Enhanced*. While both approaches improve the relevant metrics, *NFR-Enhanced* excels in readability and code structure, whereas *NFR-Integrated* demonstrates superior performance in exception handling and runtime efficiency.

3.6 Discussion

3.6.1 Discussion of Implications

Our findings highlight implications for two key groups of stakeholders: (i) practitioners and (ii) LLM researchers.

For Practitioners. Our experiments with *RoboNFR* revealed various robustness issues in LLMs that practitioners must consider, such as the fact that different prompts expressing the same meaning can significantly affect NFR-aware code generation capability, making it crucial to carefully experiment with and select the most effective prompt. Additionally, since the NFR-aware code generation ability of LLMs may vary across versions, users should either commit to a specific model version instead of automatically adopting the latest release, or reevaluate their choice whenever an update occurs. Moreover, even minor differences in workflows—such as executing a process in a single iteration (i.e., *NFR-Integrated*) versus sequentially (i.e., *NFR-Enhanced*)—can lead to different outcomes. Finally, our comparisons of functional correctness and NFR metrics demonstrate that balancing competing objectives (e.g., Pass@1 versus non-functional code quality) is essential.

In practice, to ensure the expected outcomes of LLM-based products, it is important to establish a continuous quality assurance mechanism—for example, using *RoboNFR* to monitor the robustness of deployed LLMs in real-world scenarios and prevent unexpected changes in product behavior.

For LLM Researchers. The observed robustness issues and trade-offs between functional correctness and non-functional quality point to key directions for improving research setups and training processes. Future LLM studies should report the prompts used, the specific model version (including its release date), and detailed workflow information, as our experiments demonstrate that these factors significantly impact the NFR-aware code generation capabilities of LLMs. Moreover, research should focus on developing models that effectively address both functional and non-functional requirements to reduce trade-offs and variability in functional and non-functional code quality. Future studies should also monitor robustness across various code quality aspects and benchmarks to optimize training and better align with complex software requirements.

3.6.2 Discussion of Failure Examples When The LLM Attempts To Address Both Functional And Non-Functional Requirements

To provide more insights on the robustness issues, we present some code examples exposed by *RoboNFR*, demonstrating that LLMs make some mistakes when addressing NFRs, such as reliability, readability, performance, and code design.

Reliability

To enhance reliability, LLMs often include additional exception-handling statements in the code. However, they sometimes make errors, such as using incorrect try-except formats or raising generic exceptions instead of returning specific results as described in the problem requirements.

For example, in Code 3.1, the LLM introduced extra exception-handling code that conflicts with the specified functional requirements. The error-handling code introduces extra checks for the triangle inequality theorem on lines 23-24, leading to failure. Specifically, the function's docstring states that such inputs should return False. However, the error-handling logic raises a ValueError instead, resulting in failure. This discrepancy highlights that while NFR-aware code generation can produce functionally correct logic, it may conflict with the specified evaluation criteria. Furthermore, such unexpected outputs might unfairly lead to a significant decrease in Pass@1 scores during LLM evaluation, suggesting a potential direction for designing fairer benchmarks in the future.

```
1  def right_angle_triangle(a, b, c):
2      '''
3      Given the lengths of the three sides of a triangle. Return True if the three
4      sides form a right-angled triangle, False otherwise.
5      A right-angled triangle is a triangle in which one angle is right angle or
6      90 degree.
7
8      Example:
9      right_angle_triangle(3, 4, 5) == True
10     right_angle_triangle(1, 2, 3) == False
11     '''
12
13     # Type checking
14     for side in (a, b, c):
15         if not isinstance(side, (int, float)):
```

```
            raise TypeError(f"Invalid type: {side}. All sides must be numbers (int or float).")

    # Value checking
    if a <= 0 or b <= 0 or c <= 0:
        raise ValueError("All sides must be positive numbers.")

    # Triangle inequality theorem
    if a + b <= c or a + c <= b or b + c <= a:
        raise ValueError("The given sides do not form a triangle.")

    # Check for right angle using Pythagorean theorem
    sides = sorted([a, b, c])  # Sort the sides to identify the longest side
    return sides[0]**2 + sides[1]**2 == sides[2]**2
```

Listing 3.1: HumanEval-157 Test Failure Due to *raise ValueError* Which Violates the Functional Requirement.

Code Design

When provided with only functional requirements, the LLM can generate correct code; however, the generated code may still exhibit code smell issues. For instance, as shown in Code 3.2, the function-only solution passes the evaluation but relies heavily on multiple if-else statements for validation checks, introducing potential maintainability concerns.

```
def valid_date(date):
    """You have to write a function which validates a given date string and
    returns True if the date is valid otherwise False.
    The date is valid if all of the following rules are satisfied:
    1. The date string is not empty.
    2. The number of days is not less than 1 or higher than 31 days for months
    1,3,5,7,8,10,12. And the number of days is not less than 1 or higher than 30 days for
    months 4,6,9,11. And, the number of days is not less than 1 or higher than 29 for the
    month 2.
    3. The months should not be less than 1 or higher than 12.
    4. The date should be in the format: mm-dd-yyyy
    for example:
    valid_date('03-11-2000') => True
    valid_date('15-01-2012') => False
    valid_date('04-0-2040') => False
```

```
    valid_date('06-04-2020') => True
    valid_date('06/04/2020') => False
    """
    # Check if the date string is empty
    if not date:
        return False
    # Check if the date string is in the correct format
    if len(date) != 10 or date[2] != '-' or date[5] != '-':
        return False
    # Split the date string into month, day, and year
    try:
        month, day, year = map(int, date.split('-'))
    except ValueError:
        return False
    # Check if the month is valid
    if month < 1 or month > 12:
        return False
    # Check if the day is valid based on the month
    if month in {1, 3, 5, 7, 8, 10, 12}:
        if day < 1 or day > 31:
            return False
    elif month in {4, 6, 9, 11}:
        if day < 1 or day > 30:
            return False
    elif month == 2:
        if day < 1 or day > 29:
            return False
    # If all checks pass, the date is valid
    return True
```

Listing 3.2: HumanEval-124 correct but has code smell.

When processing NFR-aware code generation, LLMs may have the capability to avoid code smells, but they can sometimes make mistakes. For instance, as shown in Code 3.3, when prompted to avoid a specific code smell, the LLM attempted to mitigate it by using a regular expression pattern. However, it failed to produce a correct and functional pattern. These inaccuracies resulted in AssertionError exceptions during evaluation when the function was executed. This highlights the potential risk that an LLM might alter the original intent when generating regular expressions.

Future research could explore how robustly LLMs can handle regular expression logic during code generation.

```python
import re
def valid_date(date):
    # Check if the date string is empty
    if not date:
        return False

    # Define the regex pattern for the date format
    date_pattern = re.compile(r'^(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])-(\d{4})$')

    # Check if the date matches the pattern
    match = date_pattern.match(date)
    if not match:
        return False

    # Extract month, day, and year from the date
    month, day, year = map(int, match.groups())

    # Define the number of days in each month
    days_in_month = {
        1: 31, 2: 29, 3: 31, 4: 30, 5: 31, 6: 30,
        7: 31, 8: 31, 9: 30, 10: 31, 11: 30, 12: 31
    }

    # Check if the day is valid for the given month
    if day < 1 or day > days_in_month[month]:
        return False
    return True
assert candidate("03-15-957") == True # match fail
```

Listing 3.3: HumanEval-124 Test Failure while LLM using re to avoid code smell but introduce other error.

Readability

Similar to its ability to address code design, LLMs can enhance the readability of generated code. However, we observed that LLMs sometimes prioritize suggesting strategies over producing more readable code. For example, as shown in Code 3.4, while the LLM attempted to improve code

readability, it occasionally failed to generate valid code. Instead, it returned strategy descriptions as responses, which led to evaluation failures. When executed, this resulted in a SyntaxError during evaluation. During our evaluation, we observed that earlier models (e.g., *GPT-3.5-turbo*) more frequently generated instructional responses instead of actual code, as illustrated here. This suggests that, to ensure the robustness of LLMs in code generation tasks, the development process should include test cases that verify whether the model's output contains valid code.

```
One way to enhance readability and improve the code is to add comments to explain the
    logic and steps of the algorithm. This will make it easier for others (and yourself)
    to understand the code.
Another way to improve the code is to use more descriptive variable names. For example,
    instead of using "l" as the variable name for the list, you could use a more
    descriptive name such as "input_list".
Additionally, you could extract the logic inside the function into smaller, more
    descriptive functions. For example, you could create a function to check if a triple
    of elements sum to zero, and then call that function inside the main function.
Lastly, you could add type hints to the function signature to make it clear what type of
    input the function expects and what type of output it returns. This can improve
    readability and help catch potential errors early on.
```

Listing 3.4: HumanEval-40 Test Failure while LLM improve readability.

Performance

In the absence of specific requirements for code generation, LLMs may prioritize generating correct code while overlooking other critical factors, such as execution time. For instance, as shown in Code 3.5, the *RawGPT* solution implements an algorithm with $O(n^2)$ time complexity. While the code is functionally correct, it is computationally inefficient.

```
def car_race_collision(n: int):
    """
    Simulates the car collisions explicitly and calculates the number of collisions.
    Arguments:
    n -- Number of cars in each group (left-to-right and right-to-left)
    Returns:
    int -- Number of collisions
    """

    # Initialize collision count
```

```
    collisions = 0

    # Iterate through each car in the left-to-right group
    for _ in range(n):
        # Each car collides with all cars in the right-to-left group
        for _ in range(n):
            collisions += 1
    return collisions
```

Listing 3.5: HumanEval-41, Function-only-generated code, correct but slow.

When tasked with optimizing an algorithm's runtime, the LLM occasionally makes mistakes, such as using incorrect formulas to solve the problem. For instance, in Code 3.6, the correct formula for calculating the number of collisions is $n \times n$. However, the LLM sometimes employed alternative mathematical formulas, resulting in incorrect outputs. During evaluation, these inaccuracies led to AssertionError exceptions when the function was executed. Research on using LLMs to enhance the execution speed of existing code may benefit from adding checkpoints to address such issues—for example, by prompting the LLM to verify whether the generated mathematical formulas align with the original functional requirements.

```
# Correct-Code:
def car_race_collision(n: int):
    return n * n

# LLM NFR-aware Generated Code: Efficient but Incorrect
def car_race_collision(n: int):
    return n * (n - 1) // 2
```

Listing 3.6: HumanEval-41 Test Failure while LLM improve performance but use wrong formula.

3.7 Threats to Validity

Internal Threats. The primary objective of our framework is to assess LLM robustness in NFR-aware code generation by evaluating both Pass@1 and non-functional code quality. Although *RoboNFR* is not explicitly pre-trained for code refinement, it mirrors how developers use LLMs,

including prompt design, model version, and workflow choices—for both code generation and refinement tasks. The insights from our evaluation can inform improvements in future model architectures, guide the prioritization of code optimization efforts, and help develop strategies for more effective and robust handling of non-functional requirements in generated code. Future work could explore refining training processes or implementing targeted NFR optimization techniques during code generation and refinement.

External Threats. We use a certain set of widely used LLMs to conduct the experiments. The results may not apply to all models, as results may vary across different architectures and training methods. Future studies could benefit from incorporating a broader range of models to validate the results. In this study, we have primarily examined Python datasets. While Python is a widely used language, the generalizability of our framework to other programming languages remains to be fully explored. However, our framework is not inherently language-specific. It is expected to be applicable to other languages and can be further verified by future studies.

3.8 Conclusion

This study examines the challenges and opportunities of integrating non-functional requirements (NFRs) into code generation using large language models (LLMs). We introduce *RoboNFR*, a generalizable framework for evaluating LLM robustness in NFR-aware code generation by incorporating prompt variations, regression testing, and diverse workflows for leveraging LLM capabilities. Our findings reveal potential robustness issues and significant trade-offs between functional correctness and non-functional code quality attributes such as design, readability, reliability, and performance.

Our study demonstrates that while incorporating NFRs into code generation reduces the functional correctness metric (e.g., Pass@1), it yields notable improvements in non-functional code quality. However, we also observed a heightened potential for robustness issues. Our analysis of three evaluation methodologies highlights the importance of robustness: The selection of semantically equivalent prompts can significantly impact both functional and NFR metrics; model

updates require regression testing to maintain consistency; and different workflows—such as *NFR-Integrated* and *NFR-Enhanced* —demonstrate varying degrees of effectiveness in addressing specific NFR aspects. By providing real-time feedback, *RoboNFR* facilitates continuous monitoring of LLM robustness and helps identify the optimal combination of prompt design, model version, and workflow selection, ultimately enhancing LLM-based solutions and development efficiency.

Chapter 4

Conclusion

4.1 Summary

In this thesis, we explored the integration of software engineering knowledge with large language models (LLMs). Chapter 2 examined how different software process models influence the code generation capabilities of LLMs. Our findings show that *FlowGen* reduces variations in generated code across changes in temperature settings and model versions, while $FlowGen_{Scrum}$ achieves the best overall performance among the models evaluated. Testing activities have the most significant impact on ensuring code correctness, whereas code design and review activities play a crucial role in enhancing code quality. Moreover, combining *FlowGen* with additional practices, such as increased testing (e.g., $FlowGen_{Scrum+Test}$), further boosts the Pass@1 score of generated code.

Chapter 3 further evaluates the robustness of LLMs in NFR-aware code generation. Our findings indicate that integrating NFRs into prompts generally leads to a decrease in Pass@1 scores compared to *RawGPT*, which does not mention NFRs—reflecting reduced accuracy in the generated code. However, this integration improves code quality in terms of performance, design, reliability, and readability. At the same time, incorporating NFRs reduces the robustness of LLMs' code generation ability, as it introduces greater variation in the generated code across both correctness and quality dimensions. We also observe that changes in model versions significantly affect both the correctness and quality of NFR-aware code. Among the models evaluated, *GPT-4o* demonstrates

the highest robustness. Furthermore, different prompting workflows yield distinct outcomes: *NFR-Integrated* achieves higher code correctness, while *NFR-Enhanced* produces better code quality.

Overall, by examining two key aspects of software engineering knowledge, this thesis highlights their importance—particularly in how they influence the code generation capabilities of LLMs. The findings also suggest that human developers should incorporate software engineering principles when integrating LLMs into real-world software development.

4.2 Discussion and Future Work

This thesis presents key findings on the importance of software engineering knowledge when applying large language models (LLMs) for code generation, and highlights several interesting directions for future applications and related research. Specifically, Chapter 2 demonstrates that process models play a critical role, suggesting that the responsibilities of human developers may evolve as LLMs are integrated into these workflows. However, since current process models are primarily designed for human developers, future work should explore how to design new models that support effective human–LLM collaboration. In particular, it is important to investigate whether the activities in existing models remain valuable in LLM-augmented workflows—that is, to study the individual impact of each activity in modernized process models.

Meanwhile, Chapter 3 highlights the need for users to be cautious of LLM robustness issues, particularly in NFR-aware code generation. Real-world LLM-based products must carefully preselect stable prompts, workflows, and model versions to ensure consistent output and mitigate unexpected behaviors caused by robustness limitations. This also suggests the need for further research that simulates real-world LLM usage—examining how developers interact with LLMs across various scenarios (e.g., different settings or versions), and how users address practical challenges beyond code correctness, such as reliability, readability, and performance.

Although the experiments in this thesis are subject to certain limitations due to resource constraints, the findings underscore the importance of incorporating software engineering knowledge into LLM-related research. Future work should continue to integrate software engineering perspectives when investigating the capabilities and applications of large language models.

References

AlOmar, E. A., Venkatakrishnan, A., Mkaouer, M. W., Newman, C. D., & Ouni, A. (2024). How to refactor this code? an exploratory study on developer-chatgpt refactoring conversations. *arXiv preprint arXiv:2402.06013*.

Anonymous. (2024). *Repository & dataset.* Retrieved 2024-03-21, from https://anonymous.4open.science/r/FlowGen-LLM-E842

Anonymous, A. (2024). *Link to our replication package.* https://anonymous.4open.science/r/NFRGen-8175.

Anthropic. (2024). *The Claude 3 Model Family: Opus, Sonnet, Haiku.* Retrieved from https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf ([Online; accessed 20-March-2025])

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., ... others (2021). Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Bassil, Y. (2012). A simulation model for the waterfall software development life cycle. *arXiv preprint arXiv:1205.6904*.

Biringa, C., & Kul, G. (2023). *Pace: A program analysis framework for continuous performance prediction.* Retrieved from https://arxiv.org/abs/2312.00918

Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G., & Chen, W. (2023). Codet: Code generation with generated tests. In *11th international conference on learning representations (iclr)*.

Chen, J., Li, Z., Hu, X., & Xia, X. (2024). Nlperturbator: Studying the robustness of code llms to

natural language variations. *arXiv preprint arXiv:2406.19783*.

Chen, J., Lin, H., Han, X., & Sun, L. (2023). Benchmarking large language models in retrieval-augmented generation. *arXiv preprint arXiv:2309.01431*.

Chen, L., Zaharia, M., & Zou, J. (2023b). How is chatgpt's behavior changing over time? *arXiv preprint arXiv:2307.09009*.

Chen, L., Zaharia, M., & Zou, J. Y. (2023a). How is chatgpt's behavior changing over time? *ArXiv*, *abs/2307.09009*.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... others (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chung, L., & do Prado Leite, J. C. S. (2009). On non-functional requirements in software engineering. *Conceptual modeling: Foundations and applications: Essays in honor of john mylopoulos*, 363–379.

Copilot. (2024a). *An overview about using copilot in visual studio.* https://code.visualstudio.com/docs/copilot/overview. (Accessed: 2024-11-05)

Copilot. (2024b). *Using github copilot to ask some coding-related questions.* https://docs.github.com/en/copilot/using-github-copilot/asking-github-copilot-questions-in-your-ide. (Accessed: 2024-11-05)

Cursor. (2024). *Cursor features.* https://www.cursor.com/features. (Accessed: 2024-11-05)

Dasgupta, S., & Hooshangi, S. (2017). Code quality: Examining the efficacy of automated tools.

De Padua, G. B., & Shang, W. (2017). Revisiting exception handling practices with exception flow analysis. In *2017 ieee 17th international working conference on source code analysis and manipulation (scam)* (pp. 11–20).

Dong, Y., Ding, J., Jiang, X., Li, G., Li, Z., & Jin, Z. (2023). Codescore: Evaluating code generation by learning code execution. *arXiv preprint arXiv:2301.09043*.

Dong, Y., Jiang, X., Jin, Z., & Li, G. (2023). Self-collaboration code generation via chatgpt. *arXiv preprint arXiv:2304.07590*.

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., & Zhang, J. M. (2023).

Large language models for software engineering: Survey and open problems. Retrieved from https://arxiv.org/abs/2310.03533

Fetzer, C., Felber, P., & Hogstedt, K. (2004). Automatic detection and masking of nonatomic exception handling. *IEEE Transactions on Software Engineering*, *30*(8), 547–560.

Fowler, M. (2005). *The new methodology.* Retrieved 2024-03-21, from https://www.martinfowler.com/articles/newMethodology.html

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Gao, S., Gao, C., Gu, W., & Lyu, M. (2024). Search-based llms for code optimization. In *2025 ieee/acm 47th international conference on software engineering (icse)* (pp. 254–266).

Han, H., Kim, J., Yoo, J., Lee, Y., & won Hwang, S. (2024). *Archcode: Incorporating software requirements in code generation with large language models.* Retrieved from https://arxiv.org/abs/2408.00994

Hong, S., Zheng, X., Chen, J., Cheng, Y., Wang, J., Zhang, C., ... others (2023). Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., ... Schmidhuber, J. (2024). *Metagpt: Meta programming for a multi-agent collaborative framework.* Retrieved from https://arxiv.org/abs/2308.00352

Huang, D., Bu, Q., & Cui, H. (2023). Codecot and beyond: Learning to program and test like a developer. *arXiv preprint arXiv:2308.08784*.

Huang, D., Bu, Q., Qing, Y., & Cui, H. (2024). *Codecot: Tackling code syntax errors in cot reasoning for code generation.* Retrieved from https://arxiv.org/abs/2308.08784

Huang, D., Bu, Q., Zhang, J. M., Luck, M., & Cui, H. (2023). Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Kang, S., Yoon, J., & Yoo, S. (2023). Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 ieee/acm 45th international conference on software engineering (icse)* (pp. 2312–2323).

Kumar, S., & Talukdar, P. (2021). Reordering examples helps during priming-based few-shot learning. *arXiv preprint arXiv:2106.01751*.

Le, V.-H., & Zhang, H. (2023). Log parsing with prompt-based few-shot learning. In *2023 ieee/acm 45th international conference on software engineering (icse)* (pp. 2438–2449).

Lee, S., Jang, S., Jang, S., Lee, D., & Yu, H. (2024). Exploring language model's code generation ability with auxiliary functions. *arXiv preprint arXiv:2403.10575*.

Li, J., Li, G., Li, Y., & Jin, Z. (2023). Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599*.

Li, Z., Chen, A. R., Hu, X., Xia, X., Chen, T.-H., & Shang, W. (2023). Are they all good? studying practitioners' expectations on the readability of log messages. In *2023 38th ieee/acm international conference on automated software engineering (ase)* (pp. 129–140).

Lin, F., Kim, D. J., Tse-Husn, & Chen. (2024). Soen-101: Code generation by emulating software process models using large language model agents. *arXiv preprint arXiv:2403.15852*.

Liu, J., Wang, K., Chen, Y., Peng, X., Chen, Z., Zhang, L., & Lou, Y. (2024). *Large language model-based agents for software engineering: A survey.* Retrieved from https://arxiv.org/abs/2409.02977

Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023). Is your code generated by chatgpt really correct. *Rigorous evaluation of large language models for code generation. CoRR, abs/2305.01210*.

Liu, P., Yuan, W., Fu, J., Jiang, Z., et al. (2023, jan). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.*, *55*(9).

Ma, L., Yang, W., Xu, B., Jiang, S., Fei, B., Liang, J., ... Xiao, Y. (2024). Knowlog: Knowledge enhanced pre-trained language model for log understanding. In *Proceedings of the 46th ieee/acm international conference on software engineering* (pp. 1–13).

Ma, Z., Chen, A. R., Kim, D. J., Chen, T.-H., & Wang, S. (2024). Llmparser: An exploratory study on using large language models for log parsing. In *2024 ieee/acm 46th international conference on software engineering (icse)* (pp. 883–883).

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., ... others (2024). Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, *36*.

Malik, H., Hemmati, H., & Hassan, A. E. (2013). Automatic detection of performance deviations in the load testing of large scale systems. In *2013 35th international conference on software*

engineering (icse) (pp. 1012–1021).

Maximilien, E. M., & Williams, L. (2003). Assessing test-driven development at ibm. In *25th international conference on software engineering, 2003. proceedings.* (pp. 564–569).

Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., & Gao, J. (2024). *Large language models: A survey.* Retrieved from https://arxiv.org/abs/2402.06196

Mishra, M., Stallone, M., Zhang, G., Shen, Y., Prasad, A., Soria, A. M., ... others (2024). Granite code models: A family of open foundation models for code intelligence. *arXiv preprint arXiv:2405.04324*.

Nashid, N., Sintaha, M., & Mesbah, A. (2023). Retrieval-based prompt selection for code-related few-shot learning. In *Ieee/acm 45th international conference on software engineering (icse)* (pp. 2450–2462).

Nunez, A., Islam, N. T., Jha, S. K., & Najafirad, P. (2024). *Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing.* Retrieved from https://arxiv.org/abs/2409.10737

OpenAI. (2023). *Chatgpt.* https://chatgpt.com/.

Park, J. S., O'Brien, J., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology* (pp. 1–22).

Pereira dos Reis, J., Brito e Abreu, F., de Figueiredo Carneiro, G., & Anslow, C. (2022). Code smells detection and visualization: a systematic literature review. *Archives of Computational Methods in Engineering*, 47–94.

Pham, H. (2000). *Software reliability*. Springer Science & Business Media.

Piantadosi, V., Fierro, F., Scalabrino, S., Serebrenik, A., & Oliveto, R. (2020). How does code readability change during software evolution? *Empirical Software Engineering*, 5374–5412.

PyCQA. (2024a). *Pylint user guide: Messages overview-convention.* https://pylint.pycqa.org/en/latest/user_guide/messages/messages_overview.html#convention. (Accessed:2024/12/03)

PyCQA. (2024b). *Pylint user guide: Messages overview - refactor.* https://

pylint.pycqa.org/en/latest/user_guide/messages/messages_overview.html#refactor. (Accessed: 2024-12-03)

Python. (2005). *Built-in exceptions.* Retrieved 2024-03-21, from https://docs.python.org/3/library/exceptions.html

Qian, C., Cong, X., Yang, C., Chen, W., Su, Y., Xu, J., ... Sun, M. (2023). Communicative agents for software development. *arXiv preprint arXiv:2307.07924*.

Rasheed, Z., Sami, M. A., Waseem, M., Kemell, K.-K., Wang, X., Nguyen, A., ... Abrahamsson, P. (2024). *Ai-powered code review with llms: Early results.* Retrieved from https://arxiv.org/abs/2404.18496

Ruiz, F. V., Grishina, A., Hort, M., & Moonen, L. (2024). A novel approach for automatic program repair using round-trip translation with large language models. *arXiv preprint arXiv:2401.07994*.

Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2023). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*.

Shao, Y., Li, L., Dai, J., & Qiu, X. (2023). Character-llm: A trainable agent for role-playing. *arXiv preprint arXiv:2310.10158*.

Shen, C., Yang, W., Pan, M., & Zhou, Y. (2023). Git merge conflict resolution leveraging strategy classification and llm. In *2023 ieee 23rd international conference on software quality, reliability, and security (qrs)* (pp. 228–239).

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., & Yao, S. (2024). Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*.

Shirafuji, A., Watanobe, Y., Ito, T., Morishita, M., Nakamura, Y., Oda, Y., & Suzuki, J. (2023). Exploring the robustness of large language models for solving programming problems. *arXiv preprint arXiv:2306.14583*.

Singhal, M., Aggarwal, T., Awasthi, A., Natarajan, N., & Kanade, A. (2024). *Nofuneval: Funny how code lms falter on requirements beyond functional correctness.* Retrieved from https://arxiv.org/abs/2401.15963

Team, P. D. (n.d.). *Pylint.* https://pypi.org/project/pylint/. (Last accessed March

2024.)

Testsigma. (n.d.). *What is devtestops?* https://testsigma.com/devtestops. (Last accessed March 2024.)

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., ... others (2023). Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Vitale, A., Piantadosi, V., Scalabrino, S., & Oliveto, R. (2023). Using deep learning to automatically improve code readability. In *2023 38th ieee/acm international conference on automated software engineering (ase)* (pp. 573–584).

Waghjale, S., Veerendranath, V., Wang, Z. Z., & Fried, D. (2024). *Ecco: Can we improve model-generated code efficiency without sacrificing functional correctness?* Retrieved from https://arxiv.org/abs/2407.14044

Walter, B., & Alkhaeir, T. (2016). The relationship between design patterns and code smells: An exploratory study. *Information and Software Technology*, 127–142.

Wang, S., Li, Z., Qian, H., Yang, C., Wang, Z., Shang, M., ... others (2022). Recode: Robustness evaluation of code generation models. *arXiv preprint arXiv:2212.10264*.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., ... others (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, *35*, 24824–24837.

White, J., Hays, S., Fu, Q., Spencer-Smith, J., & Schmidt, D. C. (2023). Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *arXiv preprint arXiv:2303.07839*.

White, J., Hays, S., Fu, Q., Spencer-Smith, J., & Schmidt, D. C. (2024). Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. In *Generative ai for effective software development* (pp. 71–108).

Wu, D., Mu, F., Shi, L., Guo, Z., Liu, K., Zhuang, W., ... Zhang, L. (2024). ismell: Assembling llms with expert toolsets for code smell detection and refactoring. In *Proceedings of the 39th ieee/acm international conference on automated software engineering* (pp. 1345–1357).

Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., ... others (2023). The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*.

Xie, Z., Chen, Y., Zhi, C., Deng, S., & Yin, J. (2023). Chatunitest: a chatgpt-based automated unit test generation tool. *arXiv preprint arXiv:2305.04764*.

Xu, J., Luo, X., Pan, X., Li, Y., Pei, W., & Xu, Z. (2022). Alleviating the sample selection bias in few-shot learning by removing projection to the centroid. *Advances in Neural Information Processing Systems*, *35*, 21073–21086.

Xu, Y., Lin, F., Yang, J., Tse-Hsun, Chen, & Tsantalis, N. (2025). *Mantra: Enhancing automated method-level refactoring with contextual rag and multi-agent llm collaboration.* Retrieved from https://arxiv.org/abs/2503.14340

Xu, Y., Wang, S., Li, P., Luo, F., Wang, X., Liu, W., & Liu, Y. (2023). Exploring large language models for communication games: An empirical study on werewolf. *arXiv preprint arXiv:2309.04658*.

Yamashita, A., & Moonen, L. (2012). Do code smells reflect important maintainability aspects? In *2012 28th ieee international conference on software maintenance (icsm)* (pp. 306–315).

Yang, H., Yue, S., & He, Y. (2023). Auto-gpt for online decision making: Benchmarks and additional opinions. *arXiv preprint arXiv:2306.02224*.

Yetiştiren, B., Özsoy, I., Ayerdem, M., & Tüzün, E. (2023). Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. *arXiv preprint arXiv:2304.10778*.

Yuan, Z., Liu, J., Zi, Q., Liu, M., Peng, X., & Lou, Y. (2023). Evaluating instruction-tuned large language models on code comprehension and generation. *arXiv preprint arXiv:2308.01240*.

Yuan, Z., Lou, Y., Liu, M., Ding, S., Wang, K., Chen, Y., & Peng, X. (2023). No more manual tests? evaluating and improving ChatGPT for unit test generation. *arXiv preprint arXiv:2305.04207*.

Zhang, J., Wang, X., Zhang, H., Sun, H., Pu, Y., & Liu, X. (2020). Learning to handle exceptions. In *Proceedings of the 35th ieee/acm international conference on automated software engineering* (pp. 29–41).

Zhang, Q., Fang, C., Xie, Y., Zhang, Y., Yang, Y., Sun, W., ... Chen, Z. (2024). *A survey on large language models for software engineering.* Retrieved from https://arxiv.org/abs/2312.15223

Zhang, Y., Qiu, Z., Stol, K.-J., Zhu, W., Zhu, J., Tian, Y., & Liu, H. (2024). Automatic commit message generation: A critical review and directions for future work. *IEEE Transactions on Software Engineering*.

Zhang, Z., Chen, C., Liu, B., Liao, C., Gong, Z., Yu, H., ... Wang, R. (2024). *Unifying the perspectives of nlp and software engineering: A survey on language models for code.* Retrieved from https://arxiv.org/abs/2311.07989

Zhao, Q., Chabbi, M., & Liu, X. (2023). *Easyview: Bringing performance profiles into integrated development environments.* Retrieved from https://arxiv.org/abs/2312.16598

Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., ... Wen, J.-R. (2025). *A survey of large language models.* Retrieved from https://arxiv.org/abs/2303.18223

Zheng, Z., Ning, K., Wang, Y., Zhang, J., Zheng, D., Ye, M., & Chen, J. (2024). *A survey of large language models for code: Evolution, benchmarking, and future trends.* Retrieved from https://arxiv.org/abs/2311.10372

Zhong, L., Wang, Z., & Shang, J. (2024). Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*.

Zhou, A., Yan, K., Shlapentokh-Rothman, M., Wang, H., & Wang, Y.-X. (2023). Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*.