1 Pilot Experiment

Our main experiment aims to analyze how LLMs modify 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 LLMs themselves consider 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, our 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 an LLM'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$_{\textbf{Pilot}}$:** What is ChatGPTs' perspective on factors of code understandability (= *key factors*)?
- **RQ2$_{\textbf{Pilot}}$:** 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$_{\text{Pilot}}$) 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$_{\text{Pilot}}$). 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.

1.1 Pilot Experiment Design

Key Factors of Code Understandability (RQ1$_{Pilot}$). To answer our first preliminary research question (RQ1$_{\text{Pilot}}$), 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$_{\text{Pilot}}$). 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 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 $RQ2_{Pilot}$, examining in our pilot experiment how consistently these factors are applied when ChatGPT iteratively refines code.

Qualitative Analysis ($RQ2_{Pilot}$). To answer 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 1 visualizes this process.

1.1.1 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.

1.1.2 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 1) are inherently difficult to assess in isolated code snippets.

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

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 1. Results of the *Thematic Analysis* on ChatGPT's "Understanding" of Key Factors of Code Understandability

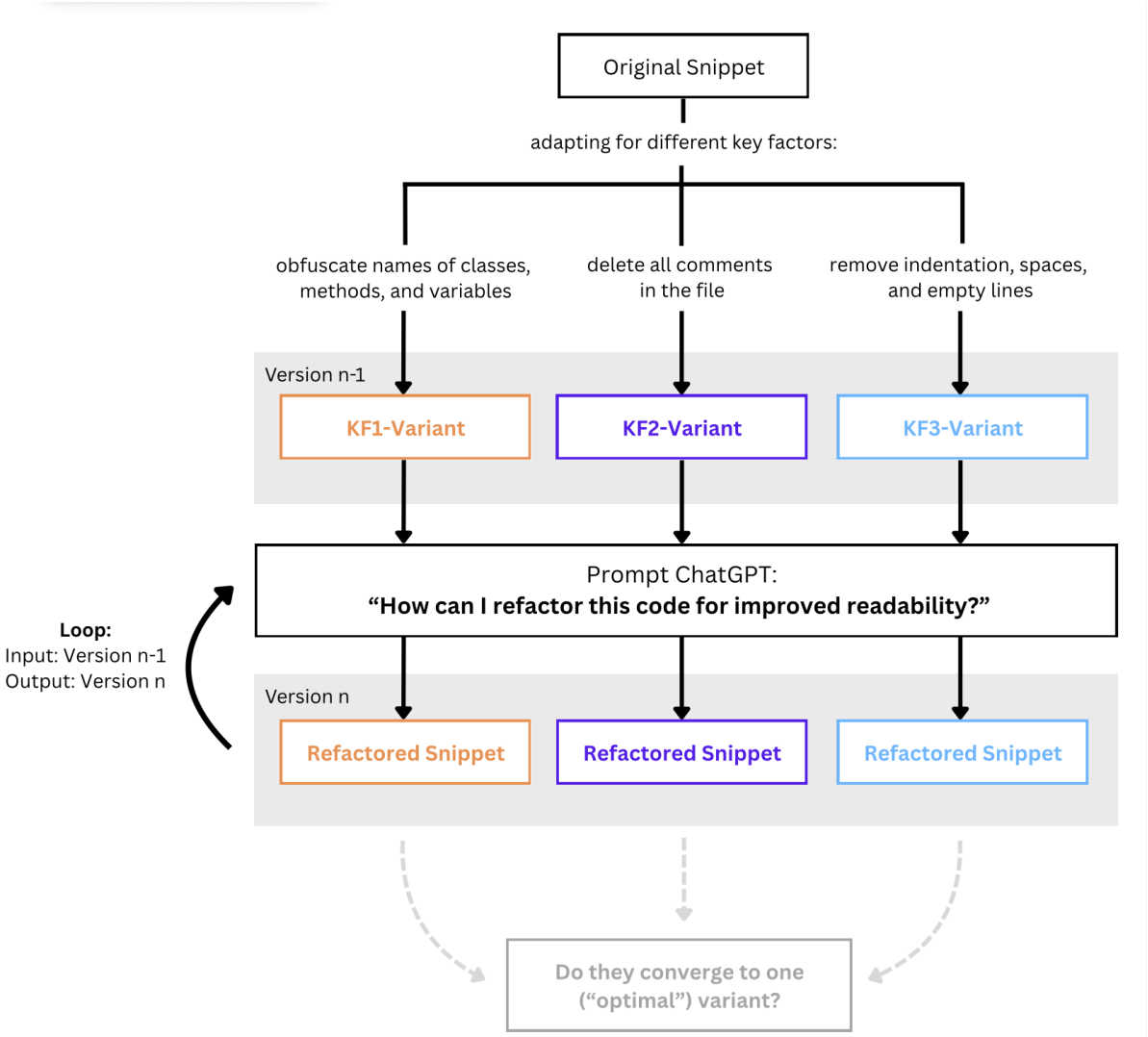

Fig. 1. Schematic Pilot Experiment Setup

- 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:

- **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.).

[inline]fix this figure

1.1.3 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.

1.1.4 Results.

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

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 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 counterproductive 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 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 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 work.

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 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.

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 [1], ensuring more stable and reproducible outputs across different iterations of the experiment.