Universität des Saarlandes
MI Fakultät für Mathematik und Informatik
Department of Computer Science

Master's Thesis

Analyzing Iterative Code Refinement: Assessing ChatGPT's Consistency in Improving Code Readability

submitted by

Julia Hess (2549454)

on September 30th, 2025

Reviewers

Prof. Dr. Sven Apel (First Examiner)

Prof. Dr. Jilles Vreeken (Second Examiner)

Erklärung Statement

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne die Beteiligung dritter Personen verfasst habe, und dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderweitigen fremden Äußerungen entnommen wurden, sind als solche kenntlich gemacht. Insbesondere bestätige ich hiermit, dass ich bei der Erstellung der nachfolgenden Arbeit mittels künstlicher Intelligenz betriebene Software (z. B. ChatGPT) ausschließlich zur Textüberarbeitung/-korrektur und zur Code-Vervollständigung und nicht zur Bearbeitung der in der Arbeit aufgeworfenen Fragestellungen zu Hilfe genommen habe. Alle mittels künstlicher Intelligenz betriebenen Software (z. B. ChatGPT) generierten und/oder bearbeiteten Teile der Arbeit wurden kenntlich gemacht und als Hilfsmittel angegeben. Ich erkläre mich damit einverstanden, dass die Arbeit mittels eines Plagiatsprogrammes auf die Nutzung einer solchen Software überprüft wird. Mir ist bewusst, dass der Verstoß gegen diese Versicherung zum Nichtbestehen der Prüfung bis hin zum Verlust des Prüfungsanspruchs führen kann.

I hereby declare that I have written this thesis independently and without the involvement of third parties, and that I have used no sources or aids other than those indicated. All passages taken directly or indirectly from publications or other external sources have been identified as such. In particular, I confirm that I have used AI-based software (e.g., ChatGPT) exclusively for the following permitted sub-tasks: text rewriting/revision and code completion, and not to address or formulate the main research questions of the thesis. All parts of the thesis that were generated and/or edited using AI-based software (e.g., ChatGPT) have been disclosed and documented in accordance with the documentation requirements. I agree that the thesis may be checked using plagiarism detection software, including checks for the use of such software. I am aware that any violation of this declaration may result in failing the examination and lead to losing the right to be examined.

Saarbrücken, _____ _____

(Datum Date) (Unterschrift Signature)

Einverständniserklärung (optional)

Declaration of Consent (optional)

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____

(Datum Date) (Unterschrift Signature)

To my family, friends, and other supporters along the (very long) way

Abstract

Large Language Models (LLMs) such as ChatGPT are increasingly used for automated code generation and refactoring. While they can produce seemingly well-readable code, their iterative refinements may exhibit inconsistencies and unpredictable behavior. This study systematically examines how ChatGPT refines code over multiple iterations, focusing on evolution patterns, convergence behavior, and the impact of explicit prompt guidance with a particular focus on code readability.

Our study combines a pilot experiment with a large-scale main experiment based on 221 Java snippets, each systematically varied and refined across five iterations under three different prompting strategies. We introduce a custom DiffParser that integrates sequence-based, token-based, and AST-based similarity measures to capture fine-grained code modifications, enabling the categorization of changes into semantic, syntactic, and comment-level transformations. The results reveal three main insights: (1) iterative refinements exhibit an initial phase of substantial restructuring followed by stabilization, suggesting a convergence tendency; (2) convergence patterns are robust across different code variants, although the type and distribution of modifications vary; and (3) explicit prompting toward key readability factors, such as naming or commenting, influences refinement dynamics but does not fundamentally alter convergence trajectories.

The contribution of this work lies in providing a reusable, LLM-independent framework for studying code evolution under iterative refinement. This methodological foundation opens pathways for future research, including comparative analyses across models, systematic evaluation of additional software quality dimensions such as maintainability or security, and the exploration of long-term evolutionary dynamics in AI-generated code.

Acknowledgements

I would like to take this opportunity to thank all those who have supported me along the way: Norman, who accompanied me throughout the many ups and downs of this long thesis process and during my time as a student assistant at the chair, always offering encouragement and support. Sven, to whom I owe the confidence and courage to pursue this degree after a phase of reorientation. My deepest gratitude goes to my family and friends who have always stood by my side with unwavering support. Finally, I wish to thank all those who, beyond the lecture halls, have taught me valuable lessons for life.

Contents

Chapter 1

Introduction

The advent of large language models (LLMs) such as ChatGPT has revolutionized the field of artificial intelligence and natural language processing. These models, powered by extensive training on diverse datasets, have demonstrated remarkable proficiency in understanding and generating human language. Among the myriad applications of LLMs, their potential to enhance software development practices stands out as particularly impactful.

Code quality and readability are fundamental aspects of software development. Readable code not only facilitates easier maintenance and debugging but also enhances collaboration among developers. Code refactoring, the process of restructuring existing code to improve readability and maintainability without altering its functionality, is a critical aspect of software engineering. Poorly written code can lead to increased maintenance costs, reduced developer productivity, and a higher likelihood of bugs and errors. Thus, ensuring that code meets acceptable standards from its initial generation is crucial.

ChatGPT, with its advanced language understanding capabilities, offers a promising avenue for both generating and refactoring code. In code generation, ChatGPT can produce initial versions of code that are clear and maintainable at first sight, hopefully reducing the need for extensive refactoring. This means that when a programmer integrates generated code into a project, it should already adhere to acceptable readability and quality standards.

By leveraging its ability to comprehend and generate natural language, ChatGPT could assist developers in generating or transforming complex, convoluted code into more readable and maintainable versions. This would not only enhance code quality but also contribute to more efficient and effective software development processes.

Given the rapid growth and adoption of LLMs, exploring ChatGPT's ability to refactor code is both timely and relevant. It aligns with the ongoing efforts to integrate AI into software engineering, ultimately aiming to improve productivity and code quality in the software development industry.

However, a critical question remains: Does ChatGPT truly have the capability to refactor code meaningfully? While ChatGPT can generate responses, the practicality and sensibility of these responses for code refactoring are essential questions for both research and practice.

The remainder of this thesis is structured as follows: Chapter 2 provides the necessary background, introducing fundamental concepts and frameworks relevant to this research. Chapter 3 discusses related work, summarizing existing approaches and highlighting the gap this study aims to address.Chapter 4 presents the proposed methodology, consisting of the pilot experiment and the subsequent main experiment. Chapter 5 reports the findings of these experiments, while Chapter 6 provides a critical discussion, outlines potential avenues for future work, and addresses threats to validity. Finally, Chapter 7 concludes the thesis by summarizing the key contributions and insights.

Chapter 2

Background

The rapid advancement of artificial intelligence (AI) and large language models (LLMs) has had a profound impact on software engineering, particularly in the areas of code comprehension, readability assessment, and automated code generation. This chapter provides the necessary theoretical and technical foundations for this study by discussing key concepts related to these topics.

2.1 Code Comprehension

An understanding of source code is crucial in software engineering, as it directly impacts tasks such as debugging, code review, and refactoring. The term *Code Comprehension* refers to the cognitive processes involved in understanding code, a fundamental task in software engineering. Code comprehension involves cognitive processes such as pattern recognition, memory retrieval, and abstraction. Research in this area seeks to determine what factors influence comprehension and how developers interact with code on a cognitive level [1].

Early studies primarily relied on self-reports and observational methods to assess comprehension difficulty [2]. More recent approaches integrate eye-tracking and neuroimaging techniques (e.g., fMRI and EEG) to gain insights into how programmers process code at a neural level [3, 4]. These studies suggest that expert programmers rely more on top-down comprehension, forming hypotheses about a code snippet before verifying details, while novices adopt a bottom-up approach, building an understanding incrementally [5].

As LLMs continue to improve, their potential to support comprehension extends beyond textual explanations [6, 7]. This growing role of LLMs in code comprehension underscores

the need for a deeper investigation into their effectiveness and limitations, particularly in the context of automated refactoring.

In the context of this study, it is important to situate this concept within the broader framework of *Understanding Source Code*, as it represents a fundamental skill for interpreting and working with code, which is directly relevant to the process of refactoring, and therefore forms the foundation for the refactoring tasks performed by ChatGPT.

2.2 Code Readability Models

A crucial aspect of code comprehension is readability, i.e. the ease with which code can be understood. Readability is influenced by factors such as variable naming, indentation, comments, and syntactic simplicity [8, 9]. The development of readability models aims to quantify and predict how readable a given piece of code is [10, 11].

Traditional readability models were based on heuristic metrics, such as Halstead complexity measures and McCabe's Cyclomatic Complexity [12]. While these models provide quantifiable measures of code complexity, they often fail to capture the subjective and context-dependent nature of readability. Readability is not solely determined by syntactic complexity but also influenced by individual cognitive factors, programming experience, and familiarity with coding conventions [13]. As a result, purely heuristic approaches may misrepresent how developers actually perceive and evaluate code readability. To address these limitations, machine learning (ML) approaches have been introduced [14, 15].

To predict readability scores, ML-based readability models not only leverage hand-crafted features, such as line length, identifier complexity, and indentation consistency, but also leverage large datasets and empirical readability assessments to provide more accurate and context-aware predictions. More recently, deep learning methods, particularly transformer-based models such as CodeBERT and GraphCodeBERT, have outperformed previous approaches by learning semantic representations from large-scale code corpora [16]. These models integrate syntactic structure with semantic context, improving their ability to assess code readability.

Beyond static analysis, readability models are now being adapted into integrated development environments (IDEs), offering real-time feedback to developers. Additionally, integrating large language models (LLMs) for readability assessment is an emerging trend, as these models can refine code suggestions dynamically based on human feedback [17].

2.3 Emergence and Evolution of Large Language Models (LLMs)

The field of natural language processing (NLP) has undergone a rapid transformation with the advent of LLMs. Early language models relied on n-gram and statistical methods, gradually evolving into recurrent neural networks (RNNs) and long short-term memory networks (LSTMs) [18]. The paradigm shift occurred with the introduction of the Transformer architecture by Vaswani et al. (2017), which laid the foundation for models such as GPT and BERT [19].

Key milestones in LLM development include:

- **GPT-1 (2018)**: Introduced transformer-based pretraining using large text corpora [20].
- **GPT-2 (2019)**: Demonstrated coherent long-form text generation with 1.5B parameters [21].
- **GPT-3 (2020)**: Scaled up to 175B parameters, enabling few-shot learning capabilities [22].
- **GPT-4 (2023)**: Enhanced reasoning, multimodal understanding, and greater context depth [23].
- **GPT-4o (2024)** : "Omni" variant with multimodal capabilities (text, image, audio).
- **GPT-5 (2025)**: Introduced in August 2025 with variants such as GPT-5 Pro, mini, nano; Goal: more intelligent routing between fast and detailed responses
- **Code-Specific Models**: Models such as Codex (OpenAI), Code Llama (Meta), and StarCoder (BigCode) were fine-tuned on code repositories, optimizing their ability to generate and understand code [24].

The integration of LLMs into software development has led to applications such as code completion, automatic documentation, and debugging assistance. Current challenges include mitigating biases, improving efficiency, and developing smaller yet powerful models that can operate on edge devices. The trajectory of LLM evolution suggests a future where AI-assisted programming becomes a standard in software engineering, complementing human expertise rather than replacing it.

Chapter 3

Related Work

3.1 LLMs in Software Engineering

In their paper *Towards an Understanding of Large Language Models in Software Engineering Tasks*, Zheng et al. [6] provide an extensive investigation into the integration of Large Language Models (LLMs) in software engineering, categorizing and analyzing 123 relevant research papers. The study emphasizes the growing importance of LLMs in various software engineering tasks, including code generation, vulnerability detection, and program repair, highlighting both the potential and current limitations of these models. Zheng et al. offer a granular breakdown of how LLMs like ChatGPT and Codex perform across different tasks, noting that while LLMs excel in tasks requiring syntactical understanding, such as code summarization, they struggle with tasks requiring deep semantic comprehension, such as complex code generation and vulnerability detection.

In comparison, Hou et al. [7] also conducted a systematic review, focusing on the broader application of LLMs in software engineering by analyzing 395 research papers published between January 2017 and January 2024. The review categorizes LLMs based on their architectures, such as encoder-only, encoder-decoder, and decoder-only models, and explores their use across a wide range of SE tasks, including code generation, program repair, and software design. The study highlights the increasing integration of LLMs in SE, particularly noting the dominance of decoder-only models like GPT and Codex in tasks requiring code generation and completion. The review also identifies key challenges, such as the need for better dataset curation and the development of more sophisticated evaluation metrics, to enhance the effectiveness of LLMs in SE.

Both studies underline the need for further refinement in LLMs, particularly in improving their understanding of code semantics and reliability in more complex tasks. However,

Zheng et al. go further by proposing specific categorization schemes and providing detailed insights into the specific challenges faced by LLMs in software engineering, offering a more focused perspective on the nuanced performance of LLMs in specific tasks, which complements Hou et al.'s broader systematic analysis.

These papers provide detailed empirical evidence on the strengths and weaknesses of LLMs in software engineering, which aligns with the exploration of how effectively ChatGPT can be employed in code refactoring and improving code quality.

3.2 ChatGPT and Programming

Tian et al. [25] conducted an empirical study to assess ChatGPT's potential as a fully automated programming assistant, focusing on three core tasks: code generation, program repair, and code summarization. The study revealed that while ChatGPT excels in generating correct code for common programming problems, it struggles with generalizing to new, unseen challenges. Additionally, the research highlighted that ChatGPT's performance in program repair is competitive but limited by its attention span and the effectiveness of provided problem descriptions. The model demonstrated an unexpected capability in summarizing code, accurately identifying the intended functionality of both correct and incorrect code snippets. These findings suggest that while ChatGPT shows promise as a programming assistant, especially in aiding code comprehension and fixing, its current limitations in handling novel problems and providing consistent repair solutions indicate that further refinement is necessary.

Jin et al. [26] conducted an empirical evaluation of ChatGPT's effectiveness in supporting developers, focusing on code generation. Their study analyzed interactions from the DevGPT dataset, comprising real-world developer conversations involving ChatGPT. The findings reveal that while ChatGPT is frequently used for generating code, its output is typically more suited for demonstrating concepts or providing documentation examples rather than being production-ready. The study highlights that generated code often requires significant modifications before integration into production, and the tool is most effective in contexts where developers request improvements or additional context rather than entirely new code generation. This research underscores the current limitations of ChatGPT in practical software development and emphasizes the need for further refinement before LLMs like ChatGPT can be fully integrated into modern development workflows.

Liu et al. [27] present a comprehensive empirical assessment of ChatGPT's code generation capabilities, focusing on correctness, complexity, and security. The study evaluates

ChatGPT's performance across 728 algorithm problems in five programming languages and 18 Common Weakness Enumeration (CWE) scenarios. The findings reveal that while ChatGPT is more effective at generating functionally correct code for problems dated before 2021, its performance significantly drops for more recent problems. The study also highlights the limitations of ChatGPT's multi-round fixing process, where attempts to correct erroneous code often result in increased code complexity without fully resolving functional issues. Moreover, the research uncovers notable security vulnerabilities in the code generated by ChatGPT, although the model demonstrates some success in addressing these vulnerabilities through iterative prompts. These findings underscore the challenges and potential risks associated with relying on ChatGPT for automated code generation, particularly in producing secure and maintainable code.

Liu et al. [28] also investigated the quality and reliability of code generated by ChatGPT, with a focus on characterizing and mitigating common code quality issues. Analyzing over 4,000 code snippets generated for 2,033 programming tasks in Java and Python, the study highlights significant challenges in the correctness and maintainability of ChatGPT-generated code. The findings reveal that nearly half of the generated code suffers from maintainability issues, such as poor code style and unnecessary complexity. Moreover, while ChatGPT can address some of these issues when provided with specific feedback, the model frequently introduces new problems during the refinement process. The study emphasizes the importance of incorporating advanced feedback mechanisms to enhance ChatGPT's ability to generate higher-quality, maintainable code.

This research is particularly relevant to the current study as it outlines the limitations and potential improvements needed for using ChatGPT in tasks like code refactoring, directly aligning with the investigation of ChatGPT's effectiveness in improving code readability and quality.

Yu et al. [29] critically assess the reliability of ChatGPT in source code-related tasks, focusing on its self-verification capabilities. The study explores how effectively ChatGPT can self-verify its own generated code, completed code, and program repairs across various datasets. The findings reveal significant limitations in ChatGPT's self-verification process, including a high rate of erroneous self-assessments, where the model often incorrectly predicts the correctness, security, and success of its code outputs. The study highlights the issue of "self-contradictory hallucinations", where ChatGPT initially generates code it believes is correct but later contradicts this during self-verification. While the use of guiding questions and test report prompts can improve the detection of vulnerabilities and bugs, they also introduce false positives, complicating the model's reliability. These findings underscore the need for human oversight and the development

of more sophisticated verification mechanisms before ChatGPT can be trusted fully in automated software development tasks.

AlOmar et al. [30] conducted an exploratory study examining developer-ChatGPT interactions during code refactoring tasks. Using the DevGPT dataset, the study analyzed 17,913 conversations to identify how developers articulate refactoring needs and how ChatGPT responds to these requests. The research highlights that developers frequently use generic terms when requesting refactoring, while ChatGPT often explicitly states the refactoring intention, focusing on improving quality attributes such as maintainability, readability, and code organization. The study also found that while ChatGPT can suggest useful refactorings, its understanding of the broader codebase context is limited, sometimes leading to incorrect or incomplete suggestions. The findings suggest that developers need to craft more specific prompts to maximize the effectiveness of ChatGPT in refactoring tasks.

DePalma et al.[31] conducted an empirical study to assess ChatGPT's capabilities in performing code refactoring, focusing on its effectiveness, consistency, and ability to preserve code functionality. The study involved refactoring 40 Java code segments across eight quality attributes, including performance, complexity, and readability. The results indicated that ChatGPT was successful in refactoring code in 319 out of 320 trials, offering both minor and significant improvements. However, while ChatGPT demonstrated strengths in generating documentation and preserving code behavior, the study highlighted its unpredictability and inconsistency, with identical prompts sometimes yielding different results. Additionally, ChatGPT struggled with complex refactoring tasks, often making superficial changes that did not address deeper issues. The research concludes that while ChatGPT can be a valuable tool for simple refactoring tasks, human oversight remains essential. This study is relevant to the current investigation into ChatGPT's role in improving code quality and readability, particularly in the context of automated refactoring.

Guo et al. [17] present an empirical study examining ChatGPT's potential for automated code refinement in the context of code review. Using the CodeReview benchmark and a newly constructed high-quality dataset, they compare ChatGPT (GPT-3.5/4) against CodeReviewer, a state-of-the-art tool based on CodeT5. Their findings show that ChatGPT achieves superior generalization and higher EM and BLEU scores (22.78 and 76.44) than CodeReviewer (15.50 and 62.88) on the new dataset, although performance remains limited overall. The study also highlights that ChatGPT performs best with low temperature settings and concise, scenario-based prompts, but struggles when review comments lack clarity, precise locations, or domain-specific knowledge. The authors identify mitigation strategies such as improving review quality and leveraging stronger

models like GPT-4. This work provides evidence that LLMs can meaningfully support automated code refinement, while also emphasizing their current limitations and the need for refined evaluation metrics and datasets.

Hu et al. [32] investigate the robustness of code language models when confronted with poor-readability code, a dimension largely neglected in existing evaluation benchmarks. While prior research has primarily tested models such as CodeBERT, CodeT5, and CodeLlama on well-structured, high-readability code, this study systematically degrades readability through obfuscation techniques that perturb both semantic (e.g., identifiers, function names) and syntactic (e.g., operators, branches) features. Their empirical results demonstrate that current models exhibit a strong dependency on semantic cues and perform poorly when these cues are eroded, revealing limited robustness to syntactic variations. To address these shortcomings, the authors introduce PoorCodeSumEval, a novel benchmark designed to assess code summarization models across multiple readability levels and perturbation types. This contribution highlights a critical gap in existing evaluation practices and provides a more rigorous framework for understanding the limitations of LLMs in real-world scenarios, where code readability often varies significantly.

Liu et al. [33] propose CodeQUEST, a framework that leverages GPT-4o for iterative code quality evaluation and enhancement across multiple dimensions, including readability, maintainability, testability, efficiency, and security. The framework integrates an evaluator, which provides structured quantitative and qualitative assessments, with an optimizer that applies the feedback to refine code in successive cycles. Evaluated on 42 Python and JavaScript examples, CodeQUEST achieved improvements in 41 cases, with the majority of gains occurring in early iterations, and demonstrated stronger alignment with established metrics such as Pylint, Radon, and Bandit compared to a baseline. Notably, the framework was able to identify issues overlooked by traditional tools, particularly in security and scalability, highlighting the potential of LLMs for systematic and multi-faceted code refinement while acknowledging limitations regarding subjectivity, stochasticity, and language coverage.

3.3 Gap Analysis

Large Language Models (LLMs) such as ChatGPT have shown potential in automating code refactoring and improvement tasks, yet fundamental questions about their behavior under iterative self-refinement remain unresolved. While prior work has raised concerns about "reality distortion" and contradictory self-improvements in AI-generated content [34], little is known about what actually happens when LLMs are repeatedly prompted to improve their own code outputs. Specifically, it is unclear whether these models eventually converge toward stable, higher-quality solutions (potentially guided by implicit notions of best practices) or whether they continue to introduce superficial, oscillating, or even regressive modifications across iterations.

Moreover, the degree to which such iterative refinements differ under unguided versus targeted prompts, or when applied to code of varying initial quality, has not been systematically investigated. Existing studies acknowledge ChatGPT's ability to refactor code [31], but they stop short of analyzing how the nature and depth of changes evolve over successive refinement cycles.

To address this gap, this research develops a framework for systematically analyzing code modifications produced by LLMs across multiple iterations. The approach begins with a broad characterization of how changes emerge at different refinement stages, and then progressively decomposes these transformations by type - insertions, deletions, and modifications of syntactic and semantic elements. This layered analysis enables a deeper understanding of the patterns underlying iterative refinement, providing insights into whether and how LLMs move toward convergence, and under which prompting or code-quality conditions such processes are more or less effective.

Chapter 4

Methodology

This study aims to analyze how ChatGPT modifies source code across multiple iterations, focusing on measurable patterns of change. Before investigating the dynamics of these iterative refinements, it is essential to clarify what ChatGPT itself considers to be readable or understandable code. Understanding the model's conceptualization of code readability provides the foundation for evaluating whether its refinements are consistent with established principles of software engineering or merely reflect superficial transformations. Therefore, the methodology consists of two main phases:

1. **Pilot Experiment:** A preliminary study designed to identify methodological challenges, refine our approach, and explore ChatGPT's behavior in code transformations.
2. **Main Experiment:** A systematic, large-scale analysis of ChatGPT's modifications using a structured dataset and quantitative metrics.

To build a structured foundation for our study, we first identified two preliminary research questions that explore ChatGPT's approach to code understandability and its consistency in iterative improvements:

- **RQ1$_{\text{Pilot}}$:** What is ChatGPTs' perspective on factors of code understandability (= *key factors*)?
- **RQ2$_{\text{Pilot}}$:** How consistent is ChatGPT with improving code snippets iteratively according to these key factors?

These questions emerged from the hypothesis that if a language model possesses an inherent understanding of a concept, its output should reflect a certain level of internal

consistency and integrity. In the context of code readability, this implies that if ChatGPT has 'learned' what constitutes readable and well-structured code, it should not only be able to articulate these principles in natural language but also apply them consistently in its generated code. Conversely, if no such understanding exists, we would not expect the model to reliably produce well-structured improvements.

Based on this reasoning, our first research question ($RQ1_{Pilot}$) investigates ChatGPT's perspective on code understandability: What factors does it consider important for readable code? If its conceptualization aligns with established research on software readability, we can then examine whether these principles are reflected in the way ChatGPT modifies code ($RQ2_{Pilot}$). This approach allows us to assess both the alignment of ChatGPT's internal representations with best practices and its ability to apply them consistently across multiple iterations of code refinement.

Section 4.1 presents the setup and findings of the pilot experiment, while Section 4.2 outlines the methodology for the main experiment.

4.1 Pilot Experiment

Key Factors of Code Understandability ($RQ1_{Pilot}$)

To answer our first preliminary research question ($RQ1_{Pilot}$), we employed a modified approach of *Thematic Analysis* to extract the *Key Factors of Code Understandability* from ChatGPT's responses to answer our first research question ($RQ1_{Pilot}$). Thematic Analysis (TA) as described by Braun and Clarke (2006) is a widely used qualitative research method for identifying, analyzing, and reporting patterns (themes) within data. This approach is flexible and can be applied across various theoretical frameworks, making it suitable for diverse research contexts. Thematic Analysis follows a structured, yet iterative process consisting of six key phases: (1) familiarization with the data, (2) generating initial codes, (3) searching for themes, (4) reviewing themes, (5) defining and naming themes, and (6) producing the final report.

Instead of generating open codes inductively, a set of responses was iteratively collected based on variations of the question: "What makes code well understandable?" These responses were then systematically mapped onto existing terminology from prior research on code understandability and readability. This deductive mapping process continued until a point of saturation was reached, ensuring that no significant new insights emerged. By adapting the thematic analysis process in this way, the study leveraged both the

depth of AI-generated qualitative data and the structured foundation of established research in software engineering.

Results

The analysis resulted in **14 key factors of code understandability**, which comprehensively cover the current state of research in this area. We categorized these factors into three levels: (1) the *Code Level*, including naming conventions, commenting, and consistent formatting; (2) the *Architecture Level*, covering modularity, reusability, and adherence to design patterns; (3) the *Maintenance Level*, involving practices such as testing, refactoring, and code reviews. Table 4.1 summarizes the results of the Thematic Analysis.

This categorization of key factors is based on a structured grouping process. We determined the most appropriate overarching category for each key factor and grouped them together. The categorization was guided by the primary impact of each factor, for example:

- "Comments" directly enhance the readability of a specific code block or line, making them part of the Code Level.
- "Patterns" contribute to the overall design clarity and structure of the code, placing them under the Architectural Level.
- "Regular Refactoring" is not an inherent property of the code itself but rather an activity that ensures readability over time, classifying it under the Maintenance Level.

Since these factors align with established research on code readability and understandability, ChatGPT's responses appear to be consistent with widely accepted software engineering principles. However, it remains an open question whether this consistency stems from a deeper conceptual understanding of readable code or merely reflects patterns learned from training data. To explore this further, we proceed to $\mathrm{RQ2}_{\mathrm{Pilot}}$, examining in our pilot experiment how consistently these factors are applied when ChatGPT iteratively refines code.

Key Factor	Description	Explanation
Category 1 - Code Level		
KF1 - Clear Names	Meaningful and consistent names for variables, functions, and classes	More readable, self-explanatory, and reduces ambiguity
KF2 - Comments	Comments should clarify complex logic, document assumptions, and explain non-trivial decisions	Helps others understand the reasoning behind the code, making maintenance easier
KF3 - Formatting	Consistent formatting including indentation, spacing, and line breaks	Improves visual clarity and uniformity, making the code easier to read
KF4 - Simplicity	Breaking down complex problems into smaller, manageable units	Easier to understand and maintain, reduces cognitive load
KF5 - Abstraction	Managing cognitive load by abstracting complex logic into reusable components	Prevents overwhelming the reader and improves modularity
KF6 - Error Handling	Clearly handling errors and exceptions in a predictable manner	Makes failures easier to debug and understand
Category 2 - Architectural Level		
KF7 - Structure	Organizing code logically into meaningful components	Enhances code maintainability and readability
KF8 - Domain Concepts	Aligning code structure with real-world concepts from the problem domain	Increases consistency and improves understanding
KF9 - Modularity	Designing modular code where functions and classes serve a single purpose	Allows better separation of concerns and easier reuse
KF10 - Patterns	Using established design patterns for common programming problems	Makes code more predictable and easier to follow
KF11 - Global State	Minimizing the use of global variables	Reduces unintended side effects and improves testability
Category 3 - Maintenance Level		
KF12 - Testing	Writing tests and examples for critical code components	Provides confidence in correctness and assists in future modifications
KF13 - Regular Refactoring	Continuously improving the structure and readability of code	Prevents technical debt and keeps the codebase clean
KF14 - Code Reviews	Peer reviewing code for feedback and improvements	Improves code quality through collective expertise

Table 4.1: Results of the *Thematic Analysis* on ChatGPT's "Understanding" of Key Factors of Code Understandability

Qualitative Analysis ($RQ2_{Pilot}$)

To anwer our second preliminary research question on how ChatGPT refactors code for better readability, we conducted an iterative, multi-round experiment[1], incorporating a qualitative data collection and analysis approach. Each of the four rounds involved providing ChatGPT with code snippets and evaluating its suggestions based on the prompt: *"How can I improve this code for better readability?"*. The refactored snippets from round *n-1* were then used as input for the round *n*, each in a separate new chat to prevent biases based on the chat history. Figure 4.1 visualizes this process.

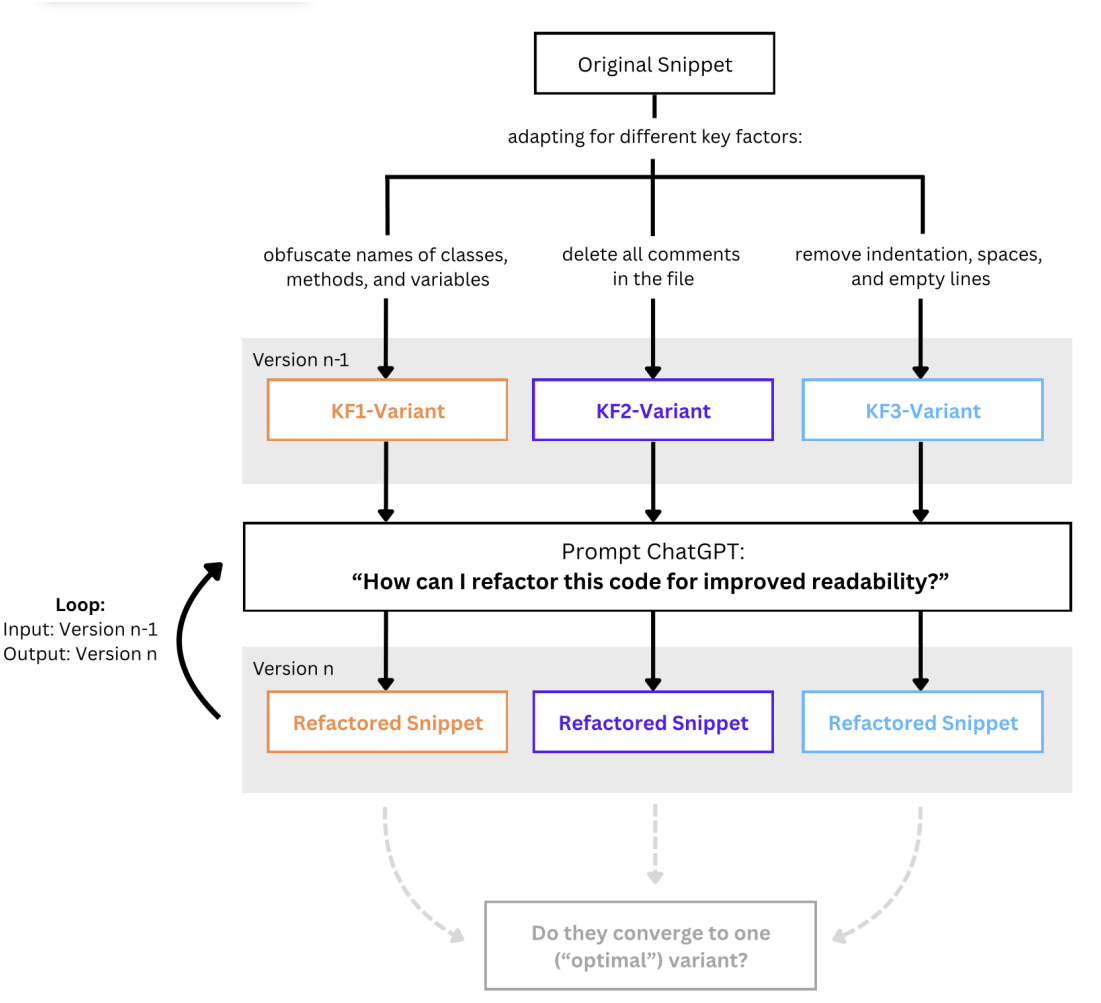

Figure 4.1: Schematic Pilot Experiment Setup

[1]The pilot experiment was conducted with OpenAI's GPT-4 model between January and April 2024.

4.1.1 Basic Terminology

To ensure clarity and consistency in the following analysis, we adopt the following terminology:

Original Snippet: an implementation of an algorithm, which is used as a starting point for the analysis.

Variant (of a snippet): corresponds to a modification made in relation to a specific *Key Factor* (KF), which drives the changes in the code. These variants are created to examine how ChatGPT refines code snippets based on different baselines.

Version (of a variant): refers to the iteration in which a particular variant was created during the process of code improvement. These versions are tracked to examine how ChatGPT refines the code across multiple iterations.

4.1.2 Key Factor and Snippet Selection

To ensure precise evaluation, we chose small, well-defined code snippets rather than large codebases. The primary reasons for this choice are:

- Small snippets are easier to construct and analyze systematically.
- Evaluation is simplified since we can focus solely on changes within the snippet.
- Architectural and maintenance factors (category 2 & 3, see Table 4.1) are inherently difficult to assess in isolated code snippets.
- While direct studies on ChatGPT's use for quick fixes or local improvements are limited, we assume that in real-world scenarios, ChatGPT is often used for quick fixes or local improvements rather than full project-level readability enhancements.

Given these considerations, we selected three key factors from *Category 1 - Code Level* that are most suitable for evaluation:

- **KF1 (Clear Names)**: The clarity and descriptiveness of variable, method, and class names. Good naming should reflect the purpose of a variable or function, making the code self-explanatory and reducing the need for additional comments.
- **KF2 (Comments)**: The presence of meaningful, concise comments that aid in understanding the code's logic and purpose. Well-structured comments, including inline explanations and documentation comments (e.g. JavaDocs), should provide necessary context without redundancy.

- **KF3 (Formatting)**: The adherence to uniform spacing, indentation, and structuring conventions that improve code readability. Proper formatting follows established coding style guidelines and ensures the visual clarity of code blocks, aiding comprehension and maintainability.

The other factors from Category 1 (*Simplicity*, *Abstraction*, and *Error Handling*) were excluded as they are more applicable to larger code segments or algorithmic challenges.

The chosen key factors are not only essential for readability but also well-suited for evaluation within the constrained scope of isolated code snippets—aligning with practical use cases where developers rely on ChatGPT for quick function generation or localized code refinement rather than full-scale project development.

To ensure a diverse selection of code structures and problem-solving approaches, we selected four simple, widely known algorithms implemented in Java[2], each representing different computational paradigms:

1. **Binary Search:** A classic divide-and-conquer algorithm that efficiently finds an element in a sorted array by repeatedly halving the search space.
2. **Bubble Sort:** A simple yet inefficient sorting algorithm that repeatedly swaps adjacent elements until the array is sorted, illustrating iterative refinement and nested loops.
3. **Check Prime:** A basic mathematical function that determines whether a number is prime by checking divisibility, representing conditional logic and loops.
4. **Fibonacci:** A recursive function that generates the Fibonacci sequence, showcasing recursion and function calls in algorithmic design.

For each algorithm, we introduced specific alterations based on the three key factors, to create different *variants* of the snippet (Example in Figure 4.2):

- **KF1-Variant:** Replaced all class, variable, and method names with meaningless identifiers (e.g. `x`, `method1`, `class1`).
- **KF2-Variant:** Removed all comments and JavaDocs.
- **KF3-Variant:** Stripped the code of proper formatting (e.g. inconsistent indentation, missing line breaks, etc.).

[2]The full code snippets are listed in the Appendix in Figure A.1.

```java
public class CheckPrime {
    public static void main(String[] args) {

        int num = 29;

        boolean flag = false; // to signal whether number is prime

        for(int i = 2; i <= num / 2; ++i) {

            if(num % i == 0) {
                flag = true;
                break;
            }
        }
        if (!flag)
            System.out.println(num + " is a prime number.");
        else
            System.out.println(num + " is not a prime number.");
    }
}
```

(a) Original Variable and Class Names

```java
public class Class3 {
    public static void main(String[] args) {

        int n = 29;

        boolean f = false; // to signal whether number is prime

        for(int i = 2; i <= n / 2; ++i) {

            if(n % i == 0) {
                f = true;
                break;
            }
        }
        if (!f)
            System.out.println(n + " is a prime number.");
        else
            System.out.println(n + " is not a prime number.");
    }
}
```

(b) KF1-Variant of CheckPrime Snippet

```java
public class CheckPrime {
    public static void main(String[] args) {

        int num = 29;

        boolean flag = false; // to signal whether number is prime

        for(int i = 2; i <= num / 2; ++i) {

            if(num % i == 0) {
                flag = true;
                break;
            }
        }
        if (!flag)
            System.out.println(num + " is a prime number.");
        else
            System.out.println(num + " is not a prime number.");
    }
}
```

(c) Original Comment(s)

```java
public class CheckPrime {
    public static void main(String[] args) {

        int num = 29;

        boolean flag = false;

        for (int i = 2; i <= num / 2; ++i) {

            if (num % i == 0) {
                flag = true;
                break;
            }
        }
        if (!flag)
            System.out.println(num + " is a prime number.");
        else
            System.out.println(num + " is not a prime number.");
    }
}
```

(d) KF2-Variant of CheckPrime Snippet

```java
public class CheckPrime {
    public static void main(String[] args) {

        int num = 29;

        boolean flag = false; // to signal whether number is prime

        for(int i = 2; i <= num / 2; ++i) {

            if(num % i == 0) {
                flag = true;
                break;
            }
        }
        if (!flag)
            System.out.println(num + " is a prime number.");
        else
            System.out.println(num + " is not a prime number.");
    }
}
```

(e) Original Formatting

```java
public class CheckPrime {
public static void main(String[] args) {

int num = 29;boolean flag = false; // to signal whether number is prime
for(int i = 2; i <= num / 2; ++i) {
if(num % i == 0) {flag = true;break;}}
if (!flag)System.out.println(num + " is a prime number.");else System.out.println(num + " is not a prime number.");}}
```

(f) KF3-Variant of CheckPrime Snippet

Figure 4.2: Example of Variant Creation for CheckPrime Snippet. Code adaptations introduced by us are highlighted with red boxes. They were not part of the input to ChatGPT.

4.1.3 Expectation

In round 1, which served primarily as a baseline, we sought to observe the kinds of improvements ChatGPT would generally apply to the snippets. We eliminated modifications unrelated to the designated key factor of a given variant subsequently by adjusting the snippets, thereby creating a cleaner version in which, ideally, only the code elements deliberately varied with respect to the corresponding key factor would attract ChatGPT's attention in round 2. By round 3, we hypothesized that the refinements would become more balanced, no longer centered around a single KF, and that we might observe a gradual convergence toward an optimized version of the code through iterative feedback loops.

4.1.4 Results

General Findings

While ChatGPT did not strictly adhere to a single key factor per round, its feedback was initially more concentrated on the specific KFs that had been deliberately altered in the respective snippets, as we had expected (see Table 4.2).

As the iterations progressed and many of these targeted improvements were incorporated, the model's suggestions became increasingly diverse (as expected), addressing multiple KFs simultaneously rather than focusing on a single one. This shift suggests that ChatGPT distributes its refinement efforts more broadly as fewer explicit weaknesses remain.

While this behavior aligns with our intuitive expectations, it is noteworthy that the iterative refinement process did not lead to a clear convergence toward a single "best" version within these four rounds. Despite the model's increasingly broad distribution of refinement efforts, no definitive consensus emerged regarding an optimal formulation, suggesting that ChatGPT's feedback remains adaptable rather than gravitating toward a singular ideal outcome.

Even worse, the iterative feedback process also showed diminishing returns, partially with back-and-forth modification. It sometimes introduces unnecessary or even counter-productive changes.

Observations per Key Factor

Over multiple iterations, formatting-related suggestions (KF3) disappeared, while naming improvements (KF1), comment adjustments (KF2), and structural simplifications (KF4) remained dominant.

- **KF1 - Clear Names:** ChatGPT frequently suggested renaming variables and methods for clarity. However, naming improvements sometimes caused unnecessary back-and-forth refinements.
- **KF2 - Comments:** ChatGPT often gave generic suggestions like "Add comments for clarification" but was inconsistent in applying them across rounds. Notably, when variable names were clear, ChatGPT suggested fewer comments, implying it may recognize code clarity through naming.
- **KF3 - Formatting:** Formatting is often fixed implicitly, without explicit recognition in ChatGPT's textual explanations. Specifically, the model neither points out the obvious instances of poor formatting nor produces poorly formatted code.

  This suggests that, within the scope of the current study, KF3 as the initial variant does not provide meaningful insights for further investigation and can be excluded from the analysis moving forward.

Table 4.2 shows ChatGPT's textual output from each round of the pilot experiment for the `BinarySearch`-Snippet in the KF1-Variant. The "Label" columns shows the category of Key Factor, to which the feedback was assigned.

Summary

In the pilot experiment, the snippets under investigation were `BinarySearch`, `BubbleSort`, `CheckPrime`, and `Fibonacci`. Each snippet had three distinct variants, labeled KF1, KF2, and KF3, each representing a different aspect of the modification process. ChatGPT was asked to refactor the snippets with respect to readability in a four-round iterative experiment.

This pilot experiment served as an initial exploration to identify potential methodological pitfalls and refine the experimental setup. A key finding was that ChatGPT's textual explanations often diverged from the changes of its generated code output.

While quantifying this discrepancy is theoretically possible, its practical relevance is limited since we cannot influence the underlying model's evolution. Instead, we can focus

Round	Output	Label	Text → Code
1	Rename method a to binarySearch.	KF1	yes
1	Rename variables for clarity (x to searchValue, y to end, z to mid).	KF1	yes
1	Use meaningful variable names in main.	KF1	yes
1	Add inline comments for clarity.	KF2	**no**
1	Replace (y - x) / 2 with (end - start) / 2 to avoid integer overflow.	noKF	yes
2	Rename method a to binarySearch.	KF1	yes
2	Rename variable n to target for clarity.	KF1	yes
2	Rename variable x to start and y to end.	KF1	yes
2	Add more descriptive comments.	KF2	yes
2	Ensure consistent formatting and indentation.	KF3	yes
3	Use JavaDoc Comments: Document the binarySearch method using JavaDoc style to help other developers understand its purpose, parameters, and return type.	KF2	yes
3	Enhance Readability of the Main Method: Consider extracting the process of printing the result into a separate method or making the logic in the main method simpler and more direct.	KF4	yes
3	Variable Naming and Scoping: Keep variable naming clear and consistent. Also, define variables close to their usage if it doesn't hinder readability.	KF1	yes
4	Separate utility methods from main method: Isolate business logic from execution.	KF4	yes
4	Use descriptive method names: Rename methods for clarity.	KF1	yes
4	Refine comments: Ensure comments are helpful and concise.	KF2	**no**

Table 4.2: ChatGPT's Code Refinement Suggestions for the `BinarySearch`-Snippet (KF1-Variant).
Label = to which of the three key factors we assigned the suggestions, where 'noKF' means the suggestions are not related to code readability, like code optimization itself.
Text → *Code* = the textual suggestions also appear in the refactored code.

on raising awareness of these inconsistencies, emphasizing that programmers should not blindly trust ChatGPT's textual descriptions when copying generated code.

Another challenge identified in the pilot study was the complexity of automating a semantic comparison between textual explanations and code snippets. A fully automated approach would require sophisticated natural language understanding to determine whether the described improvements were correctly applied in the generated code, making it infeasible within the scope of this thesis.

The pilot experiment was conducted on a small dataset consisting of four algorithms, each

in three variations, iterated over four rounds. While this provided valuable qualitative insights, our primary goal for the main experiment is to conduct a more extensive quantitative analysis of how code evolves over multiple iterations to see whether a stable and optimized version can ultimately be achieved.

Threats to Validity

We identified several potential threats to the validity of this pilot experiment and addressed them to ensure the reliability and generalizability of the results, and minimization of their effect in the main experiment.

- **Standard Algorithms Bias:** The selected snippets are well-known algorithms likely included in ChatGPT's training data. This may influence how well ChatGPT can recognize and refactor them. However, since the focus of the study is on ChatGPT's multi-round self-improvement behavior, the original code itself becomes less relevant. After the initial rounds of modification, the code evolves into a version that is essentially the product of ChatGPT's iterative improvements, regardless of whether the algorithm is a well-known one or a newly created, fictional one. Therefore, the primary concern is not the nature of the starting code but rather the number of iteration cycles, after which each input snippet should reach a similar stage of refinement.
- **Snippet Size Limitation:** Given the simplicity of the snippets, the necessity for comments (KF2) may not be as pronounced as in larger, more complex functions. In the main experiment, we will use slightly more complex code snippets, that originally contain comments.
- **Non-Determinism of ChatGPT's Responses:** One potential threat to validity is the non-deterministic nature of ChatGPT's responses. Given that the model's outputs can vary each time the same input is provided, this introduces an element of unpredictability that could affect the consistency of the results. To address this issue in the main experiment, we utilize the OpenAI API, which allows for control over the `temperature` parameter. By adjusting the temperature, we can minimize non-determinism as much as possible [35], ensuring more stable and reproducible outputs across different iterations of the experiment.

4.2 Main Experiment

The pilot experiment provided valuable insights into potential methodological challenges and informed the refinement of our experimental design. Building on these preliminary findings, the main experiment represents the core of this thesis and aims to systematically address the refined research questions. This section outlines the complete methodology employed for the main study, ranging from the selection of code snippets to the final analysis procedures.

We begin by presenting the research questions that guide the study (4.2.1). Next, we describe the process of snippet sampling and the construction of the final dataset (4.2.2). The subsequent sections detail the implementation of our *DiffParser* pipeline (4.2.3), the definition of evaluation metrics (4.2.4), and the analytical approaches applied to examine convergence and contradiction in code refinements (4.2.5). Together, these components form a coherent methodological framework that underpins the main experimental study.

4.2.1 Research Questions

With this thesis, we aim to answer the following research questions. Each of them addresses a different dimension of ChatGPT's role in iterative code refinement and is motivated by gaps in prior research and the objectives of this study:

RQ1: How do iterative refinements by ChatGPT evolve when provided with a code snippet that already adheres to best practices? This question examines whether the model is able to preserve high-quality code without introducing unnecessary or contradictory modifications. It is crucial because a refactoring assistant should ideally recognize when no further improvement is required and avoid degrading code quality.

RQ2: When multiple variations of the same code snippet - each modified with respect to a single key factor of code understandability - are iteratively refined, do the refinements converge after a certain number of iterations? This question investigates whether ChatGPT normalizes different starting points into similar final solutions, which would indicate an implicit understanding of coding conventions. Convergence is of particular interest, as it reflects the stability and reliability of the refinement process across diverse inputs.

RQ3: Do targeted refinements become more effective when the prompt explicitly emphasizes the key factor in question? This question addresses the influence of

prompting on refinement quality. Since prior research has shown that LLMs are highly sensitive to input phrasing, we assess whether explicitly guiding ChatGPT toward a specific readability factor (e.g., naming or comments) leads to more consistent and meaningful improvements compared to unguided prompts.

Based on our research questions, we formulate the following hypotheses. For RQ1 and RQ2, the hypotheses are content-oriented and operationalized using the metrics described in Section 4.2.4. For RQ3, the hypothesis is directly testable, with a corresponding statistical null hypothesis.

RQ1: Evolution of Iterative Refinements

- **H1 (Content Hypothesis):** When refining code that already adheres to best practices, ChatGPT will introduce only necessary modifications while preserving the original structure and avoiding contradictory changes.

Operationalization: We measure this by tracking the number and types of modifications (semantic changes including access, call, control, literal, operator, and other structural changes; syntax-only changes; renames; comment changes; mixed changes), and structural stability (total lines, code lines, comment lines, inline comments, empty lines, number of methods).

RQ2: Convergence Across Code Variants

- **H2 (Content Hypothesis):** Iteratively refined variants of the same code snippet will converge toward a shared structure, reflecting an implicit preference of ChatGPT for certain coding patterns or solutions.

Operationalization: Convergence is assessed using the proportion of unchanged code lines, average similarity scores of modified lines, and structural correspondence from the absolute values metrics. Total insertions and deletions per iteration are tracked to examine stabilization trends, and cross-variant comparisons reveal whether final code reflects convergence or retains traces of the original variant.

RQ3: Impact of Explicitly Emphasizing Key Refinement Factors

- **H3 (Testable Hypothesis):** Explicitly emphasizing a key factor in the prompt (e.g., variable naming or commenting) leads to more effective refinements than unguided prompts.

Operationalization: Effectiveness is measured by increased alignment with the emphasized factor, and faster stabilization of the code across iterations.

Statistical Testing:

- **H0 (Null Hypothesis):** There is no significant difference in refinement effectiveness between targeted prompts and unguided prompts.
- **H1 (Alternative Hypothesis):** Targeted prompts produce significantly more effective refinements than unguided prompts, as measured by the defined metrics.

4.2.2 Snippet Sampling and Data Preprocessing

To ensure the suitability and comparability of code snippets, we decided on four criteria:

1. **LOC Interval:** The file has between 50–200 lines (i.e. overall size of the file).
2. **Methods Ratio** The chosen files must have a comparable structural complexity. This is measured by number of methods given the file size (i.e. 1–3 methods per 50 lines).
3. **Code/Comments Ratio:** A file must contain a significant amount of code versus comments ($\geq$ 50% code lines).
4. **Best Practices:** All snippets should follow a comparable standard of code quality.

For the main experiment, we decided on Java code files from the GitHub repository `The Algorithms - Java` [3]. The reasons behind this choice are: (1) This repository provides a structured and diverse collection of Java implementations of various algorithms, covering a wide range of problem domains such as searching, sorting, mathematics, and data structures; (2) it is designed for educational and comparative purposes, ensuring that the code follows consistent formatting and best practices, which facilitates systematic analysis; (3) as an open-source repository with active contributions, it ensures accessibility and reproducibility, both of which are essential for a rigorous research study. Figure 4.3 shows the relevant excerpt of the repositories' structure.

Table 4.3 summarizes the counts of files in the repository that fulfill our conditions (excluding test files). There are 221 of originally 658 files that fulfill all of our criteria.

[3] `https://github.com/TheAlgorithms/Java/tree/master`

```
src
|-- main
|   `-- java
|       `-- com
|           `-- thealgorithms
|               |-- ciphers
|               |   |-- CaesarCipher.java
|               |   `-- VigenereCipher.java
|               |-- sorts
|               |   |-- QuickSort.java
|               |   `-- MergeSort.java
|               `-- datastructures
|                   |-- BinaryTree.java
|                   `-- LinkedList.java
|-- test
|   `-- java
|       |-- CaesarCipherTest.java
|       `-- QuickSortTest.java
```

Figure 4.3: Structure of TheAlgorithms/Java Repository (Excerpt)

LOC Interval	# Files	# Methods Ratio	# Code/Comments Ratio	# Valid Files
0–49	209	202	150	143
50–99	287	227	187	148
100–149	75	56	60	45
150–199	45	32	38	28
200–249	18	9	17	9
250–299	9	4	8	3
300–349	8	8	8	8
350–399	1	0	1	0
400–449	1	1	1	1
450–499	2	0	2	0
500–549	1	0	1	0
1200–1249	1	0	1	0
2750–2799	1	0	1	0

Table 4.3: Distribution of `.java` files across LOC intervals. # Files lists all files per interval, # Method Ratio and # Code/Comments Ratio indicate those meeting the respective conditions, and # Total Valid Files shows files satisfying both.

For each of these 221 files, we created three variants:

- **KF0 - Variant**: The unchanged original implementation.
- **KF1 - Variant (no explicit naming):** Variable, method, and class names are intentionally obfuscated.
- **KF2 - Variant (no comments):** All JavaDocs as well as inline and block comments are removed.

To answer RQ1 and RQ2, we used the same prompt (pKF0) for iterative refinements (i.e. *version creation*), while we used two adapted prompts (pKF1, pKF2) to answer RQ3 (effect of explicit prompting). Table 4.4 shows an overview of the used prompt strategies.

Prompt Type	**Prompt ID**	**Prompt Content**
Unguided	pKF0	Refactor this code for improved readability.
Targeted	pKF1	Refactor this code for improved readability with respect to class/method/variable naming.
Targeted	pKF2	Refactor this code for improved readability with respect to comments.

Table 4.4: Prompts used in the experiment

To scale the experiment, we transitioned to the ChatGPT API to generate outputs in an automated and controlled manner. The API call was configured with `temperature` $= 0$ to minimize non-deterministic output variations, as recommended in prior research [35].

Instead of the full `gpt-4o` model, we deliberately used the more lightweight `gpt-4o-mini` variant. This choice was motivated by several considerations: (i) the primary task in our setting was code generation, which does not require the advanced multimodal capabilities of the larger model, ii) the smaller model significantly reduces computational costs as well as financial expenses, thereby enabling large-scale experimentation within reasonable resource constraints, and (iii) the model's faster response times improved the efficiency of data collection.

Figure 4.4 illustrates the configuration of API calls. The API itself is stateless, i.e., each request is independent of previous ones[4].

```
1         response = self.client.chat.completions.create(
2         model="gpt-4o-mini",
3         messages=[
4         {"role": "system",
5             "content": "Return only the refactored code. No additional text or
    explanation."},
6         {"role": "user",
7             "content": f"{prompt}\n{code}"}
8         ],
9         temperature=0.0  # higher values yield more creative responses
10        )
11
```

Figure 4.4: Config for ChatGPT API Usage

[4] `https://platform.openai.com/docs/guides/conversation-state?api-mode=responses`

Summary of Dataset Creation

In total, we considered 221 files from the GitHub repository `The Algorithms - Java` for our main experiment. Each file was extended into two additional variants, resulting in three variants per original snippet. Every variant was then iteratively modified across $n = 5$ refinement rounds using three distinct prompt strategies. The total number of generated snippet instances N can therefore be expressed as:

$$N = 221 \times 3 \times 5 \times 3 = 9{,}945$$

snippet instances in total, where the factors correspond to the number of files, the number of variants per file, the number of refinement rounds, and the number of prompt strategies, respectively.

4.2.3 DiffParser: A Tool for Tracking Code Changes

To conduct our experiment and evaluate the code comparisons based on the metrics (described in 4.2.4), we have developed a set of Python scripts tailored to our study, including a custom *DiffParser* designed to meet our specific requirements. We used the following key libraries:

- `javalang (v0.13.0)`[5] - to leverage the parsers capability to work with Java code
- `difflib (3.13)`[6] - to produce the unified diffs and analyze the code changes with `difflib.SequenceMatcher`
- `code_diff (0.1.3)`[7] - for AST-based code differencing
- `NetworkX (3.4.2)`[8] - to build a graph that captures code changes over multiple iterations
- `openai`[9] - to access the ChatGPT API and streamline the generation of responses (i.e. the refactored code)

The Underlying *Unified Diff* Format

To compare code sequences between versions, we utilized the *unified diff* format. In this format, lines prefixed with `---` and `+++` indicate the compared files (the original and the modified version, respectively). Each subsequent `@@ -l,n +l,m @@` header specifies the location of the change: it marks the affected line ranges in the old (`-l`) and new (`+l`) file, together with the number of lines involved (`n, m`). Within a *diff block* (i.e. scope between two headers), removed lines from the original file are prefixed with a minus sign (`-`), while added lines in the modified version are prefixed with a plus sign (`+`).

For example, in the excerpt below, the method name was changed from `approach1` to `areAnagramsBySorting`, and variable names were updated accordingly, while the underlying logic of the algorithm remained identical.

[5] `https://github.com/c2nes/javalang`
[6] `https://docs.python.org/3/library/difflib.html`
[7] `https://github.com/cedricrupb/code_diff`
[8] `https://networkx.org`
[9] `https://openai.com/index/openai-api/`

```
--- Anagrams_KF0_v0_nop.java
+++ Anagrams_KF0_v1_pKF0.java
@@ -8 +8 @@
- * typically using all the original letters exactly once.[1]
+ * typically using all the original letters exactly once.
@@ -26 +26 @@
-    public static boolean approach1(String s, String t) {
+    public static boolean areAnagramsBySorting(String s, String t) {
        @@ -30,5 +30,5 @@
        -        char[] c = s.toCharArray();
        -        char[] d = t.toCharArray();
        -        Arrays.sort(c);
        -        Arrays.sort(d);
        -        return Arrays.equals(c, d);
        +        char[] sortedS = s.toCharArray();
        +        char[] sortedT = t.toCharArray();
        +        Arrays.sort(sortedS);
        +        Arrays.sort(sortedT);
        +        return Arrays.equals(sortedS, sortedT);
        (...)

```

Listing 4.1: Example Output of difflib's unified_diff function

Parsing Procedure Overview

To classify line-level changes between two code versions, we implemented a two-stage parsing procedure. In the first stage, the algorithm iterates over all diff headers (lines beginning with `@@`) and processes each diff block separately. Within each block, removed and added lines are compared in order to identify modifications, while unmatched lines are provisionally labeled as insertions or deletions. These provisional results are stored in a candidate set for later refinement.

In the second stage, all remaining unmatched lines are collected across diff blocks and subjected to a dedicated *crossmatching* procedure. This step allows the parser to correctly align lines that were moved between different locations or otherwise shifted in ways that prevent them from being matched within a single diff block. The results of the crossmatching stage are then integrated with the initial block-level classifications to produce the final set of modifications, insertions, and deletions.

By design, the algorithm ensures consistency through internal validation checks: the total number of lines before and after crossmatching must remain constant. This guarantees

that each line in the diff is accounted for exactly once, preventing both undercounting and duplication.

The final output of the method consists of three components:

- (1) a mapping of modifications (removed-added line pairs),
- (2) a list of deletions (removed lines without a match), and
- (3) a list of insertions (added lines without a match).

Together, these results provide a complete classification of all line-level changes between the two code versions.

Line Matching and Similarity Score Calculation

To quantify modifications between code snippets, we process each diff block by comparing removed lines $r \in R$ (lines prefixed with "-") against added lines $a \in A$ (lines prefixed with "+"). Each line is first normalized by removing diff prefixes, isolating comments, and tokenizing code and comment segments separately. This results in four representations per line: raw code $(r_{\text{code}}, a_{\text{code}})$, tokenized code $(\text{tok}(r_{\text{code}}), \text{tok}(a_{\text{code}}))$, raw comments $(r_{\text{comm}}, a_{\text{comm}})$, and tokenized comments $(\text{tok}(r_{\text{comm}}), \text{tok}(a_{\text{comm}})))$.

The use of both sequence-based and token-based similarity measures is motivated by their complementary strengths. From our observations, sequence similarity tends to be rather strict: even minor structural or positional changes in a line (e.g., reordering of terms) can cause a disproportionately low similarity score, although the semantic meaning of the line remains almost unchanged. Token-based similarity, in contrast, is more tolerant to such reordering or formatting changes, as it abstracts away from the exact sequence of characters and instead focuses on the multiset of tokens. However, this tolerance may lead to overly high similarity scores in cases where semantically important structural differences are introduced. By employing both measures in parallel, we balance these two perspectives: the sequence ratio ensures sensitivity to structural order, while the token ratio provides robustness against superficial reordering. This combination yields a more reliable overall similarity estimation for code line comparisons.

For each candidate pair (r, a), we compute multiple similarity scores:

$$s_{\text{seq,code}} = \text{SeqSim}(r_{\text{code}}, a_{\text{code}}), \quad s_{\text{tok,code}} = \text{TokSim}(\text{tok}(r_{\text{code}}), \text{tok}(a_{\text{code}})),$$

$$s_{\text{seq,comm}} = \text{SeqSim}(r_{\text{comm}}, a_{\text{comm}}), \quad s_{\text{tok,comm}} = \text{TokSim}(\text{tok}(r_{\text{comm}}), \text{tok}(a_{\text{comm}})),$$

where SeqSim denotes character-level sequence similarity, and TokSim denotes token-level similarity (both using `difflib.SequenceMatcher`).

In addition, an abstract-syntax-tree (AST) similarity $s_{\text{AST}} \in [0, 1]$ is computed when both lines can be successfully parsed. Since the input consists only of code fragments extracted from the diff, preprocessing is required to make the fragments parsable by `javalang`. This preprocessing includes basic syntax completion, such as closing unmatched brackets, and wrapping fragments in a minimal Java context (e.g., embedding method or field declarations inside a dummy class, enclosing incomplete control structures in a dummy method and block, or wrapping isolated statements). From the resulting full parse tree, a simplified AST representation was derived by extracting only structurally dominant node types (e.g., declarations, control flow constructs, expressions). The purpose of this reduction was to obtain a compact node-type sequence, which can be stored as a list and compared across versions using a greedy, string-based matching process, again via `difflib.SequenceMatcher`. If parsing fails, s_{AST} is set to -1, and only sequence and token-based scores are used.

The weighted similarity score $s(r, a)$ for each candidate pair is computed as follows:

$$s(r,a) = \begin{cases} 0.5 \cdot s_{\text{seq,comm}} + 0.5 \cdot s_{\text{tok,comm}}, & \text{if } r_{\text{code}} = \emptyset, \\ 0.2 \cdot s_{\text{seq,code}} + 0.25 \cdot s_{\text{tok,code}} + 0.5 \cdot s_{\text{AST}} + 0.05 \cdot s_{\text{seq,comm}}, & \text{if } s_{\text{AST}} \neq -1, \\ 0.4 \cdot s_{\text{seq,code}} + 0.5 \cdot s_{\text{tok,code}} + 0.1 \cdot s_{\text{seq,comm}}, & \text{otherwise.} \end{cases}$$

A pair (r, a) is accepted as a modification if $s(r, a) \geq \tau$, where τ is a predefined similarity threshold of 0.6. To resolve multiple candidates, the best match is chosen by maximizing $s(r, a)$ and, in case of ties, minimizing the index distance $|r - a|$. Each line participates in at most one match, ensuring a one-to-one mapping between removed and added lines.

The weights in the similarity function as well as the similarity threshold were determined empirically through a systematic evaluation on a custom validation set. This validation set consisted of code snippet variants for which the ground truth of line correspondences was fully known (since we create variants "manually"), enabling us to assess the accuracy of different weighting schemes. To this end, we conducted an iterative optimization procedure: starting from equal weights for all components, we gradually adjusted the relative contributions of sequence-based, token-based, and AST-based similarities. At each step, the alignment results were compared against the known correspondences. The final weights were selected as those that maximized the proportion of correctly identified matches across the diverse cases in the validation set.

Conceptually, the chosen distribution reflects the intended role of each component:

- For purely comment lines ($r_{\text{code}} = \emptyset$), only comment-level similarities are considered, with equal weights assigned to sequence and token similarity. This ensures that both structural and lexical similarity in comments are taken into account without bias.
- When AST information is available, it is given the largest weight (0.5), as it provides the most reliable signal of syntactic and semantic correspondence between code fragments. Sequence and token similarities of the code part remain important (0.2 and 0.25, respectively), but serve as complementary signals. A small weight (0.05) is reserved for comment similarity to account for aligned inline comments.
- In cases where no AST could be obtained, the sequence and token similarities of the code dominate (0.4 and 0.5, respectively), while a smaller weight (0.1) is attributed to comments. This balances strict structural sensitivity with robustness to reordering, while avoiding overreliance on comment similarity.

In sum, the weighting scheme combines theoretical considerations about the relative reliability of each signal with empirical fine-tuning against a controlled test set, ensuring that the resulting similarity score $s(r, a)$ is both principled and effective in practice.

All accepted matches form the *modifications* set, while unmatched removed lines are classified as *deletions* and unmatched added lines as *insertions*. The Average Similarity Score is then computed as the mean of $s(r, a)$ over all modifications, providing a quantitative measure of how similar the changed lines are to their previous versions.

4.2.4 Metrics

To systematically evaluate the iterative refinements produced by ChatGPT, we employ a set of structured metrics that capture both the nature and progression of code modifications. The analysis is designed to answer the research questions by assessing different aspects of code evolution, stability, and convergence. We define the following metrics:

Absolute Values

For each snippet - identified as a unique combination of Snippet, Variant, Version, and Prompt - we compute a set of absolute metrics to characterize its structural properties. These metrics include the total number of lines, lines of code, comment lines, inline comments, empty lines, and number of methods.

An example for the resulting data is shown in Table 4.5. We note that the prompt labeled `nop` (i.e. "no prompt") corresponds to the base versions of the snippets ("v0"), which were manually created rather than generated through the ChatGPT API.

Version	Prompt	Total Lines	Code Lines	Comment Lines	Inline Comments	Empty Lines	Methods
0	nop	142	81	54	0	7	5
1	pKF0	114	63	44	0	7	5
1	pKF1	115	65	44	0	6	4
1	pKF2	147	81	59	28	7	5
2	pKF0	122	69	44	0	9	7
2	pKF1	115	65	44	0	6	4
2	pKF2	147	81	59	28	7	5

Table 4.5: Exemplatory absolute values metrics for the `Anagrams` Snippet (KF0-Variant)

Comparison Values

For each comparison of snippets - identified as a unique combination of Snippet, Variant 1, Variant 2, Version 1, Version 2, and Prompt - is analyzed with respect to the following set of comparison values:

- **Unchanged Code Part:** represents the number of lines that remain identical across two code versions. Conceptually, each version of the code can be decomposed into three disjoint categories: unchanged lines, modified lines, and either insertions (for the new version) or deletions (for the old version). Given m modified lines, d deletions, and i insertions, the number of unchanged lines in the old version is computed as

  $$U_{old} = |C_{old}| - m - d,$$

  where $|C_{old}|$ denotes the total number of lines in the old version. Similarly, the number of unchanged lines in the new version is calculated as

  $$U_{new} = |C_{new}| - m - i,$$

  with $|C_{new}|$ being the total number of lines in the new version. By construction, both quantities are equal ($U_{old} = U_{new}$), yielding the *Unchanged Code Part* as a consistent measure of preserved code between the two versions.

- **Modifications:** The analyzed code modifications are categorized into distinct change types to systematically capture different aspects of code evolution.

  - **Rename:** Modifications involving consistent renaming of identifiers, such as variables, methods, or constants, without affecting program behavior.

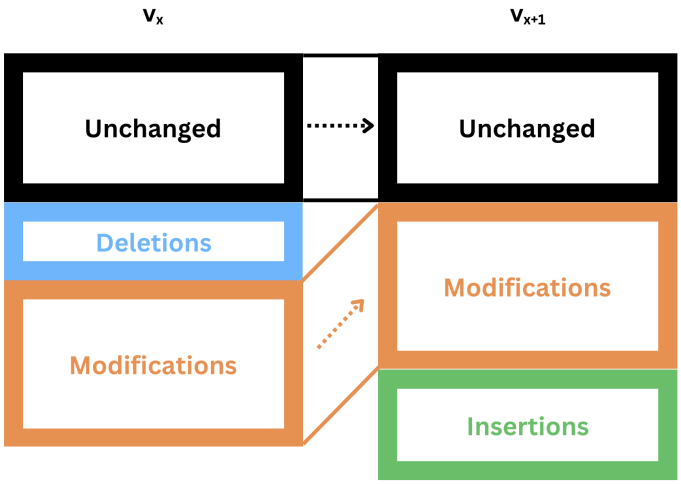

Figure 4.5: Categorization of Java source lines in file diffs

- **SyntaxOnly:** Purely syntactic adjustments, including formatting changes, reordering of code segments, or other non-semantic edits that do not alter program execution.
- **CommentChange:** Changes restricted to comments or documentation, capturing updates in explanatory or descriptive content without modifying code logic.
- **MixedChange:** Instances where multiple types of modifications occur simultaneously within a code line, reflecting combinations of semantic, syntactic, or identifier-level changes.
- **SemanticChange:** Aggregates subcategories of behavioral or functional modifications:
  - **AccessChange:** Modifications to data structure or collection accesses.
  - **CallChange:** Changes to function or method calls.
  - **ControlChange:** Adjustments to control-flow statements such as conditionals or loops.
  - **LiteralChange:** Modifications to constant values.
  - **OperatorChange:** Changes in operators affecting computation.
  - **OtherStructuralChange:** Alterations in structural constructs, e.g., method signatures or data structures.

Table A.1 shows one representative source-target pair per classification to illustrate the types of modification (excluding `SyntaxOnly`, as its example would be trivial).

- **Average Similarity Score:** quantifies the degree of resemblance between modified lines across two code versions. It is defined by first measuring the similarity $sim(l_i, l'_i)$

of each modified line l_i with its corresponding line l'_i in the preceding version, and then computing the arithmetic mean across all n modified lines:

$$\text{AvgSimScore} = \frac{1}{n} \sum_{i=1}^{n} sim(l_i, l'_i).$$

This aggregated metric provides an overall measure of how similar the modified code segments remain between two snippets.

- **Total Insertions:** Measures the total number of lines added in each snippet, further distinguished by type: code lines, comment lines, and empty lines.
- **Total Deletions:** Measures the total number of lines removed in each snippet, similarly categorized into code lines, comment lines, and empty lines.

Using these comparison metrics, we further quantify the relative changes in absolute values - such as total lines, code lines, comment lines, inline comments, empty lines, and number of methods - across snippet comparisons.

It is important to note that comparisons are obviously only conducted between snippets sharing the same name. Depending on the analysis, we either compare snippets of the same variant across different versions (horizontal comparison) or snippets of the same version across different variants (vertical comparison), which we will explain in more detail next.

4.2.5 Analysis Approaches

Understanding how code evolves over multiple iterations is essential for assessing the consistency and effectiveness of iterative refinements. In the context of our study, we investigate whether refinements introduced by ChatGPT follow a coherent progression, whether they exhibit signs of inconsistency (e.g. back-and-forth modifications), and whether different refinement paths converge toward a common structure. To systematically analyze these aspects, we consider three complementary approaches (see Figure 4.7):

Horizontal Approach: This approach compares different versions of the same variant over time. Instead of only considering directly consecutive versions, we compare each version with *all* preceding ones. This allows us to track how a particular variant evolves through iterative refinements, highlighting the nature and consistency of modifications applied by ChatGPT across different versions. If versions that are further apart show higher similarity than those that are closer together, this may indicate back-and-forth modifications.

Figure 4.6 schematically illustrates the importance of comparing non-consecutive versions. Let v_0, v_1, and v_2 denote three consecutive versions of a snippet. The LLM introduces changes from v_0 to v_1, which are subsequently reversed from v_1 to v_2. As a result, the similarity between consecutive versions is $\text{sim}(v_0, v_1) = \text{sim}(v_1, v_2) = s < 1$, while the non-consecutive versions are identical, $\text{sim}(v_0, v_2) = 1$. Comparing only consecutive versions would therefore fail to detect the equivalence between v_0 and v_2.

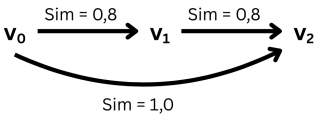

Figure 4.6: Schematic Visualization of SimScore Evolution in Back-and-Forth Modifications with v0 == v2 != v1

Vertical Approach: In this approach, code variants are compared within the same version number. This enables an analysis of how different refinements for the same base code differ at a given stage, revealing potential diversity in the refinement strategies generated by ChatGPT.

Combined Approach: This approach investigates whether different code variants gradually converge toward an 'optimal variant' as the refinement process progresses (i.e. across increasing version numbers). By analyzing both horizontal and vertical dimensions, this approach provides insights into whether the refinement process leads to a consensus on best practices or if structural differences persist over multiple iterations.

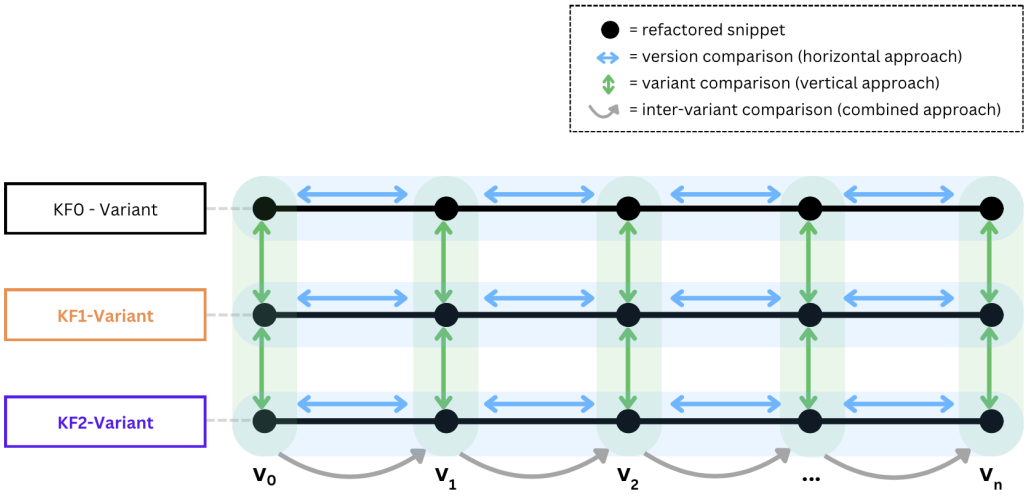

Figure 4.7: Overview of the three comparison approaches (non-consecutive comparisons omitted for readability)

Chapter 5

Results

This section presents the results of the main experiment, following the methodology outlined in Chapter 4. We begin by reporting absolute structural metrics to provide a descriptive baseline of the code snippets across refinement iterations. Subsequently, we analyze various aspects of code evolution, including the nature and distribution of modifications, as well as trends in structural convergence and stability. The results are organized according to the research questions defined earlier.

5.1 Evolution of Iterative Refinements (RQ1)

To address RQ1: *"How do iterative refinements by ChatGPT evolve when provided with a code snippet that already adheres to best practices?"*, we investigate how the model behaves when applied to code that requires little to no improvement. This setting allows us to examine whether ChatGPT merely preserves well-structured input, introduces unnecessary modifications, or converges toward a stable representation over multiple refinement steps. To this end, we focus on the original code snippets (variant KF0), which already follow established best practices in naming, commenting, and formatting. Each snippet is iteratively refined across five rounds using the base prompt pKF0 (*"Refactor this code for improved readability."*). Analyzing this process enables us to characterize the dynamics of ChatGPT's refinements when no substantial restructuring is required and to determine whether stability or persistent micro-modifications dominate over time.

5.1.1 Absolute Code Metrics Across Iterations (KF0)

Figure 5.1 provides an overview of the **average structural properties** of all code snippets for variant KF0 based on prompt pKF0 and version v0.

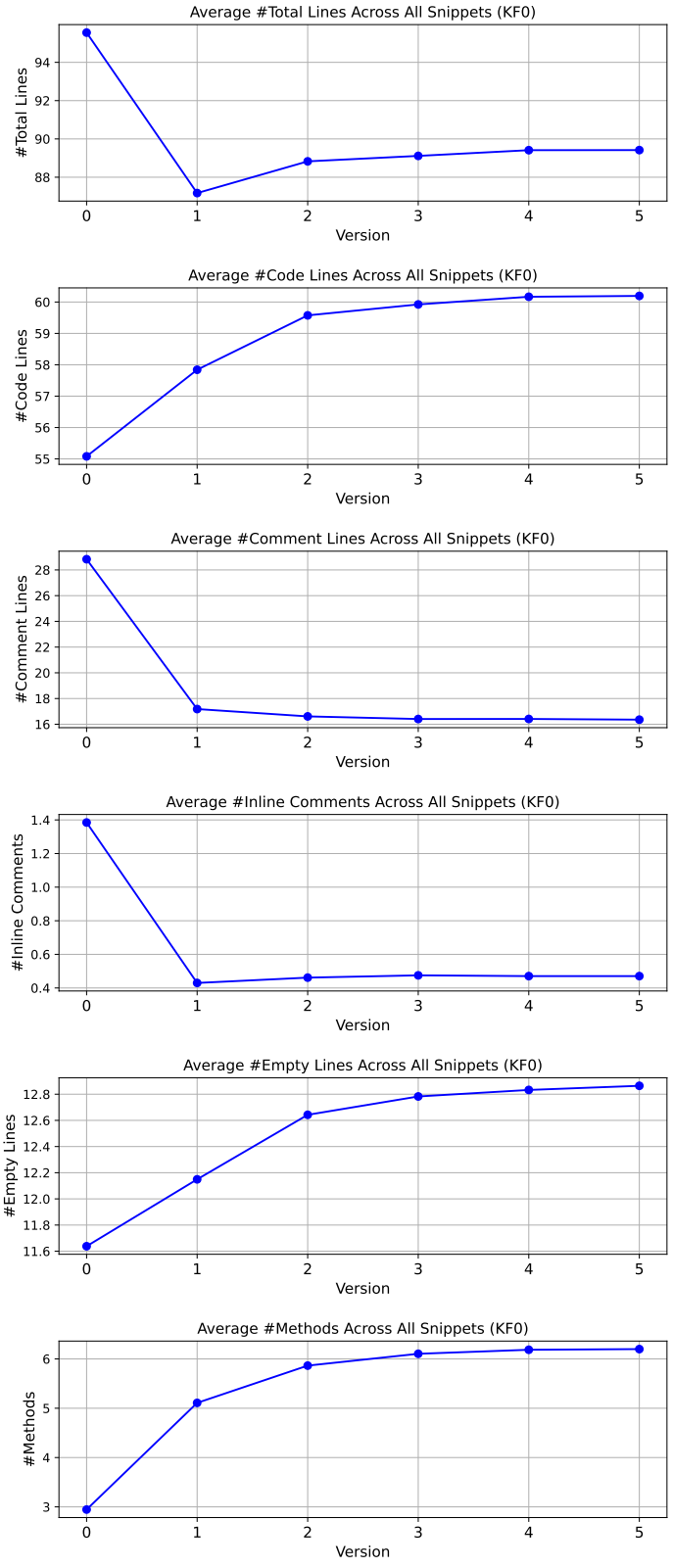

Figure 5.1: Average Absolute Metrics Across Refinement Versions (KF0, Prompt pKF0))

In the first iteration step (v0 → v1), a substantial reduction in total lines can be observed, driven primarily by a sharp decline in comment lines. The average number of comment lines decreases from approximately 28.5 to 17, while inline comments drop from 1.4 to below 0.5 per snippet. Meanwhile, the number of code lines increases steadily from version v0 to v5, rising from 55 to slightly above 60. A similar upward trend is evident in the number of empty lines and method declarations, both of which stabilize from version v3 onward.

5.1.2 Overall Change Dynamics Across Refinement Steps (KF0)

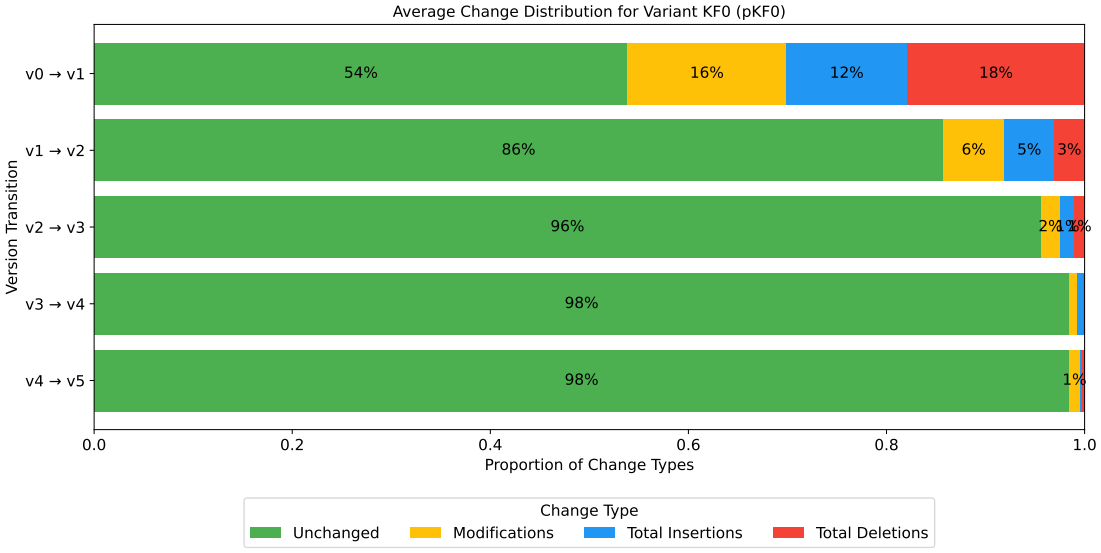

Figure 5.2: Proportional Distribution of Code Changes Across Iterations between two successive versions (KF0, Prompt pKF0)

Figure 5.2 shows the average distribution of change types across consecutive refinement steps for variant **KF0**. In the initial step (v0 → v1), 54% of the lines remain unchanged, while 16% are classified as modifications, 12% as insertions, and 18% as deletions. In the following iteration (v1 → v2), the proportion of unchanged lines increases to 86%, and all types of changes decrease accordingly. From v2 → v3 onward, the proportion of unchanged lines further increases, reaching 96%, 98%, and 98% in the final steps, respectively. All remaining changes are marginal, each accounting for less than 2% of the total lines. Overall, the data indicate a consistent decrease in the extent of changes across refinement rounds for the unaltered base variant.

Detailed Breakdown of Modification Types (KF0)

To further examine the nature of the observed changes, Figure 5.3 provides a more fine-grained breakdown of the change types for variant KF0. While the overall proportion

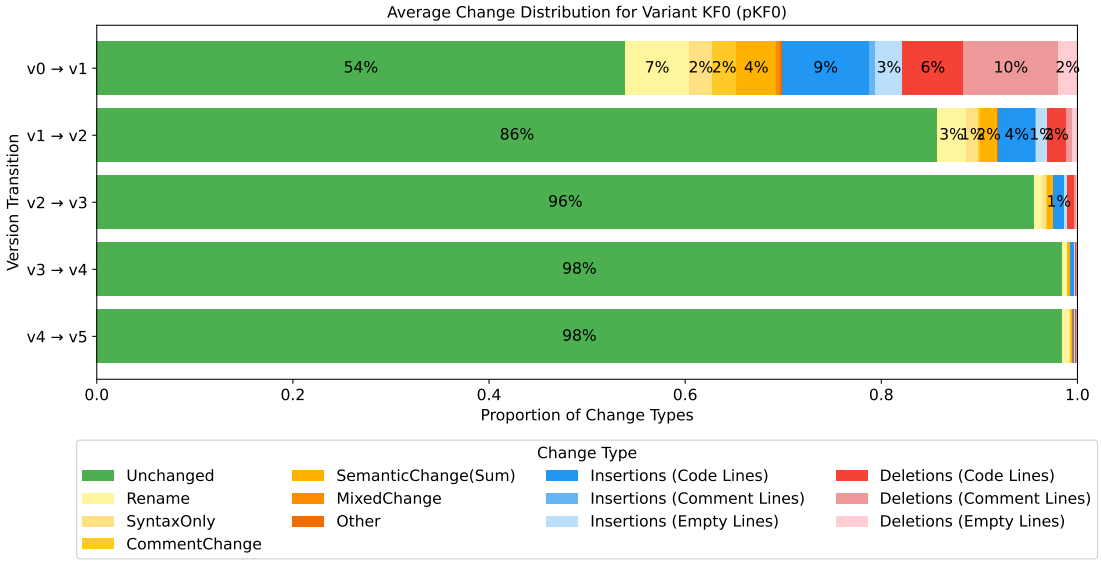

Figure 5.3: Detailed Proportional Distribution of Code Changes Across Iterations (KF0, Prompt pKF0)

of unchanged lines follows the same trend as in Figure 5.2, the extended visualization differentiates between specific modification categories as well as line-level insertions and deletions. In the first refinement step (v0 → v1), the largest individual change types include renaming (7%), code insertions (9%), and deletions of comment (10%) and code lines (6%). Semantic changes (4%) and comment changes (2%) are also present, alongside minor proportions of syntax-only modifications (2%) and mixed changes (≤1%).

From v1 → v2, the changes become more targeted and balanced: renaming (3%), syntax-only changes (1%), and semantic changes (2%) are still observed, while insertions and deletions - especially of code lines - occur at lower frequencies. In later iterations (v2 → v3 through v4 → v5), most categories fall below the 1% threshold. The vast majority of lines remain unchanged, and no single change type dominates, suggesting a stabilization of the refinement process.

Modification Flow Across Iterations (KF0)

To better illustrate the progression and persistence of individual change types over the course of the refinement process, Figure 5.4 presents a Sankey diagram for variant KF0 under prompt pKF0. While the stacked bar plots (Figures 5.2 and 5.3) show only aggregated proportions per version pair, the Sankey plot enables a more granular view of how specific modification types evolve across all five iterations.

Rename operations and semantic changes dominate the early refinement stages and persist across multiple rounds, while comment changes, syntax-only changes, and mixed

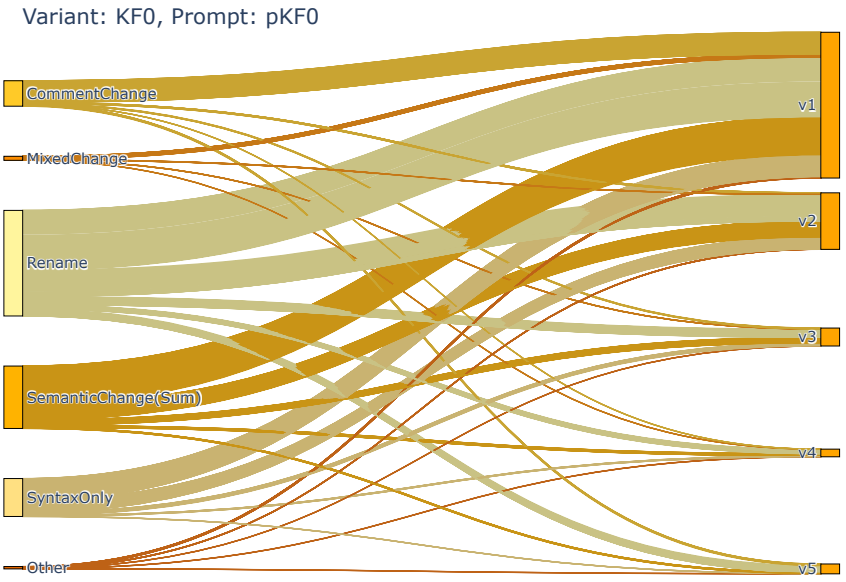

Figure 5.4: Modification Flow Across Iterations (KF0, Prompt pKF0). Each flow represents a particular type of code modification (e.g. *Rename*, *SemanticChange*, *SyntaxOnly*), connecting to the corresponding target version

changes occur less frequently and are more evenly distributed. Overall, the visualization highlights that even though the proportion of modifications becomes marginal in later iterations, a diverse set of change types remains active throughout the refinement process.

5.1.3 Pairwise Similarity Analysis Across Refinements (KF0)

While the Sankey plots provide a detailed view of how specific modification types propagate across refinement steps, they do not reveal whether these modifications lead to consistent structural convergence or occasional reversals. To address this aspect, we complement the change-type analysis with pairwise similarity scores between all versions. Visualizing these scores in a heatmap allows us to assess convergence trends across the refinement process and to identify potential back-and-forth modifications.

Figure 5.5 shows a heatmap of average similarity scores between all pairwise combinations of versions for variant KF0, based on prompt pKF0. The results show a steady increase in similarity as the refinement process progresses. The similarity between adjacent versions increases from 0.89 (v0 $\rightarrow$ v1) to 0.98 (v4 $\rightarrow$ v5). Non-consecutive comparisons (e.g., v0 $\rightarrow$ v5 = 0.87) also show high similarity, indicating that changes accumulate incrementally and consistently over iterations.

Notably, the similarity between early and late versions (e.g., v1 $\rightarrow$ v4 = 0.95, v2 $\rightarrow$ v5 = 0.94) suggests that structural convergence begins early in the refinement process and

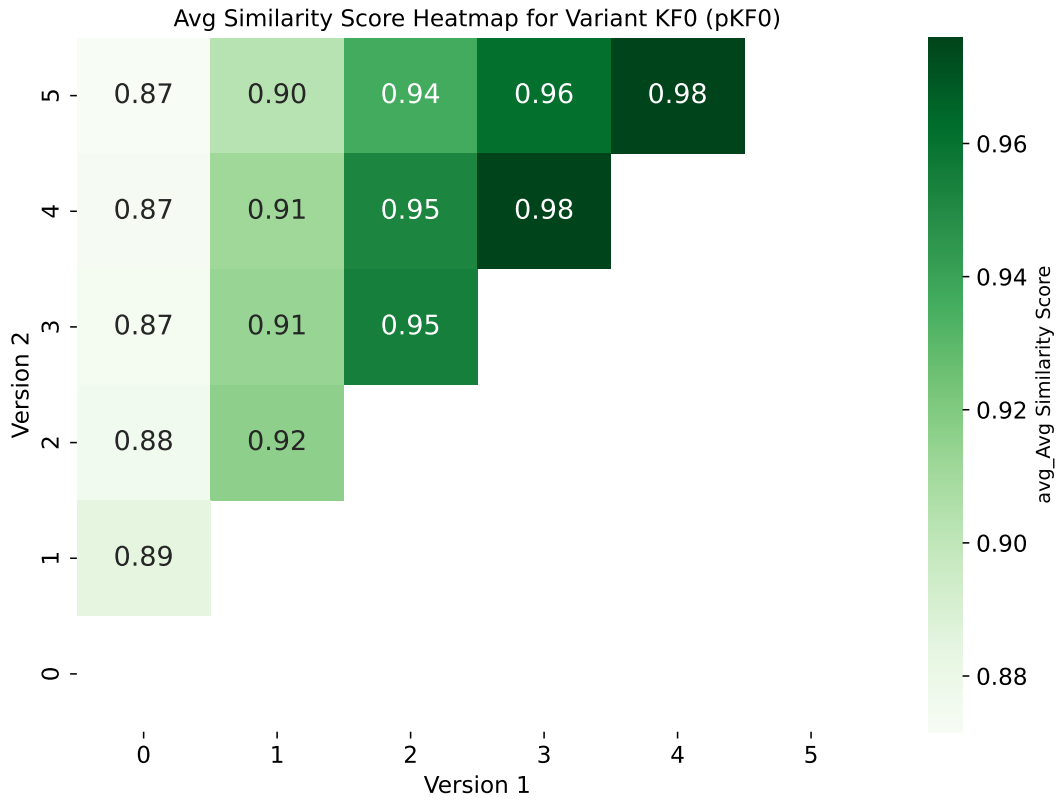

Figure 5.5: Pairwise Similarity Heatmap Across Refinement Versions (KF0, Prompt pKF0).
Each cell indicates the average similarity of modified lines between the respective version pair, using the metric described in Section 4.2.4. Only the upper triangle is populated, as the comparison is directional (from earlier to later versions).

stabilizes in later rounds. However, none of the pairwise comparisons ever reach a perfect similarity score of 1.00.

Summary of Findings for RQ1

The results provide partial support for the content hypothesis (H1). While the iterative refinements of already well-structured code (KF0) largely preserved the original structure and displayed a clear convergence trajectory, the model did not strictly limit itself to only necessary modifications. Instead, the initial iteration introduced a noticeable restructuring phase characterized by reductions in comments, renaming operations, and additional semantic and syntactic adjustments. Subsequent refinements stabilized quickly, with steadily increasing similarity scores and only marginal changes. However, the absence of perfect similarity values (1.00) indicates that the model continues to apply small modifications even when the input already conforms to best practices, suggesting a tendency toward over-refinement.

5.2 Convergence Across Code Variants (RQ2)

To address RQ2: *"When multiple variations of the same code snippet - each modified with respect to a single key factor of code understandability - are iteratively refined, do the refinements converge after a certain number of iterations?"*, we extend the analysis beyond the well-structured baseline (KF0) and investigate how ChatGPT handles systematically altered input. This setup allows us to test whether the refinement process is sensitive to different starting conditions or whether it ultimately normalizes distinct variants toward a common structural representation. For this purpose, we generate two additional variants from each original snippet: one with obfuscated identifiers (KF1) and one without comments (KF2). All three variants (KF0-KF2) are then subjected to five refinement rounds using the same base prompt pKF0 (*"Refactor this code for improved readability."*). This design enables us to disentangle the effect of initial structural differences from the general convergence dynamics of iterative refinements.

5.2.1 Baseline: Variant Creation and Differences

Before investigating the convergence of iteratively refined code variants, we first examine how distinct the variants in their initial version v0 are. As described in the methodology, we derived two variants from the same base snippet but modified with respect to a specific key factor of code readability: naming conventions (KF1), and presence of comments (KF2).

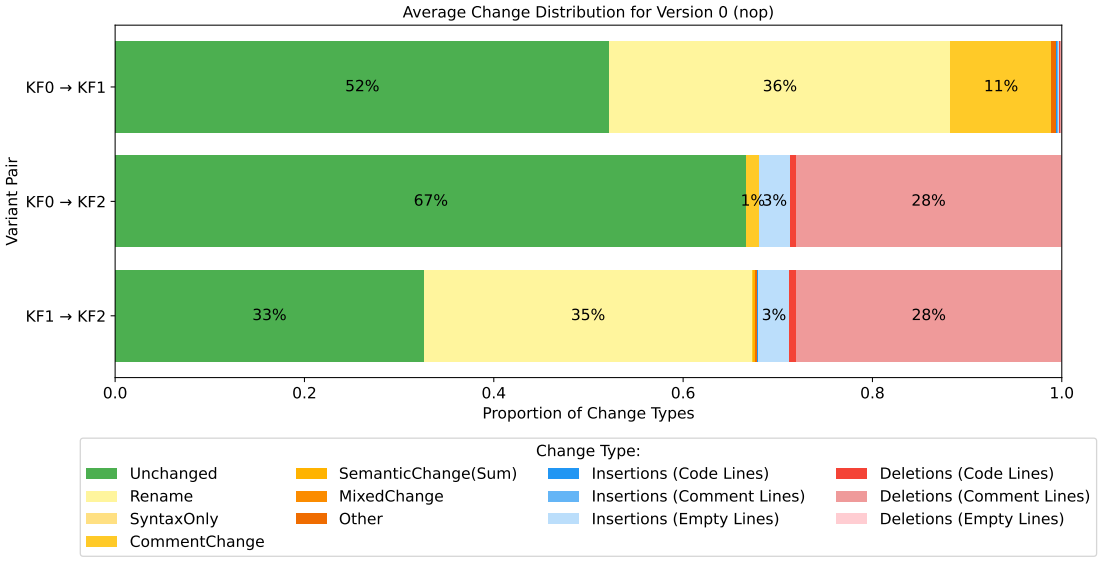

Figure 5.6: Detailed Change Composition Between KF0, KF1, and KF2 at Version 0 (nop = "no prompt")

Figure 5.6 shows the detailed change distribution between the three code variants **KF0**, **KF1**, and **KF2** at version v0. The plot illustrates the proportion of changed lines across categories, averaged over all snippets. The transformation from **KF0** to **KF1** consists of systematically obfuscating all class, method, and variable names. As expected, this results in a dominant proportion of *Rename* operations (36%), alongside *CommentChange* (11%), which arise from identifier substitutions within code comments. The distinction between these two categories is due to the matching algorithm: when a comment line (block or inline) is changed solely due to renamed identifiers, the corresponding modification is classified as a comment change.

In the case of **KF0** → **KF2**, all code comments were removed, while keeping the functional code intact. On average, comments made up approximately 28% of each snippet, which now appear as *Deletions (Comment Lines)* in the comparison. An additional 1% are classified as *CommentChange*, primarily due to inline comment removals that did not match corresponding lines in KF0. Furthermore, about 3% of all lines were replaced with empty lines to preserve code structure and maintain visual alignment. This choice was made deliberately, as we consider the insertion of empty lines to be a more appropriate structural substitute for removed comments, especially in terms of readability and comparability.

The third row in the plot (**KF1** → **KF2**) reflects the cumulative structural impact of both key factors - renaming and comment removal. On average, 77% of all lines are affected: 35% by renaming, 28% by comment deletions, and 3% are empty line insertions. Only 33% of the lines remain unchanged. Note that no additional *CommentChange* entries could be recorded in this comparison, as the KF2 variant contains no comments, thereby eliminating the possibility of a matched modification in this category. Across all comparisons, only negligible residuals of other change types ($\leq$1%) appear, which reflect inherent limitations of the parsing process.

5.2.2 Absolute Code Metrics Across Iterations (KF1 + KF2)

Figure 5.7 shows the evolution of absolute structural metrics across all refinement iterations (v0 to v5) for the three code variants **KF0**, **KF1**, and **KF2**, each refined using prompt `pKF0`. The plots report average values per snippet for six key metrics: total lines, code lines, comment lines, inline comments, empty lines, and number of methods.

At version v0, **KF0** and **KF1** contain 95.2 total lines on average, respectively, while **KF2**, due to the removal of all comments, begins with only 70 lines. After the first refinement round (v1), all three variants converge toward a reduced total line count: KF0 drops to 88.0, KF1 to 88.2, and KF2 remains constant at 70. From version v2 onward,

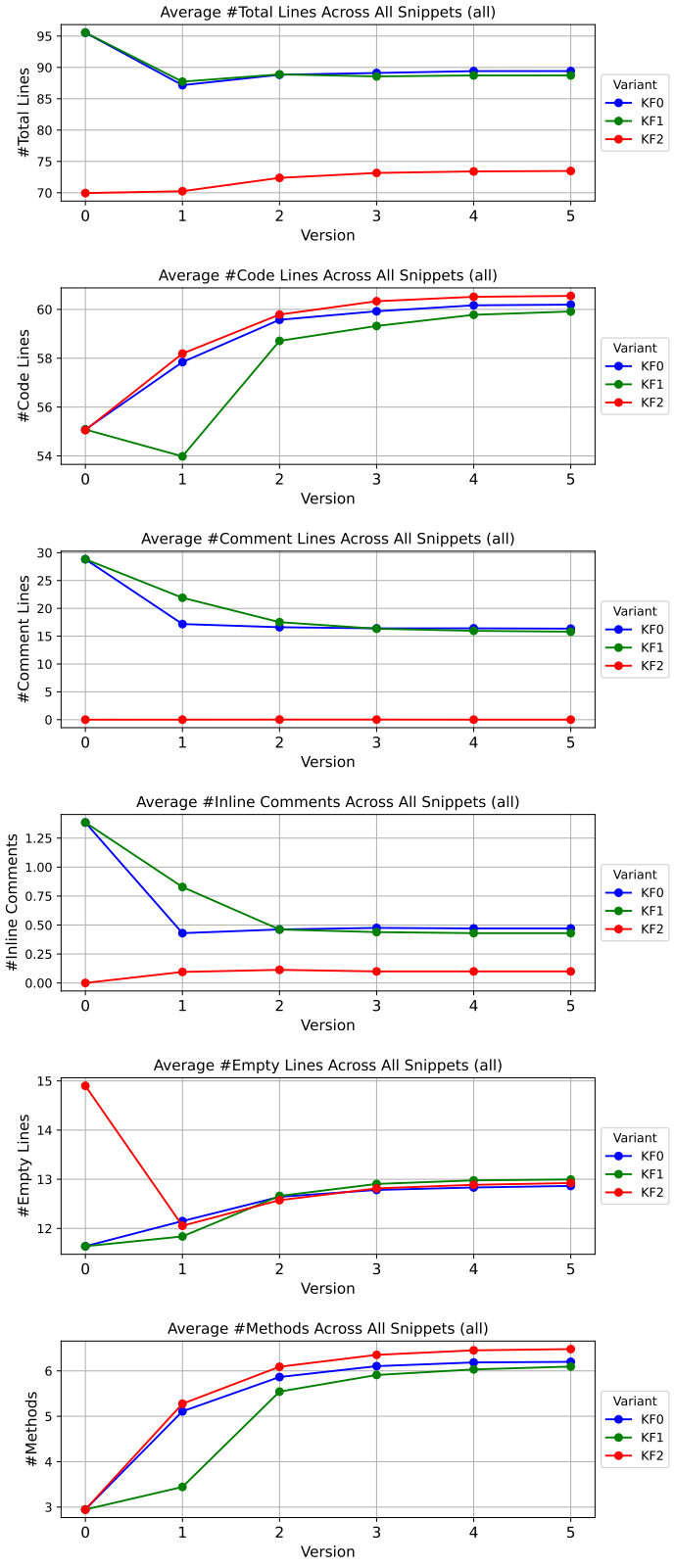

Figure 5.7: Average Absolute Metrics Across Refinement Steps for KF0–KF2 (Prompt pKF0)

total line counts increase slightly and stabilize, with KF0- and KF1-variants reaching approximately 89 lines by v5, while the KF2-variant reaches approximately 74.

The number of code lines increases steadily across all variants. All variants begin with 55.0 code lines and reach approximately 60.5 in v5. As expected, comment lines vary significantly by condition. KF0 and KF1 start with 29.0 comment lines. Both decrease over time to converge at around 17.2 lines by v5. KF2 contains no comments throughout all iterations. Inline comments follow a similar pattern: KF0 and KF1 start at 1.3, and drop to 0.45 by v5. KF2 contains virtually no inline comments at any iteration, remaining close to 0.1 throughout. Empty lines increase slightly across all conditions (after the initial drop in v1 for KF2 to the level of KF0 and KF1). KF0 and KF1 start at 11.7, KF2 from 14.9, but from there the variants don't diverge anymore, reaching a final count of approximately 13 empty lines in v5.

Lastly, the average number of methods increases for all three variants in the early iterations. All variants naturally start with the same number of methods (3.0). KF0 grows to 6.2 at v5. KF1 reaches 6.1, while KF2 increases to 6.3. All three variants show near-identical method counts by the final iteration, further indicating structural convergence.

5.2.3 Overall Change Dynamics Across Refinement Steps (KF1 + KF2)

Having established the structural development of each variant in terms of absolute metrics, we now turn to a more granular view of how the underlying changes unfold across iterations. While the previous plots captured overall trends in code size and composition, the following analysis focuses on the proportion and nature of line-level changes, i.e. how much of the code is modified, inserted, or deleted at each refinement step. This perspective allows us to assess the degree and timing of stabilization during the iterative process.

Figure 5.8 illustrates the average proportional distribution of changes across all refinement iterations for the code variants **KF1** and **KF2**, both processed using prompt `pKF0`. The stacked bar plots differentiate between unchanged lines, modifications, insertions, and deletions, providing a high-level view of how each version evolves over time.

In the case of **KF1**, the first refinement iteration (v0 → v1) introduces substantial changes, with only 40% of lines remaining unchanged. Modifications account for 41% of changes, while insertions and deletions contribute 6% and 13%, respectively. The following iterations show a marked trend toward stabilization: the proportion of unchanged lines increases to 71% in v1 → v2, 92% in v2 → v3, and 98% in the final step (v4 → v5).

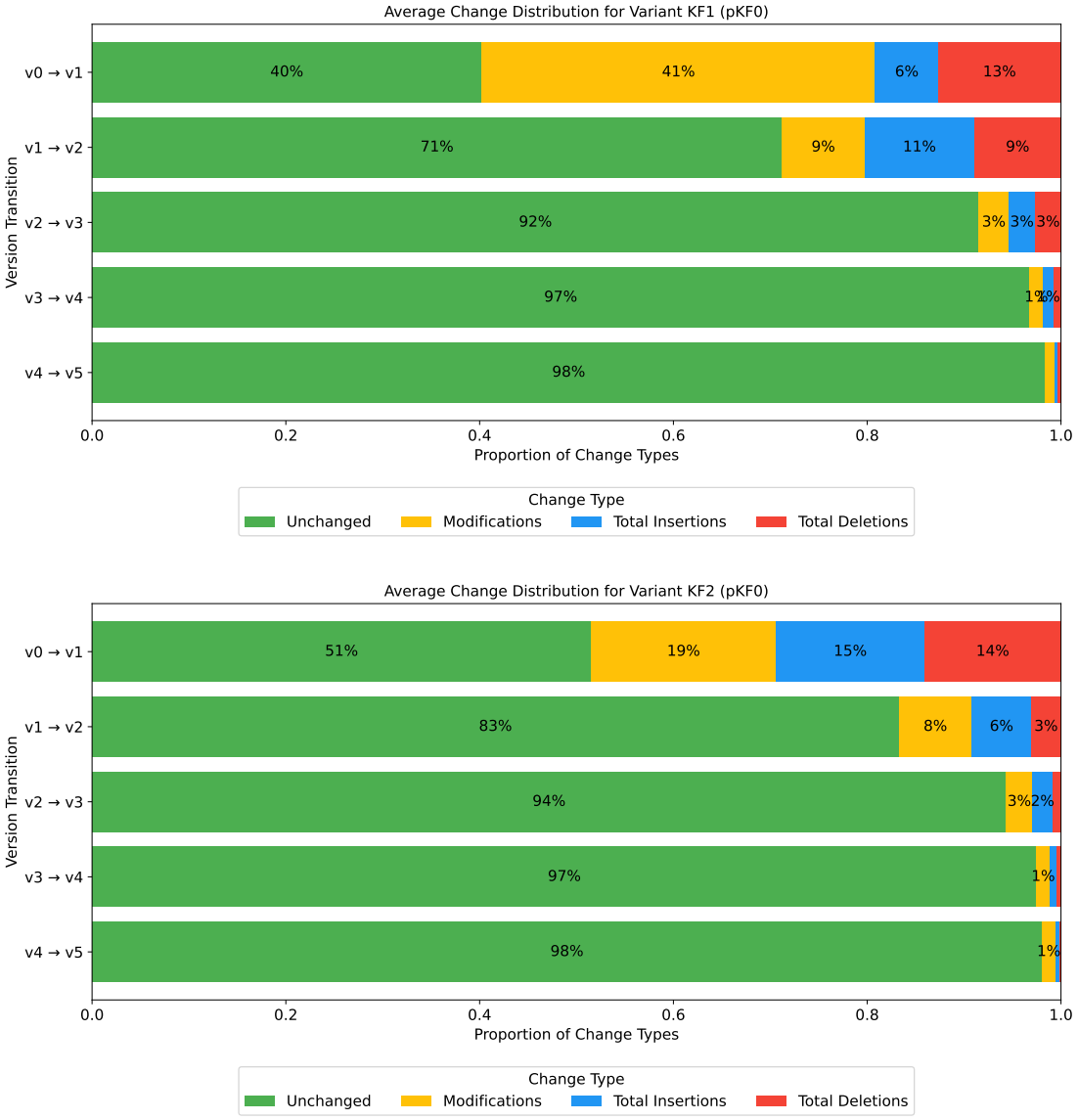

Figure 5.8: Comparison of average proportional change distributions across refinement iterations for variants KF1 (top) and KF2 (bottom) under prompt pKF0

Correspondingly, modifications and structural operations (insertions/deletions) diminish steadily.

The trend is similar for **KF2**, although the first iteration (v0 → v1) shows a slightly higher preservation rate, with 51% of lines unchanged. Modifications (19%), insertions (15%), and deletions (14%) are relatively balanced in this early step. In subsequent iterations, the proportion of unchanged lines increases to 83% (v1 → v2), 94%, 97%, and eventually 98% at v5, closely matching the trajectory observed for KF1.

Overall, both variants exhibit a convergence pattern similar to that of KF0 (see Figure 5.2), with the majority of structural adjustments occurring in the first two refinement rounds.

The final iterations involve minimal change activity, suggesting that the model reaches a stable representation of the code regardless of the starting condition.

5.2.3.1 Detailed Breakdown of Modification Types (KF1 + KF2)

While the aggregated stacked bar plots provide an overview of the general change dynamics, a more fine-grained breakdown is required to understand the nature of the underlying modifications.

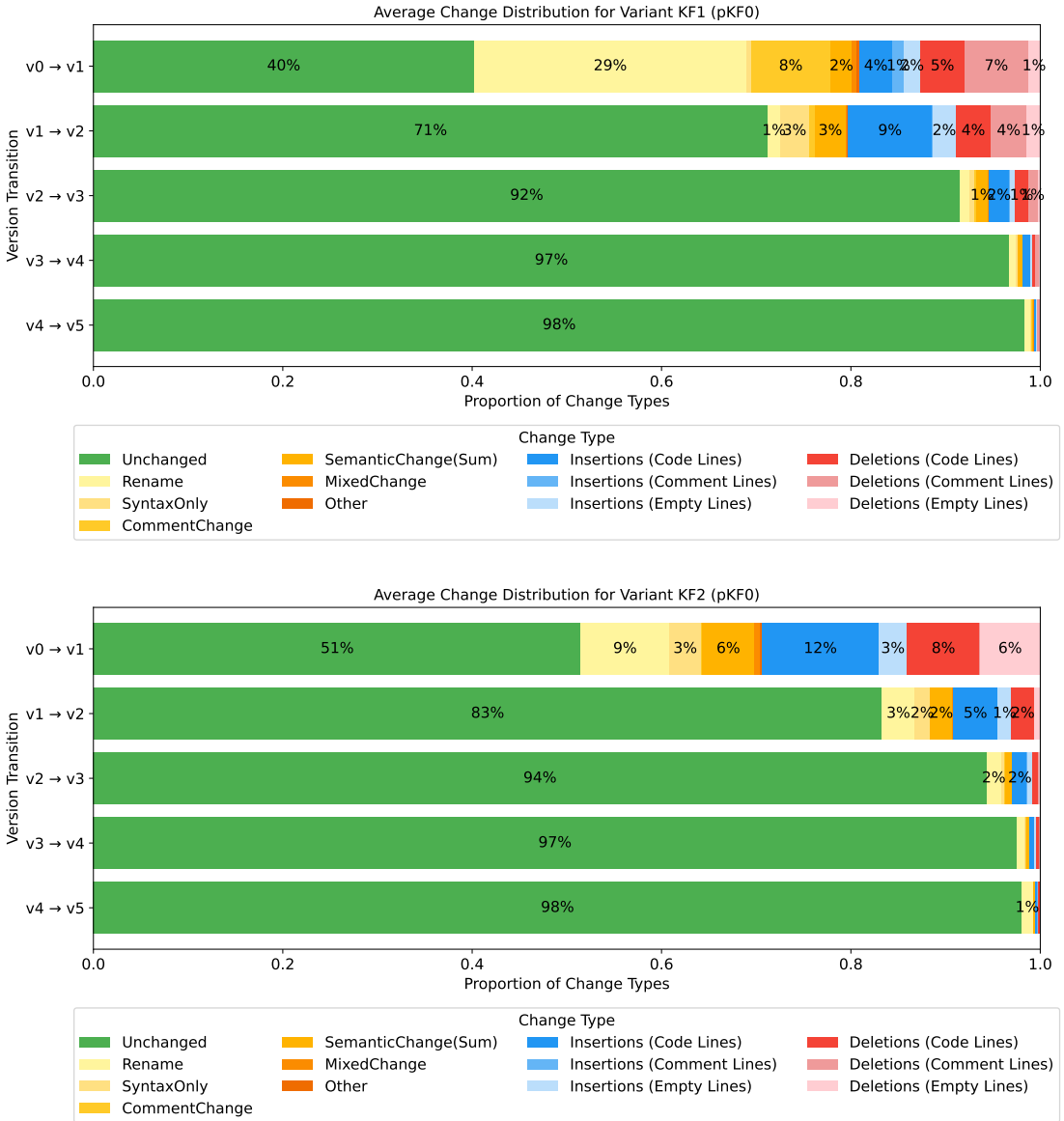

Figure 5.9: Comparison of detailed average proportional change distributions across refinement iterations for variants KF1 (top) and KF2 (bottom) under prompt pKF0

Figure 5.9 disaggregates the “modifications” category into the previoulsy introduced concrete change types - such as renaming, semantic restructuring, syntax-only edits,

or changes to comments - alongside different forms of insertions and deletions. This allows for a more nuanced assessment of how each refinement step contributes to the convergence behavior observed earlier.

As hypothesized, for the KF1-variant the largest modification type in the first refinement step (v0 → v1) consists of **Rename** operations (29%), followed by comment changes (8%) and semantic changes (2%). This is consistent with the intended purpose of KF1, which targets identifier renaming. Importantly, comment changes accounts for changes to lines where variable names were also present in comments - either inline or block-level - which were obfuscated to avoid biasing the model during future renaming. Insertions and deletions of comment lines (1% and 7%, respectively), empty lines (2% and 1%, respectively) as well as contributions from code insertions (4%) and deletions (5%), round out the change spectrum.

In the second step (v1 → v2), the Rename portion drops to 1%, comment change to 3%, and Insertions (Code + Comment) and Deletions (mostly Comments) each to under 5%, while the Unchanged portion rises to 71%. From the third iteration onward, structural edits become increasingly rare: v2 → v3 contains only minor contributions ($\leq$2%) across all categories, and v3 → v5 stabilizes at 97–98% unchanged code, similar to the convergence trend observed in KF0.

A similar pattern is visible for the KF2-variant, albeit with a different emphasis. The initial refinement (v0 → v1) shows 14% of lines deleted, consisting of code lines (8%), and empty lines (6%), as well as insertions of empty lines (3%), and code lines (12%). The remaining change types include renames (9%), semantic changes (6%), and syntax-only changes (3%). The absence of comment changes and comment deletions reflect the absence of comments in the initial version v0 of the KF2-variant. In subsequent steps, the proportion of unchanged lines increases to 83% in v1 → v2 and eventually converges to 98% by v5. From v2 onward, changes are minimal and mostly limited to isolated insertions or minor renaming modifications (1–2%).

5.2.3.2 Modification Flow Across Iterations (KF1 + KF2)

To complement the proportional change distribution plots presented above, we again include Sankey diagrams to better visualize the flow and persistence of modification types across iterative refinement steps. While stacked bar plots offer aggregated proportions per version pair, the Sankey view reveals how individual types of changes propagate over time - and to what extent they persist or fade. This is particularly relevant in the context of RQ2, where multiple code variants are initialized with targeted structural differences and may exhibit distinct convergence trajectories.

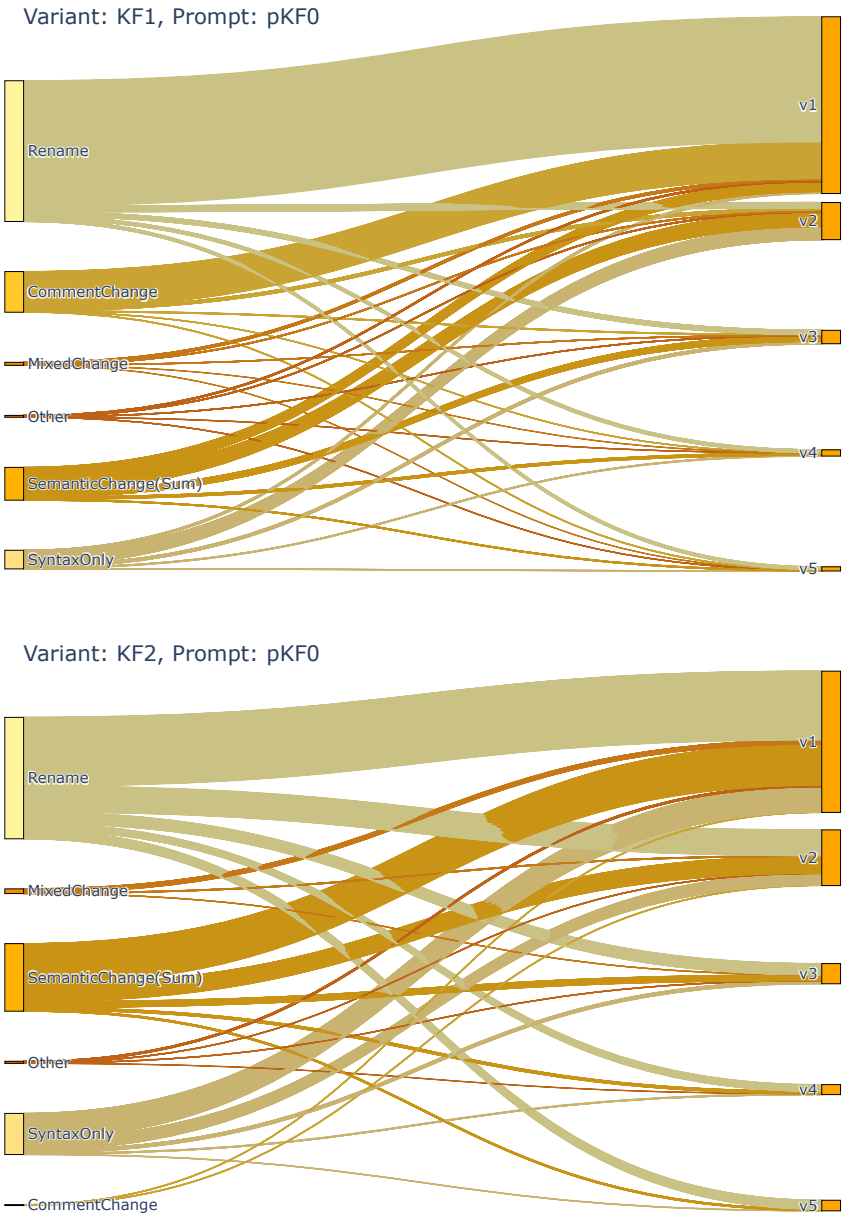

Figure 5.10: Comparison of modification flow across iterative refinements for the two variants

Figure 5.10 compares the modification trajectories for variants KF1 and KF2. In both cases, the initial iteration (v0 → v1) is dominated by the systematic renaming of identifiers. KF2 further exhibits an initial concentration of semantic changes and syntax-only changes.

Beyond the second iteration (v2 → v3 and onward), the distributions in both variants increasingly converge toward a narrow set of change types, with most activity limited to small-scale semantic changes, minor rename adjustments, and occasional syntax-only corrections. Overall, the Sankey diagrams reinforce the convergence trend already

observed in the stacked bar plots and illustrate how different initial manipulations affect the temporal dynamics of refinement.

5.2.4 Pairwise Similarity Analysis Across Refinements (KF1 + KF2)

Following the structural and semantic analysis of code changes, we now examine how similar the generated code versions are to one another throughout the iterative refinement process. For this purpose, we analyze the average pairwise similarity scores between all versions from v_0 to v_5 for each variant. This measure complements the change-based visualizations by capturing a more holistic view of transformation stability across iterations.

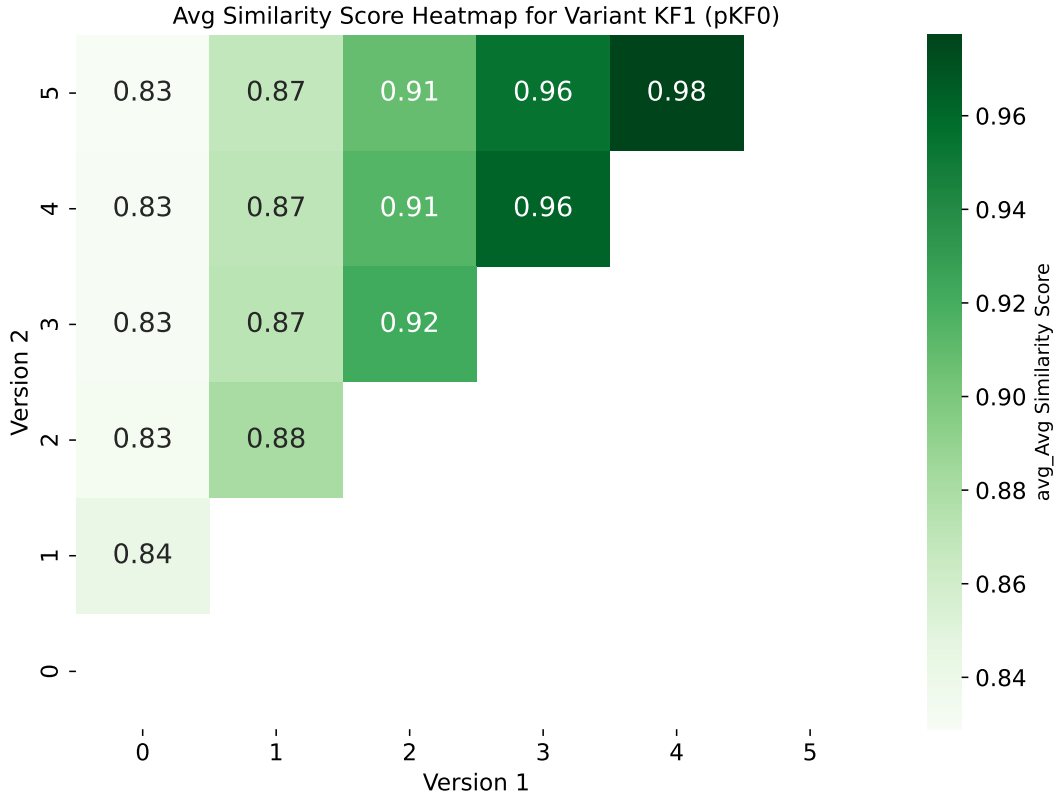

Figure 5.11: Modification Flow Across Iterations (KF1, Prompt pKF0)

Figures 5.11 and 5.12 presents the average similarity score heatmaps for the variants KF1 and KF2, both processed under prompt pKF0. Each heatmap cell indicates the average similarity score between two versions of the same snippet, computed across the entire set of evaluated snippets.

For both KF1 and KF2, we observe a consistent pattern of increasing similarity as versions progress. However, the dynamics differ slightly between the two: KF1 shows a steady increase in similarity scores, beginning at 0.84 (v0 $\rightarrow$ v1) and reaching up to 0.98

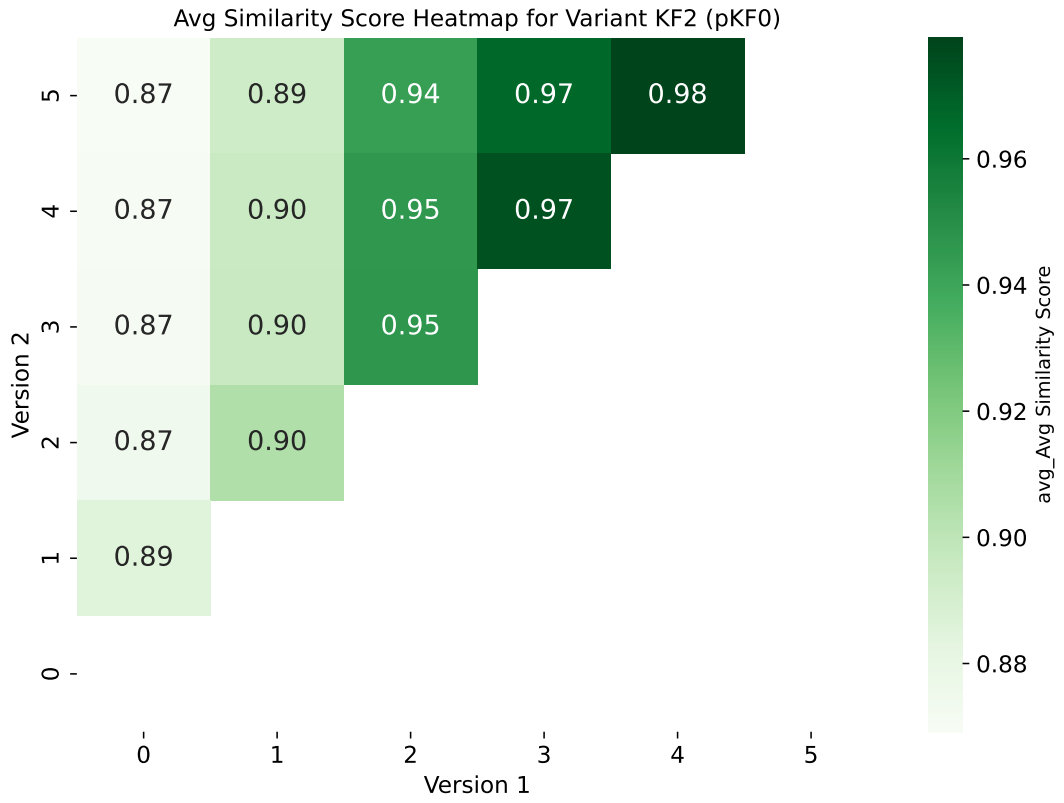

Figure 5.12: Modification Flow Across Iterations (KF2, Prompt pKF0)

(v4 → v5). The scores from v2 onwards exhibit particularly high similarity, e.g., 0.91 (v2 → v3) and 0.96 (v3 → v4), suggesting that refinements stabilize significantly after the second iteration. This observation aligns with the earlier findings from the stacked bar plots, which already indicated that most structural changes occur early, followed by incremental refinements. KF2, on the other hand, starts from a higher similarity baseline, with a v0 → v1 similarity of 0.89. This may reflect the more limited nature of the initial transformation (comment removal) relative to KF1 (rename refactoring). From v2 onwards, the similarity scores climb even faster and converge slightly earlier than in KF1. For example, v2 → 3 already reaches 0.95, and v3 → v4 scores at 0.97.

Overall, both variants exhibit convergence behavior, with increasing similarity scores indicating that the iterative refinement process becomes more stable over time. Notably, KF2 converges slightly faster than KF1, which aligns with the observation that its version transitions involve fewer structural alterations.

Similarity Score Evolution Across Variants

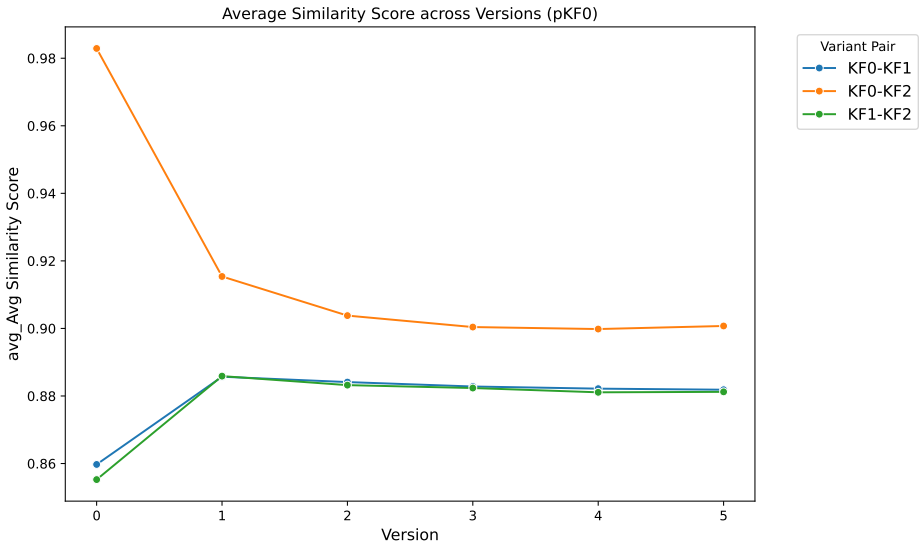

Figure 5.13: Development of average similarity scores across iterations for each variant pair (pKF0)

To complement the pairwise similarity heatmaps introduced earlier, Figure 5.13 illustrates how the average similarity score between each variant pair evolves over the five refinement iterations. This allows us to track whether and how the differences between variants persist or diminish throughout the refinement process.

At iteration 0 (i.e. the starting point), the three variant pairs exhibit expected differences:

- **KF0 and KF2** start out with the highest similarity score (0.98), due to both variants sharing identical code except for comment removal.
- **KF0 and KF1**, and **KF1 and KF2**, begin at lower values (0.86 and 0.85, respectively), reflecting the impact of identifier renaming.

However, across iterations, all variant pairs gradually **converge towards similar similarity scores**, ranging from approximately **0.88 to 0.9** by iteration 5. Most notably:

- The similarity between **KF0 and KF2** decreases significantly from 0.98 to around 0.9, reflecting the increasing structural changes introduced by the model that go beyond simple formatting or syntax-only edits.
- The **KF1–KF2** pair exhibits a slight upward trend from 0.85 to 0.88, closing in on the KF0–KF1 scores.

Ultimately, all three curves converge within a **narrow range of less than 3% difference**, indicating that despite their divergent starting points, the iterative refinements lead all variants to **similar structural representations** of the code. This provides further support for the convergence hypothesis formulated in RQ2: independently of initial transformations, the model tends to normalize and align the structure of code snippets over repeated prompts.

Summary of Findings for RQ2

The results consistently demonstrate convergence across all three code variants. Despite structural differences in their initial versions - particularly between KF1 (renamed identifiers) and KF2 (removed comments) - the iterative refinements rapidly reduce these differences. Structural metrics (code/comment/empty lines, number of methods) stabilize after the second iteration, and line-level analyses reveal that the majority of modifications occur early, followed by minimal adjustments in later rounds. Pairwise similarity scores confirm this trajectory: both within-variant comparisons (v0–v5) and cross-variant comparisons converge toward high similarity values, typically around 0.88–0.90 by the final iteration. Notably, KF2 reaches convergence slightly faster than KF1, but in the end all variants align to highly similar structural representations. Taken together, these findings suggest that the refinement process might be relatively robust to initial variations and tends to normalize code snippets toward a shared representation, thereby supporting the convergence hypothesis formulated in RQ2.

5.3 Impact of Explicitly Emphasizing Key Refinement Factors (RQ3)

To address RQ3: *"Do targeted refinements become more effective when the prompt explicitly emphasizes the key factor in question?"*, we conducted a comparative analysis of two prompt strategies designed to highlight distinct refinement objectives. Specifically, pKF1 explicitly directs the model towards renaming identifiers, while pKF2 emphasizes the improvement of comments. By analyzing pairwise similarity scores across successive refinements of the three code variants (KF0, KF1, KF2), we aim to determine whether the explicit emphasis on a refinement factor leads to more stable convergence, or whether it introduces recurring back-and-forth modifications. This section presents the observed refinement dynamics under both strategies, followed by a comparative assessment of their respective impacts on convergence behavior.

5.3.1 Absolute Code Metrics Across Iteration under different Prompts

To investigate the effect of explicitly prompting for a particular refinement factor, we compared the absolute structural metrics of the snippets across all three variants (KF0-KF2) under different prompt strategies. Specifically, we contrast the general readability prompt pKF0 with two targeted alternatives: pKF1, which emphasizes identifier naming, and pKF2, which emphasizes code comments.

Comparison pKF0 vs. pKF1: Figure 5.14 shows that explicitly instructing ChatGPT to focus on naming (pKF1) leads to noticeably different structural dynamics compared to the general prompt (pKF0). Across all snippet variants (KF0-KF2), we observe that metrics such as total lines, number of methods, and code lines remain relatively stable over the refinement iterations under pKF1. This contrasts with the behavior under pKF0, where structural changes (e.g. increases in method count and code lines) are more pronounced, particularly in the first two refinement steps.

Regarding comments, a notable trend is that comment lines are more preserved under pKF1 than under pKF0 in the variants that originally contain comments (KF0 and KF1). Comment Lines as well as inline comments decrease moderately from v0 to v2 and then plateau. For the KF2-variant (which lacks comments), the structure remains particularly stable under pKF1. There is minimal growth in lines of code or number of methods, and comment lines remain consistently at zero - demonstrating that the naming-focused prompt does not implicitly trigger the addition of documentation or structural embellishment.

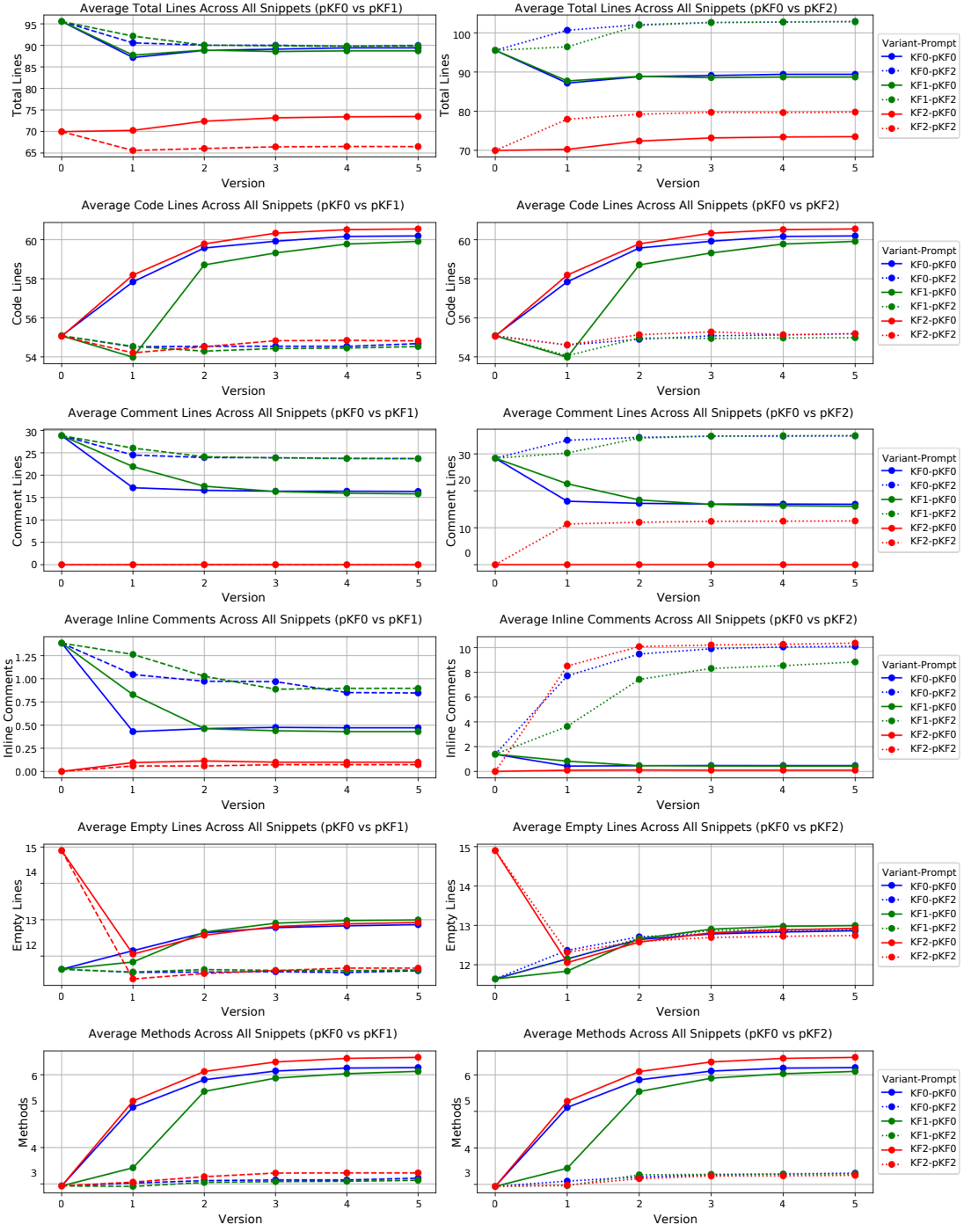

Figure 5.14: Comparison of Average Absolute Metrics Across Refinement Steps for KF0–KF2 for two different prompts (pKF1 and pKF2)

Comparison pKF0 vs. pKF2: On the other hand, we can see a markedly different dynamic when the prompt explicitly emphasizes comments (pKF2). While the number of code lines and methods remains relatively stable - mirroring the restrained restructuring seen under pKF1 - the behavior regarding comments changes substantially.

For the KF2-variant, which originally lacks comments, the number of comment lines

increases sharply from 0 to approximately 11 in the first refinement iteration (v0 → v1), and then remains stable. This pattern is reflected in the number of inline comments as well: it jumps from 0 to over 8 in v1, and further increases to nearly 10 in subsequent versions, stabilizing thereafter. Notably, these values represent the highest average inline comment counts across all variants and prompt combinations. The KF0 variant follows a similar trend, although the increase is slightly less pronounced. The KF1 variant exhibits a delayed and weaker increase.

These trends indicate that ChatGPT responds directly to the prompt by inserting (hopefullly meaningful) comments where appropriate, primarily during the first iteration. However, the lack of further increases in comment lines or inline comments in later versions suggests that the model refrains from artificially inflating documentation density, despite an explicit prompt emphasizing comments.

Another observation concerns the drop in empty lines in the KF2-variant from v0 to v1 under pKF2. This may be due to previously empty lines being filled with inline or block comments during the refinement. From v2 onward, the average number of empty lines slightly increases and then stabilizes, aligning with the levels observed in other variants.

5.3.2 Overall Change Dynamics Across Refinement Steps

To assess how a targeted prompts affect the nature and stability of iterative code changes, we analyze both the proportional distribution of change types across refinement steps and the longitudinal flow of modification categories over time. By analyzing the overall change dynamics between successive refinement iterations, we can assess whether prompts that emphasize specific key factors (i.e. naming or commenting) lead to more targeted and stable improvements, or whether they merely shift the types of changes being applied. Additionally, this perspective allows us to investigate whether such prompts accelerate convergence by reducing unnecessary structural modifications and promoting more consistent refinement behavior. The following subsections therefore analyze how the distribution of code changes evolves across versions when using prompt pKF1 (naming-focused) and prompt pKF2 (comment-focused), respectively.

KF0 - pKF1/pKF2: Figures 5.15 and 5.16 present the detailed distribution of change types across successive refinement steps for variant KF0 (the original code) under prompt pKF1 and pKF2, respectively. When we compare the two prompting strategies, we observe strikingly different trajectories of modification types. Under pKF1, which explicitly encourages identifier renaming, rename operations dominate across all iteration

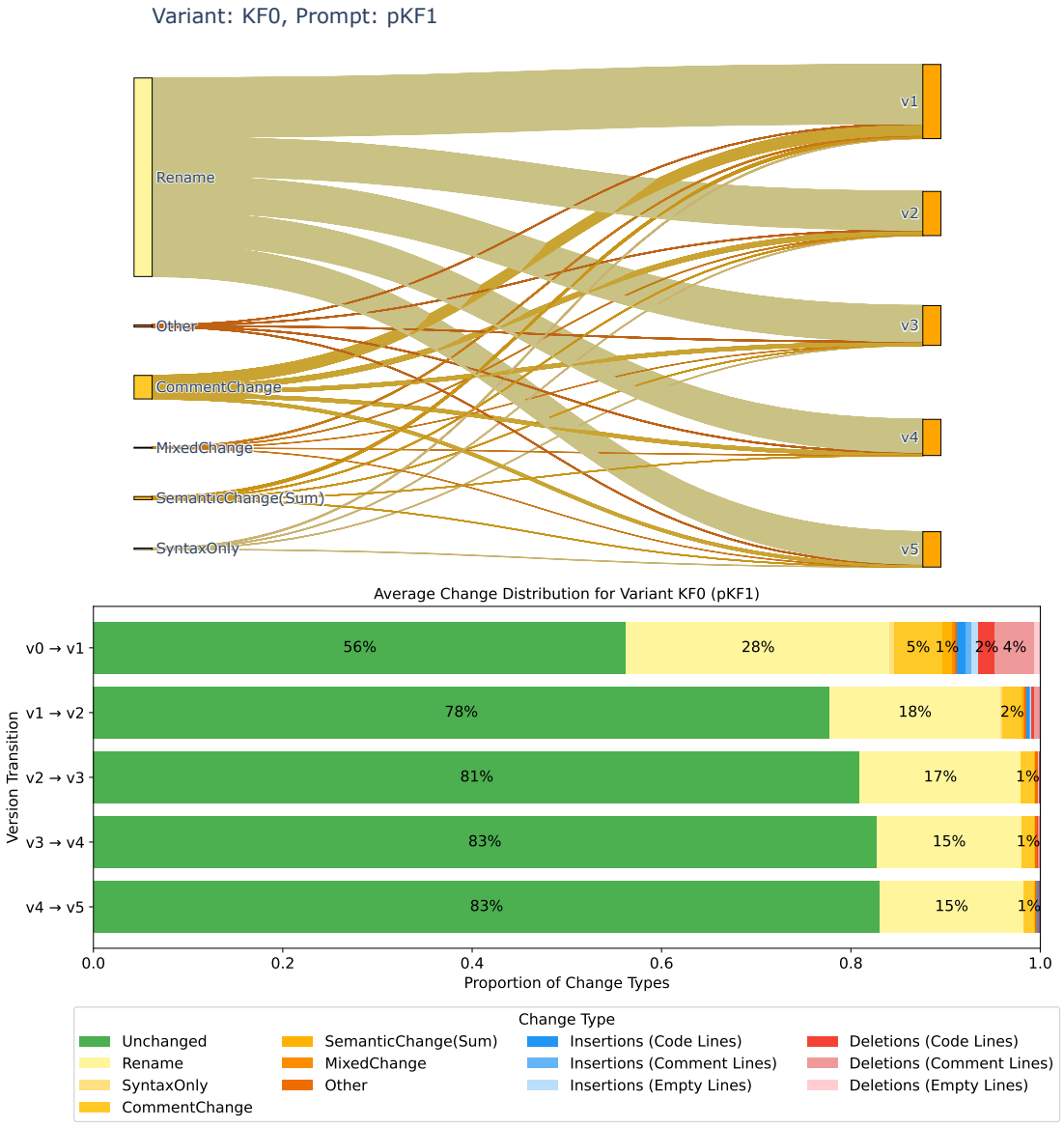

Figure 5.15: Distribution of Change Types for KF0 (pKF1)

steps. From the initial transition (v0 → v1) through to the later versions, renaming remains the primary form of change, while other modification types occur only marginally.

In contrast, pKF2, which emphasizes comment changes, produces a very different pattern: the vast majority of changes concentrate in the first transition, where comment modifications account for the bulk of activity. After this initial step, subsequent iterations show only minimal modifications, and by the final transition ($v4 \rightarrow v5$), the process essentially stabilizes with no observable changes.

KF1 - pKF1/pKF2: Figures 5.17 and 5.18 present the detailed distribution of change types across successive refinement steps for variant KF1 (the original code with obfuscated

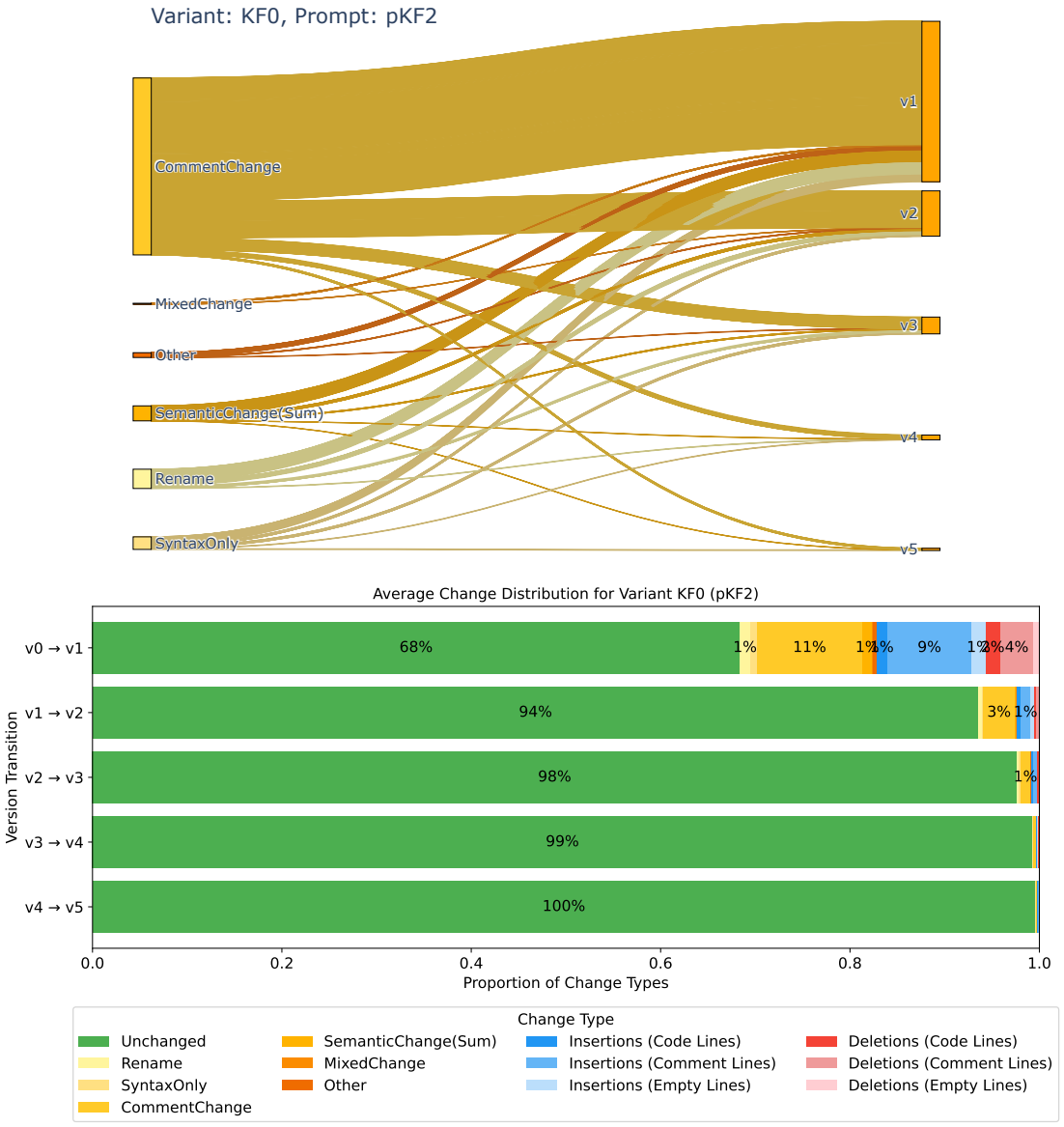

Figure 5.16: Distribution of Change Types for KF0 (pKF2)

identifiers) under prompt pKF1 and pKF2, respectively. Again, we find distinct effects of the two prompting strategies. Under pKF1, rename operations dominate from the very first iteration and persist across subsequent transitions, mirroring the pattern observed in KF0. It is noteworthy that, unlike with the unguided prompt pKF0, the modifications in KF1 do not converge rapidly but instead show a persistent share of changes that decreases only slowly across iterations. Renaming remains the dominant type of modification throughout, while smaller proportions of comment changes and other categories continue to appear in later steps.

In contrast, pKF2 produces a more heterogeneous distribution of changes: although the prompt emphasizes comments, the first transition (v0 → v1) still contains a large

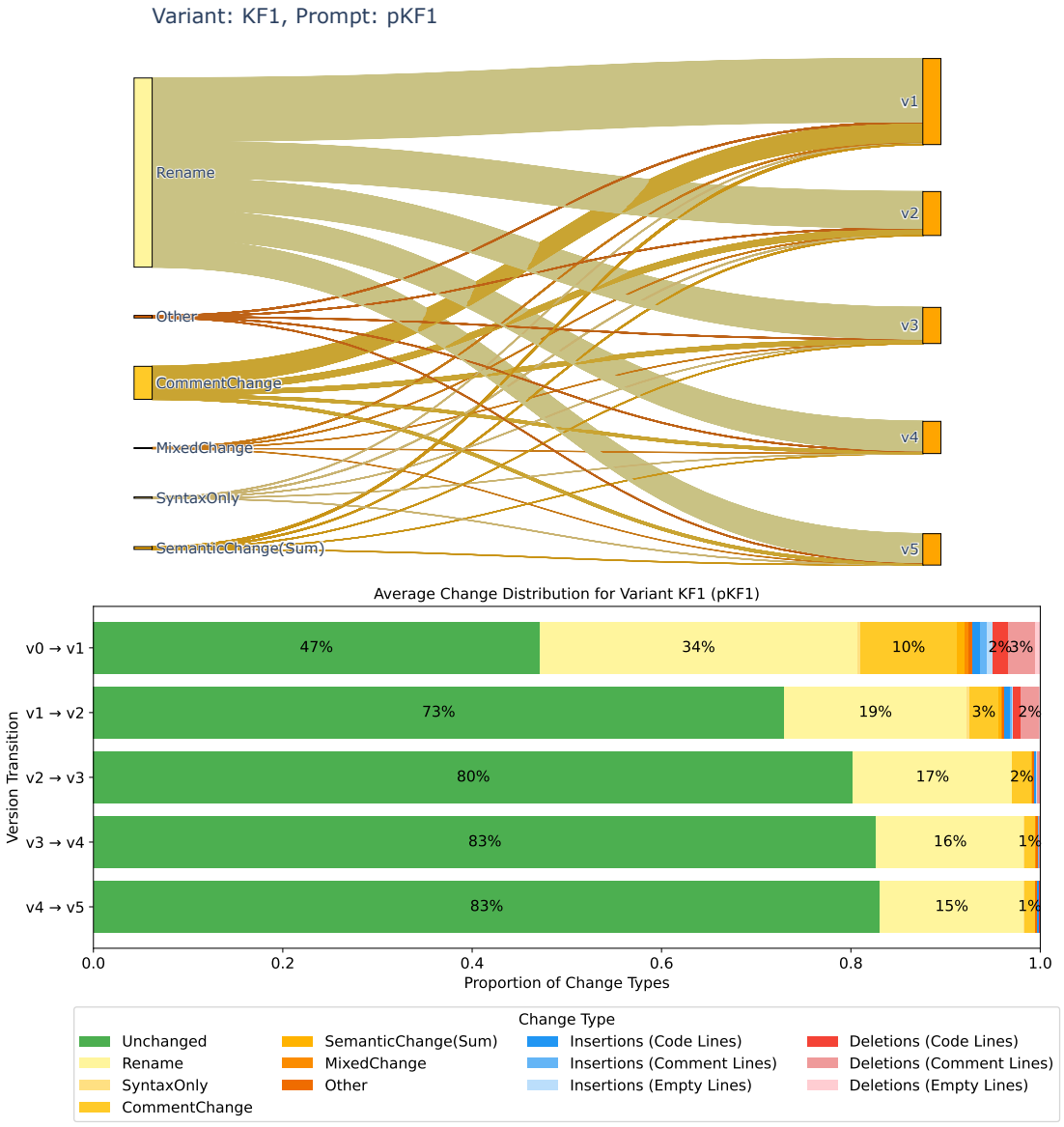

Figure 5.17: Distribution of Change Types for KF1 (pKF1)

proportion of renames, likely reflecting the necessity of re-establishing meaningful identifiers in the obfuscated code. In this initial step, comment modifications account for around 11% of all changes, likely reflecting identifier replacements within comments. We also observe approximately 6% insertions and 4% deletions, which may partially result from alignment and the inherent limitations of the parser's matching process rather than genuine removals of comments.

KF2 - pKF1/pKF2: Figures 5.19 and 5.20 present the detailed distribution of change types across successive refinement steps for variant KF1 (the original code with obfuscated identifiers) under prompt pKF1 and pKF2, respectively. Here also, we find that the two

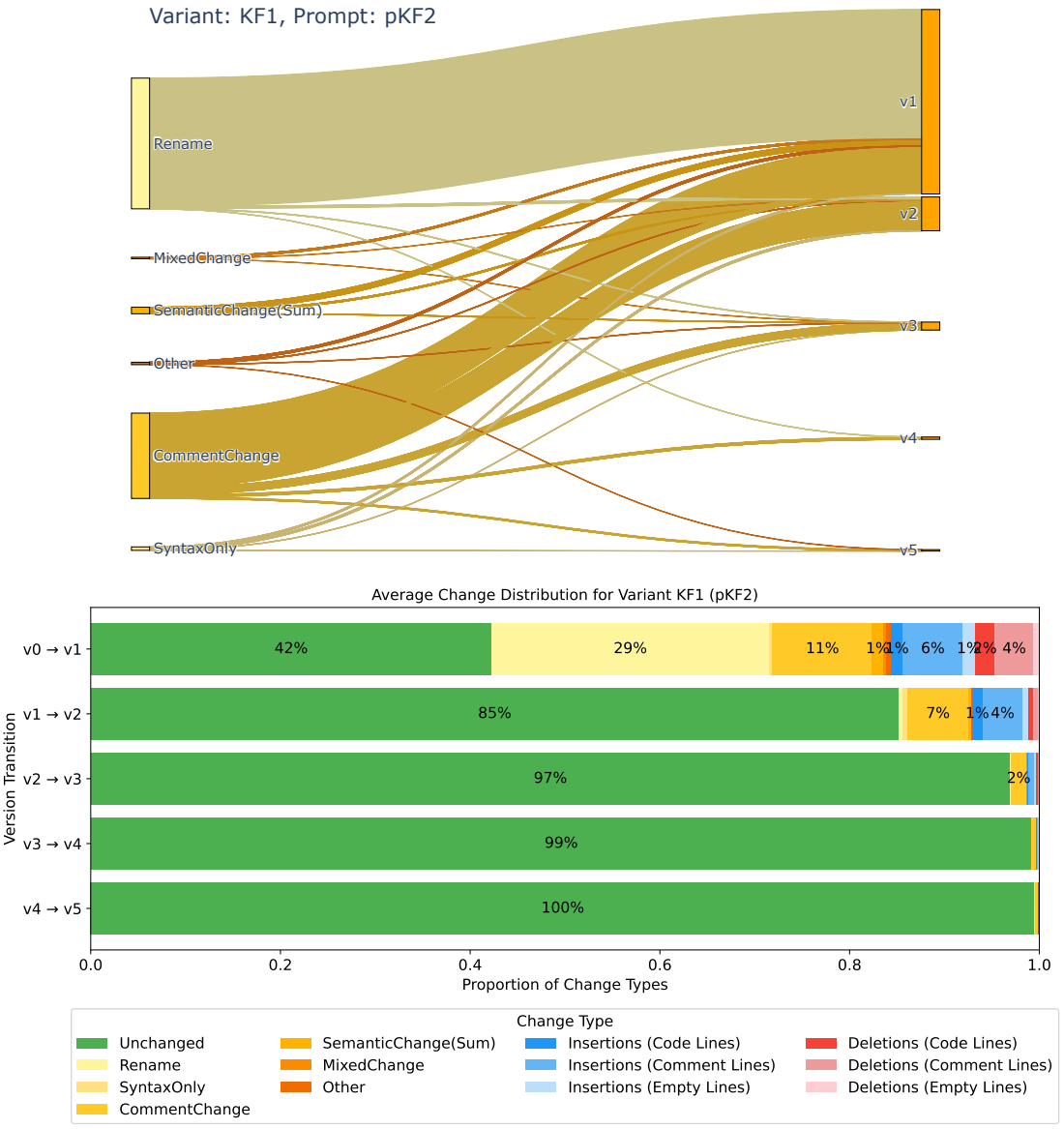

Figure 5.18: Distribution of Change Types for KF1 (pKF2)

prompting strategies drive markedly different change dynamics. Under pKF1, rename modifications dominate across all iterations, starting with a large proportion of renames in the initial transition (v0 *rightarrow* v1) and persisting through later steps. Although the proportion of unchanged lines increases with each iteration, renaming remains the most prominent activity throughout the refinement process.

In contrast, pKF2 produces a pattern centered on comment modifications: the first iteration shows a substantial share of comment changes, accompanied by insertions and deletions that likely reflect alignment issues. After this initial step, activity rapidly diminishes, with subsequent transitions approaching near-stability and almost no further modifications by the final versions.

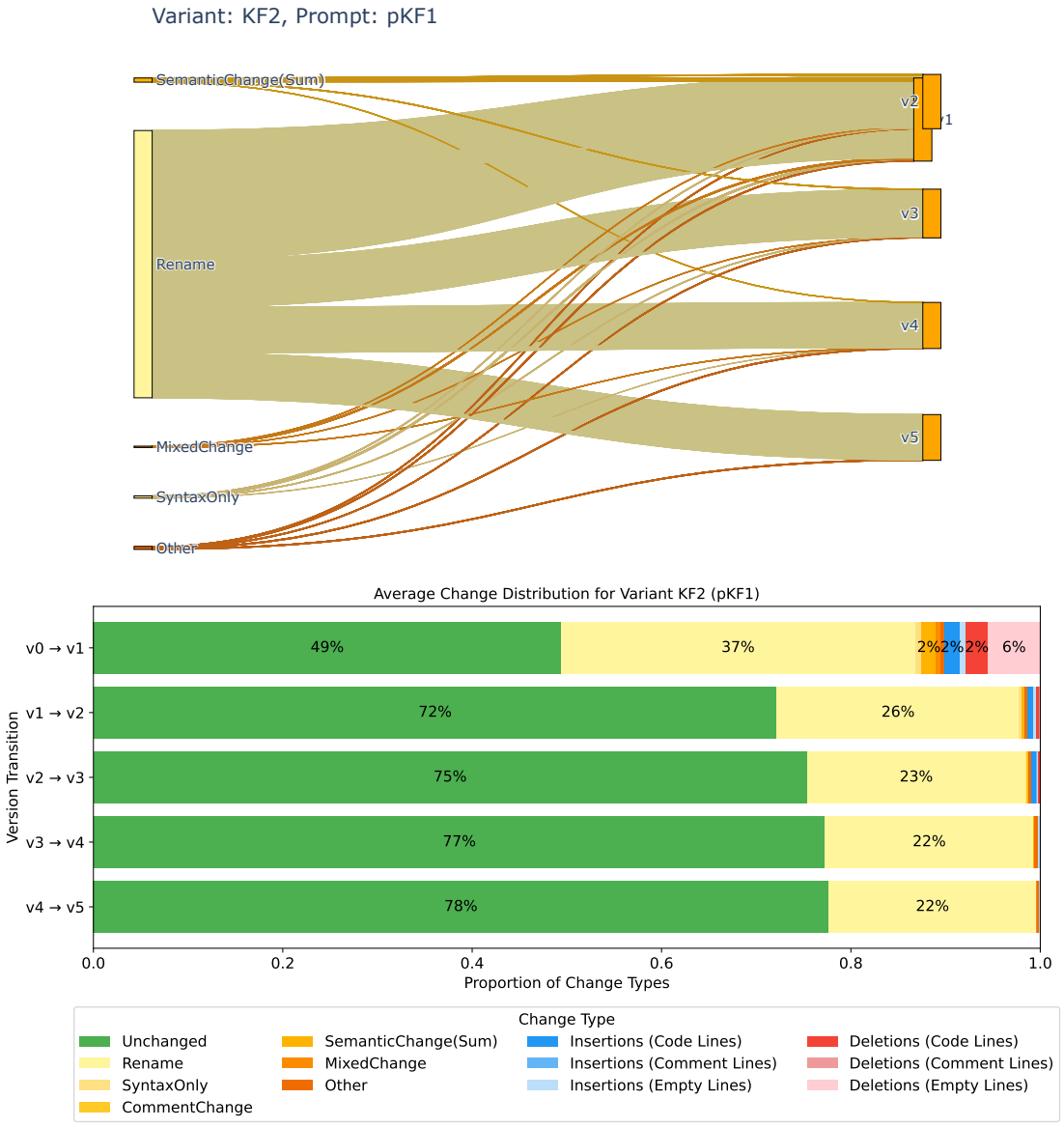

Figure 5.19: Distribution of Change Types for KF2 (pKF1)

To avoid confusion, we note that the large proportion of "CommentChange" observed in v1 arises from the way modifications are classified. Whenever a line of code receives an inline comment, this modification is labeled as a "CommentChange." This does not necessarily imply that a comment was already present and subsequently altered; rather, it indicates that some aspect of the comment component of the line - its presence or its content - has changed.

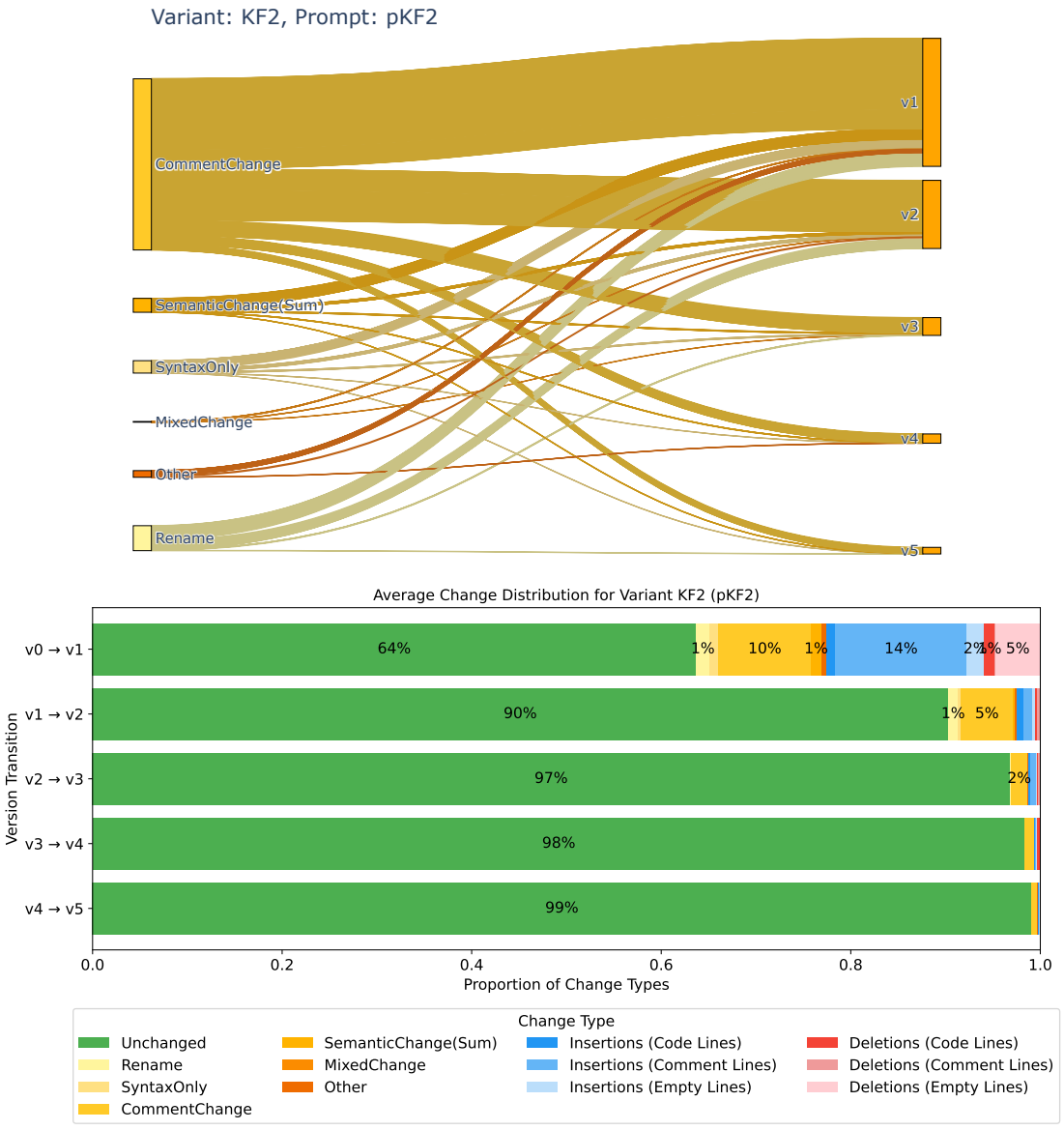

Figure 5.20: Distribution of Change Types for KF2 (pKF2)

5.3.3 Pairwise Similarity Analysis Across Refinements (pKF1 / pKF2)

This section presents the results of the pairwise similarity analysis across successive refinements under the two prompt strategies, pKF1 (identifier renaming) and pKF2 (comment improvement). The goal of this analysis is to examine how similarity scores evolve across versions, and to identify whether different refinement strategies lead to distinct convergence patterns or to recurring modification behaviors. To illustrate these dynamics, heatmaps of the average similarity scores are provided again, complemented by line plots that compare the convergence tendencies of the variant pairs (Figure 5.21).

In the case of pKF1, we can clearly observe seemingly back-and-forth modifications across all variants, which exhibit a very similar pattern and highly comparable similarity scores. Starting from version v1, alternating similarity scores become apparent: pairs of versions separated by two steps consistently share the same score, whereas the direct successor in between always shows a lower score. For example, the similarity between v2 and v3 is 0.93, yet v2 is more similar to v4 with a score of 0.94. This alternation continues, as v2 to v5 returns to 0.93. Notably, the comparison of v3 to v5 yields the highest similarity score of 0.96, while v4 to v5 only reaches 0.94.

An interesting observation in pKF2 is that relatively little change occurs, and the similarity scores resemble those obtained from the base prompt. Already from the transition between v2 and v3, only very few modifications remain. Under pKF2, both KF0 and KF1 even reach absolute convergence, with a similarity score of 1.0. This finding is particularly noteworthy, as pKF1 exhibited back-and-forth modifications involving renaming operations, whereas in the case of requesting improved comments, the model appears to eventually reach an optimum at which no further changes are introduced.

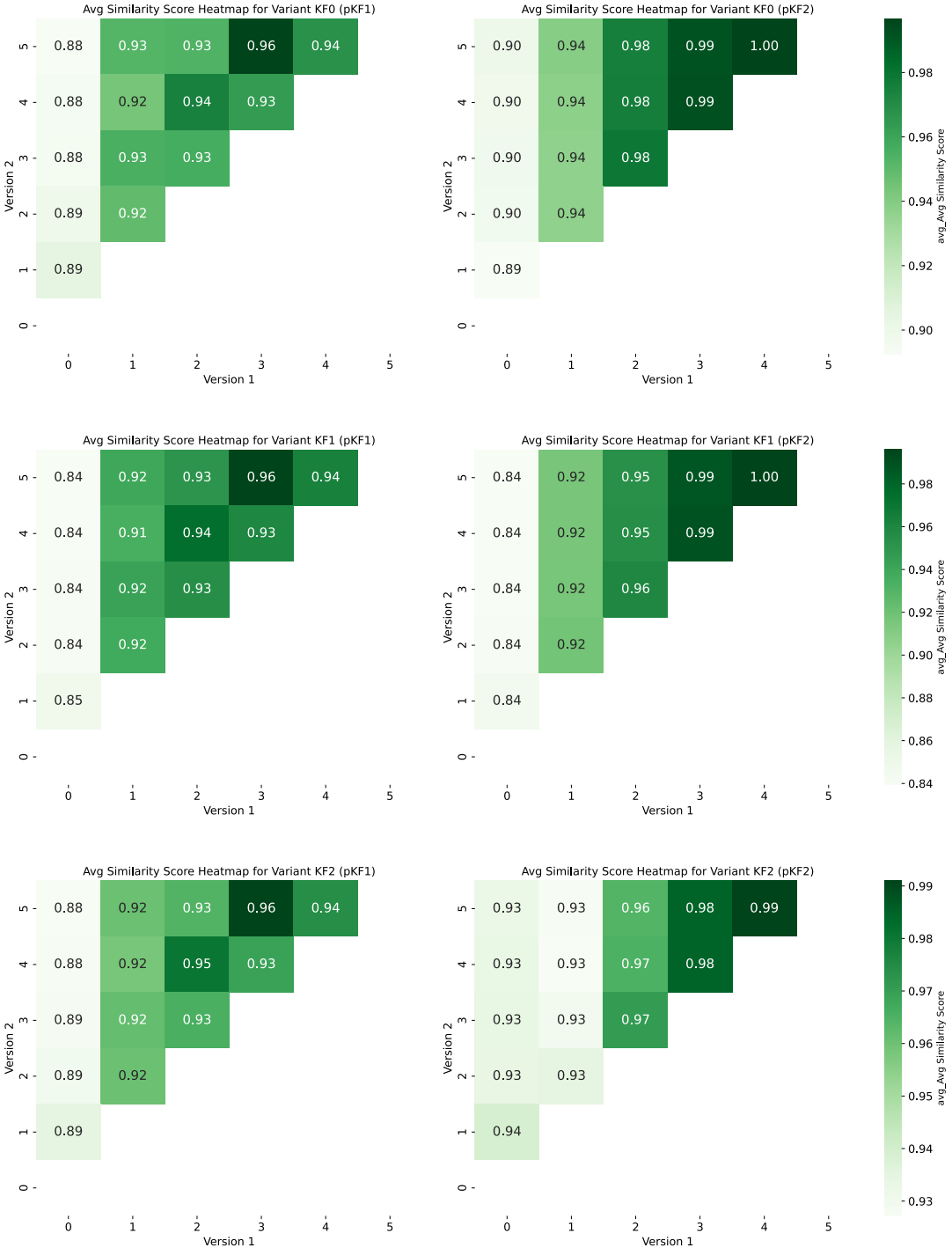

Figure 5.21: Average similarity score heatmaps across all code variants under pKF1 (left) and pKF2 (right)

Similarity Score Evolution Across Variants (pK1 / pKF2)

To directly compare the convergence tendencies across prompt strategies, we analyze the evolution of similarity scores across variant pairs. As we can see in Figure 5.22, the two pairs involving the renaming variant (KF1) consistently maintain either increased or stable similarity scores (approximately 0.915 for pKF1 and 0.875 for pKF2 from v1 to v2). In contrast, the pair combining the original code (KF0) with the original code without comments (KF2) exhibits a more pronounced initial decrease, attributable to the introduction of inline comments (as seen in Figure 5.20, an average of 10% of the code is augmented with inline comments.) For both prompt strategies, the similarity scores across the three variant pairs converge rapidly, with the most substantial change occurring during the transition from v0 to v1. Overall, pKF1 achieves a higher average similarity score across the variant pairs compared to pKF2.

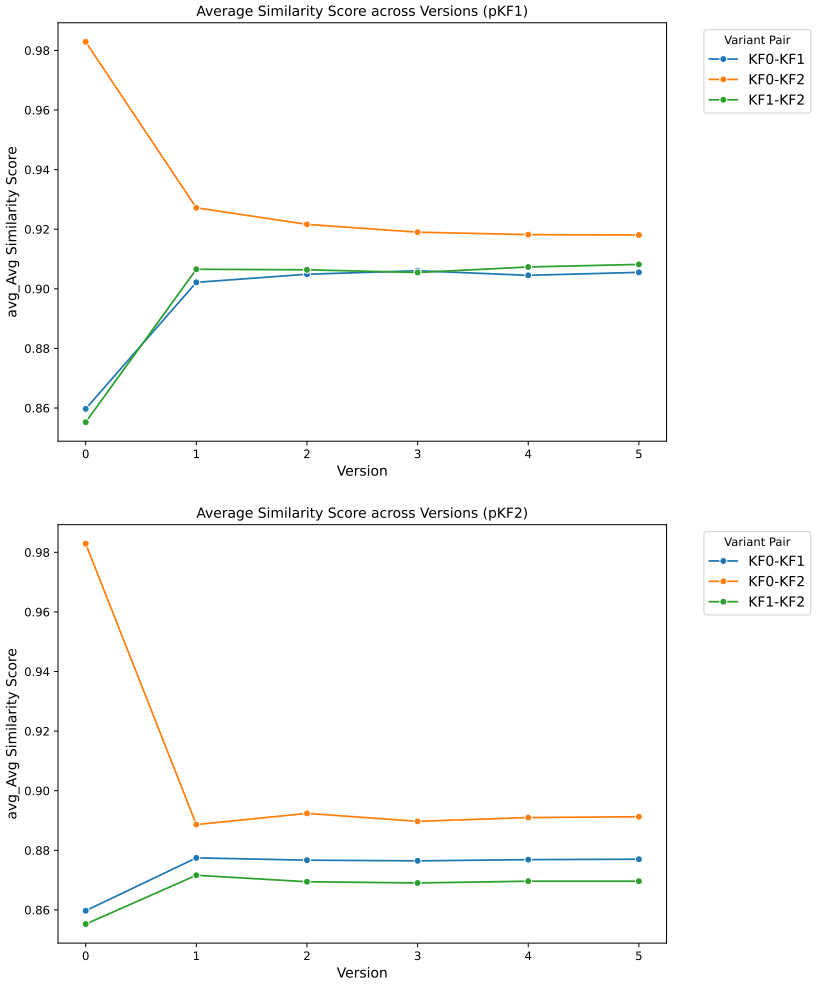

Figure 5.22: Convergence of Average Similarity Score across all code variants under pKF1 (left) and pKF2 (right)

5.3.4 Statistical Analysis

Descriptive Statistics

Table 5.1 reports the mean percentages and standard deviations of *Rename* and *CommentChange* operations across all prompt variants and code versions. These descriptive statistics provide an intuitive understanding of the magnitude and variability of changes induced by different prompts. For instance, the Rename changes under pKF1 consistently show higher means compared to pKF0 and pKF2, such as in KF2 (26.0% ± 15.0%), suggesting a strong prompt-induced tendency toward Rename modifications. Similarly, CommentChange activity under pKF2 peaks in several configurations, indicating differential prompt sensitivity across transformation types. The high standard deviations also highlight the heterogeneity in model responses, further motivating the use of non-parametric statistical testing.

Prompt	Rename	CommentChange
	KF0	
pKF0	2.3% ± 5.3%	0.6% ± 2.0%
pKF1	18.7% ± 11.9%	2.2% ± 3.4%
pKF2	0.3% ± 2.1%	3.2% ± 6.0%
	KF1	
pKF0	6.5% ± 11.9%	1.9% ± 4.4%
pKF1	20.1% ± 12.4%	3.5% ± 5.2%
pKF2	6.0% ± 12.3%	3.9% ± 5.8%
	KF2	
pKF0	3.3% ± 6.5%	0.0% ± 0.4%
pKF1	26.0% ± 15.0%	0.0% ± 0.0%
pKF2	0.5% ± 3.2%	3.7% ± 6.7%

Table 5.1: Descriptive Statistics (Mean ± SD) for Rename and CommentChange

Inferential Statistics

Building on the descriptive patterns observed above, we applied statistical tests to determine whether the observed differences between prompts are statistically significant. To evaluate the influence of different prompt variants on code transformations, we applied a combination of non-parametric and mixed-effects statistical methods. Below, we briefly explain the rationale behind each method and its application to our dataset.

While exploratory analyses were also conducted for all other modification types with false discovery rate correction, here we concentrate on the directed analyses of Rename

and Comment Changes. We report the full set of exploratory analyses, including other change types, in the appendix in Table A.2.

Kruskal-Wallis Test The Kruskal-Wallis H test is a non-parametric alternative to one-way ANOVA. It tests whether samples originate from the same distribution, focusing on differences in median ranks between multiple independent groups. This test is particularly suited for our data, as the distributions of code change metrics are proportional (i.e. normalized between 0 and 1) and therefore not guaranteed to follow a normal distribution. We used the Kruskal-Wallis test to determine whether different prompts had a statistically significant effect on code changes within each code variant (KF0, KF1, KF2).

The tests revealed significant effects of prompt variants on both *Rename* and *CommentChange* modifications across all code variants:

- **Rename**: Significant effects in KF0 ($H = 2199.42$, $p < 0.001$), KF1 ($H = 1123.63$, $p < 0.001$), and KF2 ($H = 2208.56$, $p < 0.001$).
- **CommentChange**: Significant effects in KF0 ($H = 294.96$, $p < 0.001$), KF1 ($H = 182.26$, $p < 0.001$), and KF2 ($H = 936.77$, $p < 0.001$).

Post-hoc Pairwise Comparisons While the Kruskal-Wallis test indicates whether a difference exists among groups, it does not specify which groups differ from each other. To resolve this, we conducted post-hoc pairwise comparisons using the Mann-Whitney U test (also known as the Wilcoxon rank-sum test), which compares the distributions of two independent groups. This allowed us to identify which specific prompt pairs (e.g. pKF0 vs. pKF1) contributed to the overall significant differences.

The post-hoc tests confirmed that these effects of prompt variants on the change types were driven by significant differences between individual prompt conditions. For instance:

- **Rename** (KF0): pKF0 vs. pKF1 ($U = 99398.5$, $p \approx 4.87 \times 10^{-268}$); pKF0 vs. pKF2 ($U = 763607.0$, $p \approx 7.88 \times 10^{-53}$).
- **CommentChange** (KF1): pKF0 vs. pKF1 ($U = 448546.0$, $p \approx 1.40 \times 10^{-35}$); pKF0 vs. pKF2 ($U = 460593.5$, $p \approx 1.56 \times 10^{-31}$).

Mixed Linear Models (MixedLM) To assess the global effect of prompt stratgies across all code variants we employed mixed-effects linear models. These models allow for both fixed effects (prompt) and random effects (variant), enabling us to control for repeated

measures and nested variability. The test statistic from the model comparison (via likelihood ratio tests) follows a chi-square distribution. The mixed-effects models further confirmed the prompt effects across all code variants:

- **Rename (ALL)**: $\chi^2 = 8575.25$, $p < 0.001$
- **CommentChange (ALL)**: $\chi^2 = 16769.06$, $p < 0.001$

These findings provide robust statistical evidence that the choice of prompt has a significant and consistent impact on how models perform both Rename and CommentChange operations, across all code variants examined.

5.3.5 Summary of Findings for RQ3

The analysis of RQ3 investigated whether explicitly emphasizing specific refinement factors in prompts affects the effectiveness and stability of iterative code refinements. Results demonstrate that prompt design substantially shapes refinement trajectories. When the prompt targeted identifier renaming (pKF1), refinements were dominated by recurring rename operations, producing back-and-forth modifications across iterations without reaching stable convergence. In contrast, prompts emphasizing comments (pKF2) induced a sharp increase in comment density during the first refinement step, followed by rapid stabilization and minimal subsequent changes, reflecting clear convergence behavior. Pairwise similarity analyses confirmed this contrast: pKF1 exhibited oscillating similarity scores indicative of instability, whereas pKF2 often achieved high, sometimes absolute, convergence.

The statistical tests provide sufficient evidence to reject the null hypothesis (H0), which assumed no significant difference in refinement effectiveness between targeted and unguided prompts. Both the Kruskal–Wallis tests and the mixed-effects models identified significant prompt-related effects for the two central modification types (Rename and Comment Changes). In line with the alternative hypothesis (H1), targeted prompts systematically produced distinct refinement outcomes: pKF1 consistently elicited a higher frequency of rename modifications, while pKF2 exerted stronger effects on comment-related changes. These findings indicate that explicitly emphasizing a refinement factor in the prompt significantly shapes the trajectory of code modifications, thereby supporting H1. Building on our findings, the discussion turns to a broader interpretation of what the observed refinement dynamics reveal about ChatGPT's behavior as a code-refactoring agent.

5.4 Complete Summary of Key Findings

Overall, the results demonstrate that ChatGPT's iterative refinements follow a consistent trajectory across different experimental conditions. For well-structured code (RQ1), the model largely preserved the original organization but nonetheless introduced unnecessary adjustments in early iterations before stabilizing toward a near-converged state. When applied to systematically varied code snippets (RQ2), the refinements converged across all variants, indicating that the model normalizes divergent inputs into structurally similar representations over time. Finally, explicit prompt design (RQ3) was shown to exert a strong influence on refinement trajectories: while prompts emphasizing identifier renaming induced recurring back-and-forth modifications, prompts highlighting comments facilitated rapid stabilization and, in some cases, absolute convergence. Taken together, these findings provide robust evidence that iterative refinements by ChatGPT are characterized by early restructuring followed by convergence, with prompt design serving as a decisive factor in determining whether this process stabilizes or remains oscillatory.

Chapter 6

Discussion

The open question identified in the gap analysis was how LLMs iteratively self-improve code, whether this happens in a stable and reliable manner, and which effect different starting conditions and prompt strategies have. Our study partially fills this gap by the developed framework to measure these code changes substantially.

This chapter discusses the findings of the study in relation to the research questions. It interprets the observed refinement patterns, compares them to existing work, and highlights overarching themes that emerge across the analyses. In doing so, it outlines how the results contribute to a deeper understanding of iterative code modifications by LLMs and sets the stage for identifying future research directions.

6.1 Interpretation for RQ1

The analysis of RQ1 reveals that ChatGPT tends to modify code even when it already adheres to established best practices. However, these modifications stabilize after a few refinement iterations. The most prominent structural refactoring involves encapsulating functionality into smaller, more modular functions, which on the one hand partially supports earlier findings that (iteratively) generated code often suffers from unnecessary or increasing complexity ([28], [36]). On the other hand, the pattern we found suggests that the model initially identifies a substantial number of aspects to “improve", particularly when dealing with code that has not been previously refined by an LLM, but then structural refactorings stabilize quickly after only a few iterations.

One possible interpretation is that what is conventionally perceived as “best practice” code may not fully align with ChatGPT’s internalized representation of best practices. Another interpretation is that the model employs an alternative, model-specific notion

of code quality that extends beyond established human conventions. Importantly, the diminishing number of modifications in subsequent iterations indicates that the model converges toward stability unless explicitly prompted to continue applying targeted changes (e.g. systematic renamings). This raises the question of whether defining a convergence criterion is necessary at all, or whether such a criterion could effectively minimize or even eliminate the large-scale alterations observed in the initial iterations.

Another striking observation is that, in the absence of explicit prompting, ChatGPT consistently removes comments from the code. While this behavior might streamline the code's appearance, it risks impairing readability for human developers and erases potentially valuable contextual information. Such comments may serve to document design rationales, highlight critical sections of the code, or record bug fixes and other implementation-specific considerations that would otherwise be lost. For example, comments that extend beyond mere functional descriptions (e.g., those addressing error resolution) exhibit a particularly strong correlation with successful bug fixing ([37]).

On the other hand, poorly written comments are more harmful than having no comments at all, as they provide software developers and maintainers with distorted and misleading information ([38]). Consequently, removing comments that do not align with the code context may in fact improve readability rather than harm it. However, substantiating such an effect would require either a dedicated qualitative study or the application of advanced natural language processing techniques capable of assessing the contextual appropriateness of code comments.

6.2 Interpretation for RQ2

The analysis of RQ2 reveals a striking pattern: despite starting from variants with degraded readability features such as inconsistent naming or missing comments most versions ultimately converge toward highly similar code after several refinement iterations. Variants initialized with obfuscated identifiers indeed underwent a larger number of renaming operations, particularly in the early stages of refinement. This aligns with prior findings demonstrating that ChatGPT is capable of deriving clearer and more meaningful variable names even when the code is difficult to read or intentionally obfuscated ([31]).

This behavior points to a broader implication: ChatGPT tends to systematically reduce structural and stylistic differences between code variants, producing what could be described as a homogenization effect. The high frequency of renamings in poorly chosen identifier sets suggests that the model possesses a relatively fine-grained understanding of what constitutes "good" variable naming in the given context. While the present

analysis cannot fully determine whether the chosen names optimally enhance readability or semantic clarity, our spot checks indicated that the replacements were context-appropriate and consistently expressed the intended function of the variables. A qualitative follow-up study would be necessary to more rigorously assess whether the new identifiers improve readability for human developers.

Interestingly, the overall convergence stands in contrast to initial expectations. One might have hypothesized that, given the vast space of possible code formulations, variants would diverge into increasingly distinct versions over multiple iterations. Instead, the opposite occurred: across different starting conditions, the end versions gravitated toward a common form. This outcome may in fact be considered a best-case scenario, as it demonstrates that ChatGPT does not simply generate arbitrary transformations, but rather follows a discernible direction in refining code toward internally consistent and arguably more standardized solutions.

6.3 Interpretation for RQ3

Our findings highlight that explicitly emphasizing particular refinement factors substantially shaped the trajectory of modifications. Naming-focused prompts (pKF1) consistently induced persistent back-and-forth renaming, whereas comment-focused prompts (pKF2) produced rapid stabilization and, in several cases, near-perfect convergence. This pattern resonates with prior work on refactoring interactions, which similarly observed that developers' more specific prompts yield more targeted and effective model responses ([17, 30]).

A closer comparison underscores two complementary insights. First, when guided by naming-focused prompts, the model engaged less in structural reorganization or code expansion. Auxiliary methods were seldom introduced, and control flow restructuring was minimized. This constrained focus suggests that explicit task framing channels ChatGPT away from its otherwise more creative or exploratory refinements, a tendency that is consistent with recent findings showing that prompts and role-play settings significantly influence the model's creativity ([39]). However, the same condition also produced a recurring pattern of back-and-forth renaming. The absence of a strict convergence criterion under pKF1 appears to allow continuous oscillation around variable names rather than stabilization once an arguably optimal naming scheme is reached.

By contrast, comment-focused prompts (pKF2) exhibited a markedly different dynamic. Here, convergence occurred even in the absence of an explicit convergence criterion.

ChatGPT appeared to "recognize" that additional comment insertions would not further improve readability, thereby stabilizing after only minor adjustments. While it remains unclear whether this reflects genuine semantic reasoning or merely a reduction in modification tendencies under comment-emphasis, the effect is notable: targeted prompting does not inherently prevent convergence, but its outcome depends on the nature of the targeted factor. Interestingly, despite pKF2's explicit focus on comments, a substantial number of renaming operations still occurred particularly in the KF1 variant with obfuscated identifiers, indicating that the model might recognize the quality of identifier names and prioritizes it even under a prompt targeted at another key factor.

6.4 Future Work

Convergence vs. Oscillation. Our results demonstrate that iterative refinements exhibit a clear pattern of early restructuring followed by stabilization. This supports the findings by Liu et al. [33] who found that the majority of code improvements occur in early iterations. In line with the open question of whether LLMs converge toward higher-quality solutions, we find that ChatGPT consistently normalizes structurally divergent inputs, even when identifiers are obfuscated or comments removed. However, convergence remains partial: similarity scores often plateau below 1.0, and micro-modifications persist across iterations, indicating that the model occasionally engages in over-refinement and might benefit from a principled stopping criterion.

An additional dimension that may influence convergence is the availability of context. Given that API requests are stateless, the back-and-forth modifications cannot be explained by the model retaining information about previous snippet versions. Rather, the observed oscillation appears to arise from the model's internal distribution over plausible identifier names, which leads it to revert to familiar candidates in the absence of persistent contextual anchoring.

Future research could therefore examine whether maintaining context across refinement steps would alter convergence dynamics, potentially reducing oscillatory behavior. Nevertheless, our findings already indicate that even in a stateless setup, ChatGPT sometimes oscillates within a narrow set of highly similar identifier names when repeatedly prompted to improve naming. This suggests that without contextual anchoring, the model may converge only locally and remain vulnerable to back-and-forth modifications in semantically equivalent but stylistically similar solutions.

Decomposition of Refinement Types. By analyzing insertions, deletions, and different change types of modifications over time, this study provides the layered characterization

of iterative refinement that was previously missing. The results reveal that early iterations are dominated by structural transformations (comment deletions, renaming, or semantic changes), while later iterations narrow down to minor surface-level edits.

Naturally, this raises further questions regarding the nature of these structural transformations. Which code locations are particularly frequent or rare targets of change? Which AST node types are most commonly affected? Which change types are especially associated with specific AST node categories? Moreover, which change types tend to co-occur, and which are largely mutually exclusive? Another important aspect is whether code size plays a role. For instance, are certain change types applied more frequently in larger snippets compared to smaller ones?

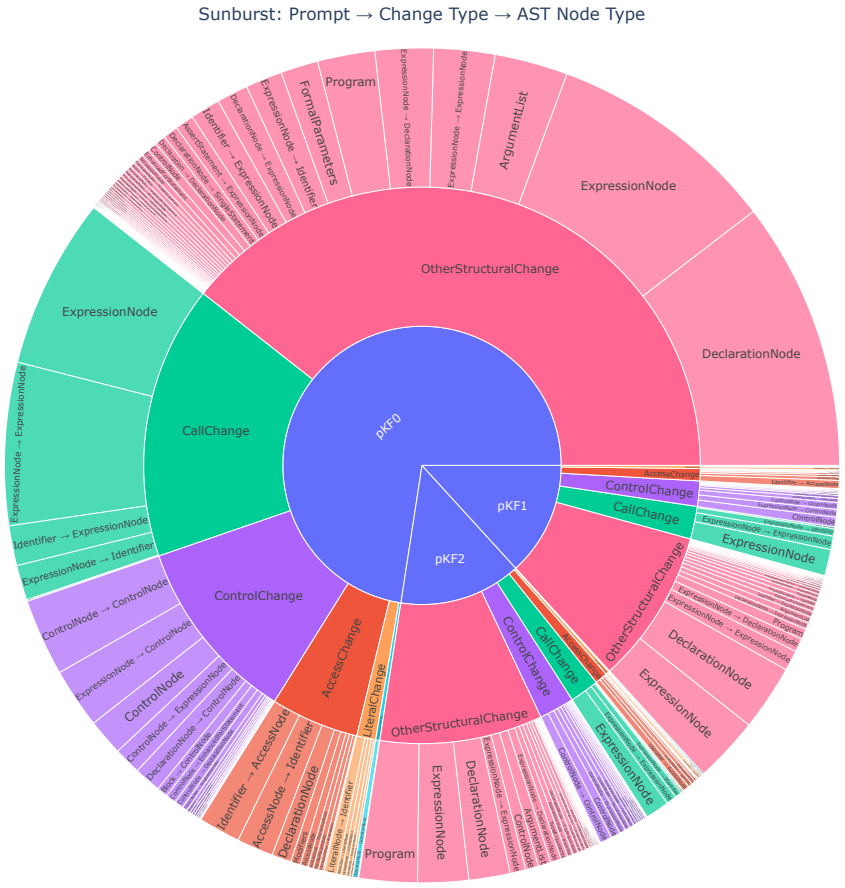

Figure 6.1: Sunburst diagram showing the distribution of AST node types affected by semantic changes, grouped by prompt (center), change type (middle layer), and node category (outer layer)

These questions unfortunately go beyond the scope of this thesis. However, the framework has been designed in such a way that the necessary data (i.e. information on which AST node types are affected by a given modification at the line level) are already partially

collected during the DiffParsing process. An exemplatory overview of affected AST node types per semantic change type is provided in Figure 6.1. The figure presents a hierarchical distribution of changes across prompts, semantic change types, and affected AST node categories. It shows that a substantial portion of current detections are still grouped under *OtherStructuralChange*, reflecting the early stage of implementation for semantic change classification. Nonetheless, clear patterns already emerge for implemented categories such as *CallChange* or *ControlChange*, which frequently affect specific node types like `ExpressionNode` or `ControlNode`. This structure lays the groundwork for more fine-grained analyses, including the co-occurrence of change types and the identification of structural hotspots within code. Understanding which AST node types are affected by specific change types can shed light on common refinement strategies, developer behavior, and code evolution over time. This knowledge could inform the development of smarter tooling for code review, automated refactoring, and the evaluation of code-generating models. Moreover, it provides a foundation for more nuanced comparisons between different kinds of prompts or user strategies.

6.5 Threats to Validity

Although the study was carefully designed, certain limitations remain that may affect the interpretation and generalizability of the results. Following established guidelines in empirical software engineering, we discuss threats to validity along four dimensions: internal validity (factors that may have influenced the correctness of our measurements), external validity (the extent to which our findings can be generalized), conclusion validity (the robustness of the inferences drawn from the data), and construct validity (the adequacy of our operationalization of the studied concepts).

Internal Validity. The accuracy of our results depends directly on the performance of the matching process implemented in our parser. Correctly classifying actual code modifications is a non-trivial task, and while the parser was iteratively refined, the development dataset naturally could not cover all possible cases of code line pairs that may occur in a diff. It is therefore likely that some lines were mismatched, left unmatched, or otherwise incorrectly processed. Such errors may have introduced noise into the analysis, potentially affecting the precision of our measurements.

Beyond the matching process itself, further limitations arise from the design of our change-type classification. One notable example concerns the handling of rename operations. Rename operations were not normalized in our analysis: if a variable occurred multiple times in the code, each instance was counted as a separate renaming operation. While

the parser could in principle be extended to consolidate such cases into a single renaming event, this is not a trivial enhancement. At the same time, reporting absolute counts is informative, as renaming a variable with many occurrences can have greater practical impact than changing a variable used only once. Thus, although this design choice should be kept in mind, it does not fundamentally distort the overall results. A more critical question, however, would be whether the model consistently renames all instances of a given variable. Inconsistent renaming would not only produce misleading counts but could also render the code dysfunctional or subtly alter its behavior - an aspect that remains untested in our study. This limitation should therefore be kept in mind when interpreting the relative weight of renaming operations in the overall results.

Another limitation arises from the fact that we did not evaluate the executability or functional correctness of the generated code, as such an assessment would have exceeded the scope of this study. Nevertheless, this omission implies a potential threat: the likelihood of introducing hidden errors may increase with the number of modifications applied to a snippet. Without systematic testing, we cannot rule out the possibility that some refinements reduced correctness while still appearing to improve readability or structure.

External Validity. The generalizability of our findings is limited by the fact that we relied on a single model throughout the study. Moreover, we did not employ the most recent version (GPT-5o), as it became available only after the data collection had been completed. Consequently, our results cannot be assumed to fully extend to other large language models (LLMs). At the same time, the methodological pipeline itself is not bound to any specific model: as long as an API key is available, only minimal adjustments to the request format are required. Thus, while our conclusions remain tied to the model under study, the approach itself can be readily transferred to different LLMs.

Another limitation concerns the dataset itself. Since we relied on code from a publicly available repository, it is highly likely that at least parts of this material were included in the model's training data. While this may have influenced the specific refinements proposed by ChatGPT, it does not diminish the value of our framework, which systematically captures and analyzes modifications regardless of the source. In principle, the same pipeline could also be applied to proprietary code that is unlikely to have been part of the training data, thereby enabling direct comparisons. Nevertheless, our current findings should not be generalized to all types of source code. For instance, code containing poor practices might yield entirely different refinement patterns, a question that lies outside the scope of our present study but could be explored in future work.

Conclusion Validity. By averaging results across a large number of snippets and variants, important details may be obscured. For example, back-and-forth modifications can occur in individual refinement sequences, yet their impact may be diluted in the aggregate analysis. As a consequence, such phenomena might not strongly affect overall similarity scores, even though they represent meaningful dynamics at the micro level. Capturing these nuances would require complementary qualitative analyses, which were beyond the scope of this study.

A further aspect concerns the design of our similarity measure. Our average similarity score does not account for the change categories of insertions and deletions, and therefore does not theoretically capture the overall similarity between entire snippets. Instead, it reflects only the similarity of the code segments modified between two successive versions. This design choice could, in principle, distort the measurement of convergence. However, when considering the distribution of change types across iterations, it becomes evident that the proportion of insertions and deletions decreases sharply with higher version numbers and is negligible in later stages. As a result, the average similarity score remains a representative proxy for overall similarity in the context of this study.

Insertions and deletions were also not explicitly represented as distinct change types in our classification scheme. Similar to the design of the average similarity score, this choice could in principle limit the completeness of the analysis. However, the distribution of change types across iterations shows that insertions and deletions occur only rarely in later versions and therefore do not substantially affect the main findings. For the sake of completeness, though, future work could extend the classification to also capture code growth or reduction explicitly—for example, by reflecting the expansion or contraction of comment sections in the Sankey plots rather than focusing solely on modifications to existing comments.

A further concern relates to the inherent non-determinism of LLM outputs. Ideally, the experiment should be repeated multiple times to examine whether the overall outcomes remain stable across different runs. Such replication would provide stronger evidence regarding the reliability of the observed patterns. In our study, however, we followed current best practices by fixing the generation parameter to temperature=0, which is commonly used to minimize randomness. While this increases reproducibility to some extent, it does not fully eliminate the possibility that repeated runs could yield slightly different results.

Construct Validity. A major limitation of our study lies in the fact that we did not directly measure whether the resulting code was indeed more readable. While we frequently observed indications of improved readability during development, these impressions

remain anecdotal rather than systematically validated. An alternative approach could have involved the use of static analysis tools to assess code quality issues, as demonstrated in related work ([33]). Such tools might have been employed to first evaluate the "best practice" snippets and identify the improvements they suggest, thereby enabling a comparison with ChatGPT's refinements. Nevertheless, even without tool-based measurement, we qualitatively observed clear improvements during the pilot experiment: ChatGPT consistently produced more uniform and well-formatted code outputs and, in cases of variable renaming, was able to correctly infer and assign meaningful names based on context. Prior research has likewise demonstrated the model's strong capability to summarize code and capture contextual information ([25]). In the main experiment, given the size of the dataset, we relied on sample-based qualitative checks rather than systematic validation. This decision also reflected our primary focus on building a framework to measure code modifications across arbitrary iterations of LLM-optimized code. Ultimately, however, determining whether one snippet is more readable than another remains inherently subjective, underscoring the need for a larger-scale qualitative study to rigorously assess improvements or regressions in code readability.

Chapter 7

Conclusion

This thesis investigated ChatGPT's capacity to iteratively refine source code and its consistency in improving code readability. Building on a pilot experiment and a large-scale main study, we developed a systematic framework to capture, classify, and quantify changes across refinement rounds, with a particular focus on convergence and the role of prompting strategies. The results show that ChatGPT exhibits both strengths and limitations as a refactoring tool. When applied to code that already follows best practices, the model typically preserved structure but introduced minor, sometimes unnecessary, modifications before stabilizing. Across structurally varied variants, iterative refinements often converged toward highly similar final versions, suggesting a normalization effect in which ChatGPT aligns diverse inputs with implicit coding conventions. Prompt design proved critical: emphasizing variable naming frequently led to oscillatory changes, whereas prompts stressing comments supported faster stabilization and, in some cases, full convergence. These findings make three key contributions: (1) a methodological pipeline for analyzing iterative code modifications at scale, (2) empirical evidence that LLM-based refinements tend toward convergence while remaining vulnerable to back-and-forth modifications, and (3) insights into how prompt strategies influence refinement trajectories. Collectively, the work provides an empirical foundation for assessing the reliability of LLM-assisted code improvement.

Future research should extend this work along several dimensions: assessing functional correctness in addition to structural similarity, exploring larger and more diverse codebases, and systematically varying prompt strategies and contextual anchoring. By addressing these aspects, subsequent studies can further clarify the conditions under which LLM-driven refinements contribute meaningfully to code quality. In doing so, they will bring us closer to harnessing the full potential of large language models as partners in software development.

List of Figures

List of Tables

Appendix A

Appendix

Table A.1: One representative example per classification of code modifications

AccessChange
Source: `for (int row = 0; row < numRows; row++) {`
Target: `for (int row = 0; row < table.length; row++) {`
CallChange
Source: `return columnarTransposition(fractionatedText.toString(), key);`
Target: `return fractionatedText.toString();`
CommentChange
Source: `* Class for converting from "any" base to "any" other base`
Target: `* Class for converting numbers between any two bases`
ControlChange
Source: `if (i + word.length() <= target.length() && target.substring(i, i + word.length()).equals(word)) {`
Target: `return index + word.length() <= target.length() && target.substring(index, index + word.length()).equals(word);}`
LiteralChange
Source: `charCountMap.put(c, charCountMap.getOrDefault(c, 0) + 1);`
Target: `charCountMap.put(c, charCountMap.getOrDefault(c, 0) + increment);`
MixedChange
Source: `private void storeAllPathsUtil(Integer u, Integer d, boolean[] isVisited, List<Integer> localPathList) {`

Target: `private void findAllPathsUtil(int current, int destination, boolean[] visited, List<Integer> path) {}`

OperatorChange

Source: `if ((currentMask & (1 « personIndex)) != 0) {`

Target: `if ((currentMask & (1 « personIndex)) == 0) {}`

OtherStructuralChange

Source: `return sumOfDividers(a, a) == b && sumOfDividers(b, b) == a;`

Target: `return sumOfDivisors(a) == b && sumOfDivisors(b) == a;`

Rename

Source: `for (char c : plaintext.toCharArray()) {`

Target: `for (char c : text.toCharArray()) {}`

```java
public class BinarySearch {

    // Perform binary search on a sorted array

    int binarySearch(int arr[], int x) {

        int l = 0, r = arr.length - 1;

        while (l <= r) {
            int m = l + (r - l) / 2;
            if (arr[m] == x)
                return m; // Element found
            if (arr[m] < x)
                l = m + 1; // Search in right half
            else
                r = m - 1; // Search in left half
        }
        return -1; // Element not found
    }

    public static void main(String args[]) {

        BinarySearch ob = new BinarySearch();

        int arr[] = { -18, 12, 17, 23, 38, 54, 72 };
        int n = 23;

        int result = ob.binarySearch(arr, n);

        if (result == -1)
            System.out.println(n + " not present");
        else
            System.out.println(n + " found at index " + result);
    }
}
```

Binary Search

```java
public class BubbleSort {

    public static void main(String[] args) {

        int[] numbers = {3, 1, 4, 5, 2}; // Array of numbers to sort

        bubbleSort(numbers); // Sorting the array

        for (int number : numbers) {
            System.out.print(number + " "); // Printing sorted numbers
        }
    }

    // Method to perform bubble sort

    public static void bubbleSort(int[] array) {

        int n = array.length;

        for (int i = 0; i < n - 1; i++){
            for (int j = 0; j < n - i - 1; j++){
                if (array[j] > array[j + 1]) {
                    // Swap elements if they are in wrong order
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    }
}
```

Bubble Sort

```java
public class CheckPrime {
    public static void main(String[] args) {

        int num = 29;

        boolean flag = false; // to signal whether number is prime

        for(int i = 2; i <= num / 2; ++i) {

            if(num % i == 0) {
                flag = true;
                break;
            }
        }
        if (!flag)
            System.out.println(num + " is a prime number.");
        else
            System.out.println(num + " is not a prime number.");
    }
}
```

Check Prime

```java
public class FibonacciIterative {
    public static void main(String[] args) {
        int n = 10; // Number of elements in the series
        fibonacci(n);
    }

    public static void fibonacci(int n) {

        int first = 0, second = 1;

        System.out.println("Fibonacci Series up to " + n + ":");

        for (int i = 1; i <= n; ++i) {

            System.out.print(first + " ");

            // Compute the next term

            int next = first + second;

            first = second;
            second = next;
        }
    }
}
```

Fibonacci Sequence

Figure A.1: The four algorithms as starting point for the pilot experiment

Table A.2: Full set of exploratory analyses with Mixed Linear Models to assess the global effect of prompt strategies across all code variants

Change Type	Code Variant	Statistical Test	Test Statistic	p-value	Corrected p-value (FDR)
Rename	KF0	Kruskal-Wallis	2199.4225709643983	0.0	0.0
Rename	KF0	Post-hoc pKF0 vs pKF1	99398.5	4.866452592291743e-268	5.422618602839371e-267
Rename	KF0	Post-hoc pKF0 vs pKF2	763607.0	7.87874923662615e-53	2.048474801522799e-52
Rename	KF0	Post-hoc pKF1 vs pKF2	1183464.5	0.0	0.0
Rename	KF1	Kruskal-Wallis	1123.625739339032	1.0180540601904548e-244	9.926027086856935e-244
Rename	KF1	Post-hoc pKF0 vs pKF1	209423.5	5.022844741038836e-163	3.56165354364572e-162
Rename	KF1	Post-hoc pKF0 vs pKF2	690900.5	3.6207823538323333e-11	6.418659627248228e-11
Rename	KF1	Post-hoc pKF1 vs pKF2	1025229.5	2.594074675754458e-180	2.0233782470884777e-179
Rename	KF2	Kruskal-Wallis	2208.564002095438	0.0	0.0
Rename	KF2	Post-hoc pKF0 vs pKF1	91284.5	1.1413454391988733e-273	1.4837490709585353e-272
Rename	KF2	Post-hoc pKF0 vs pKF2	791221.5	5.576230826912632e-65	1.8910695847790665e-64
Rename	KF2	Post-hoc pKF1 vs pKF2	1184227.0	0.0	0.0
Rename	ALL	MixedLM	8575.250496802524	0.0	0.0
SyntaxOnly	KF0	Kruskal-Wallis	328.26308112519405	5.230914571069791e-72	2.266729647463576e-71
SyntaxOnly	KF0	Post-hoc pKF0 vs pKF1	734941.5	3.552913646304459e-48	8.66022701286712e-48
SyntaxOnly	KF0	Post-hoc pKF0 vs pKF2	723729.5	7.122149188628981e-38	1.5014260451704338e-37
SyntaxOnly	KF0	Post-hoc pKF1 vs pKF2	599489.0	0.011410508937903968	0.013088524958183963
SyntaxOnly	KF1	Kruskal-Wallis	282.4845309040467	4.563136945269622e-62	1.423698726924122e-61
SyntaxOnly	KF1	Post-hoc pKF0 vs pKF1	722950.5	6.210336302457013e-41	1.384017804547563e-40
SyntaxOnly	KF1	Post-hoc pKF0 vs pKF2	715619.0	1.8888942892240464e-34	3.777788578448093e-34

Change Type	Code Variant	Statistical Test	Test Statistic	p-value	Corrected p-value (FDR)
SyntaxOnly	KF1	Post-hoc pKF1 vs pKF2	602853.5	0.0923793573453854	0.102936998184858
SyntaxOnly	KF2	Kruskal-Wallis	255.10215214792208	4.0296887257420504e-56	1.1225561450281426e-55
SyntaxOnly	KF2	Post-hoc pKF0 vs pKF1	732772.5	5.391859552133956e-41	1.2369560149013193e-40
SyntaxOnly	KF2	Post-hoc pKF0 vs pKF2	718985.5	1.730142402117491e-30	3.291490423540592e-30
SyntaxOnly	KF2	Post-hoc pKF1 vs pKF2	595281.0	0.010295090911506209	0.012091985489722134
SyntaxOnly	ALL	MixedLM	25353.530449313388	1.6350198764706986e-61	4.905059629412096e-61
CommentChange	KF0	Kruskal-Wallis	294.9577975475699	8.927463132550365e-65	2.9014255180788686e-64
CommentChange	KF0	Post-hoc pKF0 vs pKF1	402052.0	2.1718248422288394e-65	8.066777985421404e-65
CommentChange	KF0	Post-hoc pKF0 vs pKF2	450201.0	8.229793486924839e-44	1.945223915091326e-43
CommentChange	KF0	Post-hoc pKF1 vs pKF2	632797.5	0.09854581773749083	0.10826160258484908
CommentChange	KF1	Kruskal-Wallis	182.2601166278274	2.646784232245119e-40	5.7346991698644245e-40
CommentChange	KF1	Post-hoc pKF0 vs pKF1	448546.0	1.3964188913040234e-35	2.8663335137293115e-35
CommentChange	KF1	Post-hoc pKF0 vs pKF2	460593.5	1.5551871848984086e-31	3.032615010551897e-31
CommentChange	KF1	Post-hoc pKF1 vs pKF2	613311.5	0.8415347024257779	0.8415347024257779
CommentChange	KF2	Kruskal-Wallis	936.768107559207	3.8316870574515915e-204	3.3207954497913793e-203
CommentChange	KF2	Post-hoc pKF0 vs pKF1	614932.5	0.004617218155398055	0.005808758324533037
CommentChange	KF2	Post-hoc pKF0 vs pKF2	379712.0	1.8787007472691939e-109	1.0467047020499794e-108
CommentChange	KF2	Post-hoc pKF1 vs pKF2	376252.5	2.3519680664113193e-114	1.4111808398467916e-113
CommentChange	ALL	MixedLM	16769.06136267254	4.375792834703939e-23	8.126472407307315e-23
SemanticChange_Sum	KF0	Kruskal-Wallis	360.5263945012056	5.160456765011395e-79	2.367738986299346e-78
SemanticChange_Sum	KF0	Post-hoc pKF0 vs pKF1	777087.0	9.963956050477593e-51	2.5070599094750073e-50
SemanticChange_Sum	KF0	Post-hoc pKF0 vs pKF2	780280.5	6.405838873210104e-53	1.72294976589789e-52

Change Type	Code Variant	Statistical Test	Test Statistic	p-value	Corrected p-value (FDR)
SemanticChange_Sum	KF0	Post-hoc pKF1 vs pKF2	613038.0	0.745910343166119	0.7555974904799647
SemanticChange_Sum	KF1	Kruskal-Wallis	470.77443977503543	5.924194277587001e-103	3.0805810243452406e-102
SemanticChange_Sum	KF1	Post-hoc pKF0 vs pKF1	815773.5	9.201984490023579e-69	3.777656790641259e-68
SemanticChange_Sum	KF1	Post-hoc pKF0 vs pKF2	811538.5	1.6750601883174145e-65	6.532734734437917e-65
SemanticChange_Sum	KF1	Post-hoc pKF1 vs pKF2	606478.0	0.6191488593257453	0.643914813698775
SemanticChange_Sum	KF2	Kruskal-Wallis	426.92964665175464	1.965192731242888e-93	9.58031456480908e-93
SemanticChange_Sum	KF2	Post-hoc pKF0 vs pKF1	791845.0	2.5730617264629466e-56	7.433289432004068e-56
SemanticChange_Sum	KF2	Post-hoc pKF0 vs pKF2	803887.5	3.3518157934136494e-65	1.1883710540284757e-64
SemanticChange_Sum	KF2	Post-hoc pKF1 vs pKF2	620787.5	0.19000653259733524	0.20584041031377984
SemanticChange_Sum	ALL	MixedLM	25359.684091458075	2.303526455929644e-157	1.4972921963542684e-156
MixedChange	KF0	Kruskal-Wallis	37.78660200742396	6.233667436748707e-09	9.724521201327981e-09
MixedChange	KF0	Post-hoc pKF0 vs pKF1	627833.0	0.0008475923754967419	0.0010838066440778012
MixedChange	KF0	Post-hoc pKF0 vs pKF2	638128.0	3.4715196780454304e-09	5.526092548725379e-09
MixedChange	KF0	Post-hoc pKF1 vs pKF2	620518.5	0.004795142108726402	0.005936842610804116
MixedChange	KF1	Kruskal-Wallis	32.47346704418149	8.881293598148576e-08	1.3321940397222865e-07
MixedChange	KF1	Post-hoc pKF0 vs pKF1	635996.5	1.194563019211175e-05	1.5792528050588414e-05
MixedChange	KF1	Post-hoc pKF0 vs pKF2	638069.5	1.6402373785655565e-06	2.32615482778388e-06
MixedChange	KF1	Post-hoc pKF1 vs pKF2	612310.5	0.6865982805481233	0.7046666563520213
MixedChange	KF2	Kruskal-Wallis	45.69353662595356	1.1961195598521304e-10	2.0732739037436928e-10
MixedChange	KF2	Post-hoc pKF0 vs pKF1	624935.5	0.01038670548476132	0.012091985489722134
MixedChange	KF2	Post-hoc pKF0 vs pKF2	643146.0	7.466667252698853e-12	1.3544187109546754e-11
MixedChange	KF2	Post-hoc pKF1 vs pKF2	628277.0	4.268344639522096e-06	5.840892664609184e-06

Change Type	Code Variant	Statistical Test	Test Statistic	p-value	Corrected p-value (FDR)
MixedChange	ALL	MixedLM	37085.53203031491	0.33200813434153664	0.3547484175156145
Other	KF0	Kruskal-Wallis	43.30883096306202	3.9410037447265016e-10	6.682571567144937e-10
Other	KF0	Post-hoc pKF0 vs pKF1	587774.5	9.98044949209716e-05	0.0001297458433972631
Other	KF0	Post-hoc pKF0 vs pKF2	621472.0	0.008328398012689022	0.009994077615226827
Other	KF0	Post-hoc pKF1 vs pKF2	643691.5	5.584222620797366e-10	9.267433285578608e-10
Other	KF1	Kruskal-Wallis	26.033930247014233	2.2223060865588513e-06	3.0953549062784e-06
Other	KF1	Post-hoc pKF0 vs pKF1	594925.5	0.013845497179588467	0.015651431594317398
Other	KF1	Post-hoc pKF0 vs pKF2	624748.5	0.005085121019515771	0.006197491242534846
Other	KF1	Post-hoc pKF1 vs pKF2	639581.5	4.2735343287151774e-07	6.172882919255257e-07
Other	KF2	Kruskal-Wallis	35.67717824679583	1.789778055016991e-08	2.7373076135553982e-08
Other	KF2	Post-hoc pKF0 vs pKF1	585019.5	8.472866118191149e-06	1.13945440899812e-05
Other	KF2	Post-hoc pKF0 vs pKF2	613769.0	0.4479188010527574	0.4721306281366902
Other	KF2	Post-hoc pKF1 vs pKF2	638993.5	3.3022311715127355e-07	4.85988738449044e-07
Other	ALL	MixedLM	29233.498890334085	2.967884338659644e-09	4.822812050321921e-09

Statement on the Usage of Generative Digital Assistants

For this thesis, the following generative digital assistants have been used:

We have used ChatGTP for code completion and text rewriting. We employed ChatGPT primarily to paraphrase paragraphs, with a particular focus on refining transitions. In addition, we used the tool to convert text passages and CSV tables into LaTeX format, including LaTeX-compatible tables. Beyond these tasks, ChatGPT also supported us in writing parts of functions as well as entire smaller functions, in a manner comparable to the assistance provided by an AI assistant integrated within a code editor.

We are aware of the potential dangers of using these tools and have used them sensibly with caution and with critical thinking.

Bibliography

[1] M. Wyrich, J. Bogner, and S. Wagner, "40 years of designing code comprehension experiments: A systematic mapping study," vol. 56, no. 4, pp. 1–42. [Online]. Available: http://arxiv.org/abs/2206.11102

[2] R. Tiarks, "What maintenance programmers really do: An observational study."

[3] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "Understanding understanding source code with functional magnetic resonance imaging," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, pp. 378–389. [Online]. Available: https://dl.acm.org/doi/10.1145/2568225.2568252

[4] N. Peitek, J. Siegmund, C. Parnin, S. Apel, J. C. Hofmeister, and A. Brechmann, "Simultaneous measurement of program comprehension with fMRI and eye tracking: a case study," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/10.1145/3239235.3240495

[5] J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. Hofmeister, C. Kästner, A. Begel, A. Bethmann, and A. Brechmann, "Measuring neural efficiency of program comprehension," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, pp. 140–150. [Online]. Available: https://dl.acm.org/doi/10.1145/3106237.3106268

[6] Z. Zheng, K. Ning, J. Chen, Y. Wang, W. Chen, L. Guo, and W. Wang, "Towards an understanding of large language models in software engineering tasks." [Online]. Available: http://arxiv.org/abs/2308.11396

[7] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review." [Online]. Available: http://arxiv.org/abs/2308.10620

[8] R. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.

[9] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, pp. 121–130. [Online]. Available: https://dl.acm.org/doi/10.1145/1390630.1390647

[10] R. P. L. Buse and W. R. Weimer, "Learning a metric for code readability," vol. 36, no. 4, pp. 546–558. [Online]. Available: http://ieeexplore.ieee.org/document/5332232/

[11] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk, "A comprehensive model for code readability," vol. 30, no. 6, p. e1958. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/smr.1958

[12] M. Halstead, *Elements of Software Science*. Elsevier.

[13] S. Wagner and M. Wyrich, "Code comprehension confounders: A study of intelligence and personality," pp. 1–1. [Online]. Available: https://ieeexplore.ieee.org/document/9611030/

[14] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability," vol. 47, no. 3, pp. 595–613. [Online]. Available: https://ieeexplore.ieee.org/document/8651396/

[15] A. Vitale, V. Piantadosi, S. Scalabrino, and R. Oliveto, "Using deep learning to automatically improve code readability," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, pp. 573–584. [Online]. Available: https://ieeexplore.ieee.org/document/10298369/

[16] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages." [Online]. Available: http://arxiv.org/abs/2002.08155

[17] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, and X. Peng, "Exploring the potential of ChatGPT in automated code refinement: An empirical study," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3623306

[18] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model."

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need."

[20] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training."

[21] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners."

[22] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, and T. Henighan, "Language models are few-shot learners."

[23] OpenAI, J. Achiam, S. Adler, and A. et al., "GPT-4 technical report." [Online]. Available: http://arxiv.org/abs/2303.08774

[24] M. Chen, J. Tworek, and J. et al., "Evaluating large language models trained on code." [Online]. Available: http://arxiv.org/abs/2107.03374

[25] H. Tian, W. Lu, T. O. Li, X. Tang, S.-C. Cheung, J. Klein, and T. F. Bissyandé, "Is ChatGPT the ultimate programming assistant – how far is it?" [Online]. Available: http://arxiv.org/abs/2304.11938

[26] K. Jin, C.-Y. Wang, H. V. Pham, and H. Hemmati, "Can ChatGPT support developers? an empirical evaluation of large language models for code generation." [Online]. Available: http://arxiv.org/abs/2402.11702

[27] Z. Liu, Y. Tang, X. Luo, Y. Zhou, and L. F. Zhang, "No need to lift a finger anymore? assessing the quality of code generation by ChatGPT." [Online]. Available: http://arxiv.org/abs/2308.04838

[28] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo, "Refining ChatGPT-generated code: Characterizing and mitigating code quality issues." [Online]. Available: http://arxiv.org/abs/2307.12596

[29] X. Yu, L. Liu, X. Hu, J. W. Keung, J. Liu, and X. Xia, "Fight fire with fire: How much can we trust ChatGPT on source code-related tasks?" number: arXiv:2405.12641. [Online]. Available: http://arxiv.org/abs/2405.12641

[30] E. A. AlOmar, A. Venkatakrishnan, M. W. Mkaouer, C. D. Newman, and A. Ouni, "How to refactor this code? an exploratory study on developer-ChatGPT refactoring conversations." [Online]. Available: http://arxiv.org/abs/2402.06013

[31] K. DePalma, I. Miminoshvili, C. Henselder, K. Moss, and E. A. AlOmar, "Exploring ChatGPT's code refactoring capabilities: An empirical study," vol. 249, p. 123602. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0957417424004676

[32] C. Hu, Y. Chai, H. Zhou, F. Meng, J. Zhou, and X. Gu, "How effectively do code language models understand poor-readability code?" in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ACM, pp. 795–806. [Online]. Available: https://dl.acm.org/doi/10.1145/3691620.3695072

[33] R. Liu, A. Frade, A. Vaidya, M. Labonne, M. Kaiser, B. Chakrabarti, J. Budd, and S. Moran, "On iterative evaluation and enhancement of code quality using GPT-4o." [Online]. Available: http://arxiv.org/abs/2502.07399

[34] I. Shumailov, Z. Shumaylov, Y. Zhao, N. Papernot, R. Anderson, and Y. Gal, "AI models collapse when trained on recursively generated data," vol. 631, no. 8022, pp. 755–759. [Online]. Available: https://www.nature.com/articles/s41586-024-07566-y

[35] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "An empirical study of the non-determinism of ChatGPT in code generation," vol. 34, no. 2, pp. 1–28. [Online]. Available: http://arxiv.org/abs/2308.02828

[36] N. v. Stein, A. V. Kononova, L. Kotthoff, and T. Bäck, "Code evolution graphs: Understanding large language model driven design of algorithms." [Online]. Available: http://arxiv.org/abs/2503.16668

[37] Q. Song, X. Kong, L. Wang, and B. Li, "An empirical investigation into the effects of code comments on issue resolution," in *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, pp. 921–930. [Online]. Available: https://ieeexplore.ieee.org/document/9202422/

[38] F. Zhao, J. Zhao, and Y. Bai, "A survey of automatic generation of code comments," in *Proceedings of the 2020 4th International Conference on Management Engineering, Software Engineering and Service Sciences*. ACM, pp. 21–25. [Online]. Available: https://dl.acm.org/doi/10.1145/3380625.3380649

[39] Y. Zhao, R. Zhang, W. Li, D. Huang, J. Guo, S. Peng, Y. Hao, Y. Wen, X. Hu, Z. Du, Q. Guo, L. Li, and Y. Chen, "Assessing and understanding creativity in large language models," vol. 22, no. 3, pp. 417–436. [Online]. Available: http://arxiv.org/abs/2401.12491