Do Prompt Patterns Affect Code Quality? A First Empirical Assessment of ChatGPT-Generated Code

Antonio Della Porta
adellaporta@unisa.it
University of Salerno
Salerno, Italy

Stefano Lambiase
slambiase@unisa.it
University of Salerno
Salerno, Italy

Fabio Palomba
fpalomba@unisa.it
University of Salerno
Salerno, Italy

ABSTRACT

Large Language Models (LLMs) have rapidly transformed software development, especially in code generation. However, their inconsistent performance, prone to hallucinations and quality issues, complicates program comprehension and hinders maintainability. Research indicates that *prompt engineering*—the practice of designing inputs to direct LLMs toward generating relevant outputs—may help address these challenges. In this regard, researchers have introduced *prompt patterns*, structured templates intended to guide users in formulating their requests. However, the influence of prompt patterns on code quality has yet to be thoroughly investigated. An improved understanding of this relationship would be essential to advancing our collective knowledge on how to effectively use LLMs for code generation, thereby enhancing their understandability in contemporary software development. This paper empirically investigates the impact of prompt patterns on code quality, specifically maintainability, security, and reliability, using the Dev-GPT dataset. Results show that Zero-Shot prompting is most common, followed by Zero-Shot with Chain-of-Thought and Few-Shot. Analysis of 7583 code files across quality metrics revealed minimal issues, with Kruskal-Wallis tests indicating no significant differences among patterns, suggesting that *prompt structure may not substantially impact these quality metrics in ChatGPT-assisted code generation.*

CCS CONCEPTS

• **Computing methodologies → Information extraction**; • **Software and its engineering → Software reliability**; *Automatic programming*.

KEYWORDS

Prompt Engineering; Prompt Patterns; Source Code Quality; Empirical Software Engineering

ACM Reference Format:
Antonio Della Porta, Stefano Lambiase, and Fabio Palomba. 2018. Do Prompt Patterns Affect Code Quality? A First Empirical Assessment of ChatGPT-Generated Code. In *Proceedings of The 29th International Conference on Evaluation and Assessment in Software Engineering (EASE 2025).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Generative AI, particularly through Large Language Models (LLMs), is driving significant changes across various domains [10], including software engineering [15]. In this domain, LLMs are transforming workflows by automating code generation, assisting in building software components, aiding in decision-making, and identifying maintenance issues [15, 19]. As LLM integration expands, researchers are actively studying how these models can support software engineering tasks, documenting their impact on productivity, quality, and problem-solving in development workflows [3, 12, 20, 24, 30, 40].

Although LLMs have demonstrated substantial contributions to software development tasks [2, 19], such as enhancing developers' productivity, they are not without limitations. Their performance often lacks consistency and remains vulnerable to "hallucinations," referring to the generation of incorrect or irrelevant outputs [16]. Consequently, when employing LLMs for tasks such as code generation, their outputs cannot be assumed accurate or ready for immediate use. Instead, users must rigorously validate the results and frequently engage in prolonged interactions with the tool to achieve the desired outcome. This requirement undermines the initial promise of increased productivity and efficiency, limiting the tool's effective and potentially transformative benefits. Therefore, studying and acquiring foundational knowledge to effectively utilize LLM in the shortest possible time has become essential.

To address these challenges, researchers have increasingly focused on *prompt engineering* defined as *the practice of crafting precise inputs to improve LLM performance by guiding responses without modifying model parameters* [41]. Recent studies have indeed shown that performance may vary by as much as 45.48% between optimal and suboptimal prompts in some models, underscoring the sensitivity of outputs to prompt design [9]. This emphasis on prompt quality has led to the development of *prompt patterns* [49], which are structured templates akin to design patterns that offer reproducible frameworks for optimizing LLM output. Examples of these patterns include formats that encourage step-by-step reasoning [17], define specific personas [23], or provide a few illustrative examples [11], each crafted to enhance different aspects of LLM performance. These patterns have consistently proven effective in producing more reliable, contextually relevant outputs, leading to greater accuracy in code generation [1].

While current research acknowledges the impact of prompt patterns on code generation, *there is a notable lack of studies investigating how these patterns affect the quality of generated code.* Given the existing body of knowledge, it is reasonable to expect that prompt design could significantly influence aspects such as code maintainability, security, and reliability. However, empirical studies in this area remain limited, partly due to the challenges of constructing datasets that capture real-world usage scenarios. In response to this gap, Xiao et al. [54] recently introduced the Dev-GPT dataset, a collection of developer interactions with ChatGPT. Later on, Wu et al. [53] conducted a preliminary analysis of prompt patterns with a focus on code quality.

In order to contribute filling this research gap and provide novel knowledge to the field of prompt engineering, we defined the following objective for this investigation.

◎ Paper Objective

Our objective was to advance understanding of the role of ***prompt patterns*** *in code generation through an investigation into their relationship with various aspects of* ***code quality****, focusing on maintainability, security, and reliability.*

The scientific novelty lies in the analysis of how specific prompt patterns could influence key quality attributes in generated code, with direct implications for software developers. By examining these dimensions, we provide insights into how prompt engineering can be refined to produce more understandable, secure, and maintainable code, ultimately supporting developers in interpreting and trusting LLM-generated outputs in diverse software development contexts. The prompt patterns analyzed have been informed by the work of Hou et al. [19] where we selected a set of the prompt patterns that the authors considered the most easy to use.

Moreover, since the dataset used in the work contains only code and conversations that were made using ChatGPT, the scope and implications of this work can't be generalized to all the LLMs. Nonetheless, since ChatGPT is among the most widely used LLM by practitoners in software development [42], our findings impacts a large percentage of practitioners.

Concretely, this study provided three main contributions:

(1) An **empirical analysis of prompt patterns and code quality**, revealing no statistically significant relationship between the prompt patterns analyzed and code quality across maintainability, reliability, and security dimensions.
(2) A **data contribution** through a refined version of the Dev-GPT dataset, with duplicates and redundancies removed and enriched with metadata on conversation topics, prompt patterns, and code quality metrics.
(3) A **publicly available online appendix** [4], containing all data and scripts used in this study to support transparency and reproducibility.

Structure of the paper. Section 2 summarizes the related literature and how our work advances the current state of the art. Section 3 introduces the research questions driving our work and the research methods employed to address them. Section 4, we present the results of the study, while Section 5 provides the actionable implications that our work has for researchers, educators, and practitioners. In Section 6, we discuss the limitations of the work and how we mitigated them while designing the study. Finally, Section 7 concludes the paper and outlines our future research agenda.

2 RELATED WORK

Our work builds on the insights from several studies in the realm of Large Language Models for software engineering.

In the context of LLM usage, a prompt is defined as a *"set of instructions or input data provided to a Large Language Model to guide its output"* [35]. Effective prompt design has been shown to directly shape the model's generated responses. For instance, Brown et al. [8] demonstrated that *prompt engineering*, that is, the practice of crafting precise inputs to improve the capabilities of LLMs [57], may sometimes yield results comparable to or even better than model fine-tuning, which requires extensive task-specific training data to adjust the model's pre-trained weights. Indeed, prompt engineering leverages the model's existing knowledge, allowing for optimized performance without the need for additional training.

Sasaki et al. [41] further defined *prompt engineering patterns* as *"a systematic approach to structuring interactions, providing a versatile framework applicable across various domains"*. The significance of prompt engineering in software engineering has been pointed out by White et al. [50], who found that effective prompt usage can enhance early stages of the software development lifecycle. Similar results were found by Arvidsson and Axell [5] and Rodriguez et al. [39], who assessed the role of prompt engineering on requirements engineering and traceability recovery tasks, respectively.

Wang et al. [45, 46] examined prompt tuning in various code intelligence tasks, such as defect prediction, code search, code summarization, and code translation, demonstrating that prompt tuning consistently outperforms traditional fine-tuning across these areas. Yu et al. [55] investigated automated code review, showing that refined prompts can significantly improve the accuracy and comprehensibility of code assessments. O'Brien et al. [33] studied prompt effectiveness in code generation with GitHub Copilot, analyzing how prompts interact with *TODO* comments to influence code suggestions. Hagar and Masuda [18] explored prompt engineering in software testing, finding that customized prompts can assist users of varying expertise, from beginners to experts, in developing effective test architectures. More recent studies continued to expand the applications of prompt engineering in software engineering. Li et al. [28] proposed a ChatGPT-based approach for rapid source code development using structured prompts, which improved both the speed and quality of code generation. Fagadau et al. [14] empirically analyzed the impact of prompt variations on automated method generation with GitHub Copilot, offering insights into how different prompts can affect performance and accuracy in generated methods.

With respect to these previous studies, our work is complementary, as we focus specifically on the impact of prompt patterns on quality attributes. While previous research has demonstrated the effectiveness of prompt engineering in various tasks, including code generation, our work provides a deeper analysis of how prompt patterns may affect maintainability, security, and reliability of the code generated by LLMs.

A notable contribution to software engineering research was introduced by Xiao et al. [54], who proposed Dev-GPT, a dataset specifically designed to investigate how developers interact with ChatGPT in software development contexts. The dataset comprises 29 778 ChatGPT prompts and responses, including 19 106 code snippets, and is linked to software development artifacts such as source code, commits, issues, and pull requests. Sourced from shared ChatGPT conversations on GitHub and Hacker News, Dev-GPT provides a valuable resource for examining developer queries, the effectiveness of ChatGPT for code generation and problem-solving, and its broader impact on software engineering.

Building upon Dev-GPT, Wu et al. [53] examined the role of prompt patterns in improving developer-ChatGPT interactions throughout the software development lifecycle. The study focused on (1) analyzing the structure and duration of developer-ChatGPT conversations, (2) identifying prompt patterns that elicit high-quality responses, and (3) optimizing these patterns for software development tasks. As part of this investigation, the authors also evaluated the impact of individual prompt patterns on code quality metrics, assessing responses based on code size, complexity, and nesting levels. Leveraging the prompt patterns proposed by White et al. [48]—which include techniques for enhancing input semantics, customizing outputs, identifying errors, and refining prompts—the researchers assigned quality scores to ChatGPT responses based on an evaluation performed by models such as Code-Llama and Mistral. The study identified multiple patterns, e.g., *Output Customization* and *Error Identification*, as particularly effective, with the former emerging as the most frequently used and impactful pattern, especially in tasks like code generation and software management.

To the best of our knowledge, the work by Wu et al. [53] represented the first attempt to investigate the impact of prompt patterns on software quality, making it the closest study to our investigation. When comparing the two studies, multiple aspects should be noted. First, Wu et al. explored a broader range of software engineering tasks, while our study focuses on code generation, enabling a deeper analysis of how prompt patterns may impact multiple quality attributes of the ChatGPT-generated code. Additionally, Wu et al. relied on a catalog of prompt patterns [48] that has yet to be validated by the scientific research community and lacks evidence of adoption by practitioners [19].

In contrast, our study focuses on a set of prompt patterns detailed by Hou et al. [19] that are more commonly recognized by practitioners. We specifically selected a set of basic patterns used in a software engineering context, which are also the most easily usable by non-expert practitioners due to their simple structure.

Finally, since Wu et al.'s initial analysis, the Dev-GPT dataset has been expanded with new developer-ChatGPT interactions, enabling a more comprehensive exploration of prompt effectiveness.

Related Work: Summary and Contribution.

Existing studies demonstrate that prompt engineering can enhance a range of software engineering tasks, from requirements elicitation to software testing. However, the specific impact of prompt engineering patterns on quality attributes remains underexplored. While recent investigations have offered initial insights, our work delivers a more comprehensive analysis of these quality dimensions.

3 RESEARCH DESIGN

The *goal* of this work was to investigate the extent to which specific prompt patterns influence the quality attributes of source code generated by LLMs, with the *purpose* of expanding current knowledge in prompt engineering and potentially uncovering the side effects of code generation on maintainability, reliability, and security. More specifically, our work seeks to verify the following working hypothesis:

The use of different prompt patterns, which are hypothesized to affect the performance of LLMs in generating source code, could lead to variations in the quality of the generated code.

The hypothesis is supported by recent literature, e.g., [46, 50], which highlights the promising role of prompt patterns in improving model outputs. However, despite their potential, prompt patterns remain a developing area with definitions that lack explicit consideration of quality attributes that are critical to software engineering applications. For instance, current classifications include prompt types like "Zero-shot", which provide minimal structural guidance to the model. This limited focus on quality-related aspects differentiates prompt patterns from traditional design patterns, highlighting the need for an empirical investigation that aligns prompt engineering with quality outcomes, thereby directly supporting our hypothesis.

3.1 Research Questions

To reach the defined goal and test the working hypothesis, we formulated two research questions (**RQ**s) aiming to shape and guide the research process. In the following, we introduce each research question along with its motivation.

RQ$_1$: Prominence of Prompt Patterns. *What prompt patterns are most commonly used in conversations with ChatGPT?*

The first RQ seeks to identify the most commonly used prompt patterns in conversations with ChatGPT. We informed the selection of patterns utilizing the findings of [19], specifically selecting a set of basic patterns that practitioners more commonly recognize and use also due to their simple structure.

The specific choice of ChatGPT is that it is the most widely used LLM by practitioners in software development [42], and so our findings will impact a large percentage of practitioners. This first question must be considered *preliminary*, but *essential* for our research, as understanding the prominence of specific prompt patterns provides a foundational context to interpret the findings of the subsequent analysis. Additionally, this preliminary step may provide insights into the real-world usage of prompt patterns, thereby informing researchers on which patterns are most likely to be applied in practice and guiding future work on optimizing prompt design for practical applications.

RQ$_2$: Prompt Patterns and Code Quality. *Is there a statistical difference in the quality of ChatGPT-generated code when using different prompt patterns?*

The second research question forms the core of this work, guiding the primary statistical analysis of the potential relationship between different prompt patterns and the quality of generated source code. Addressing this question is essential for achieving the study's overall objective, as it provides insights into how specific prompt patterns may affect key quality attributes in code. For researchers, these insights expand current knowledge on prompt engineering while offering practitioners practical guidance on selecting prompt patterns to meet quality-related goals in real-world

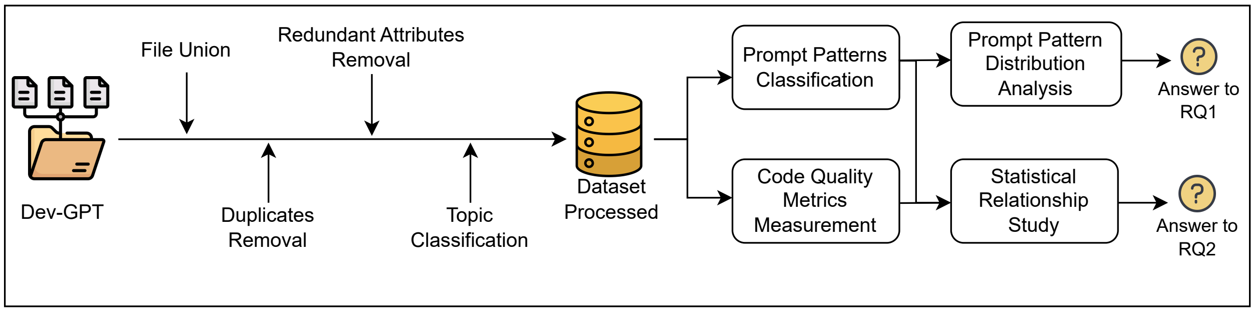
File Union
Redundant Attributes Removal
Dev-GPT
Duplicates Removal
Topic Classification
Dataset Processed
Prompt Patterns Classification
Code Quality Metrics Measurement
Prompt Pattern Distribution Analysis
Statistical Relationship Study
Answer to RQ1
Answer to RQ2

Figure 1: Overview of the Research Method Employed in Our Study.

applications. Figure 1 overviews the design of the research procedure employed to address our two research questions. In particular, we adopt multiple steps, as reported in the following:

(1) First, a dataset of conversations between practitioners and ChatGPT was selected. This dataset, Dev-GPT, developed during prior research [54], was deemed relevant for investigating prompt patterns in software development [53]. Various data preparation steps were undertaken, with the most critical being topic classification, which allowed the identification of conversations and prompts relevant to the target users of our investigation, such as developers.
(2) Second, we used LLM-based classification to analyze each conversation. Moreover we also employed a human validation process to ensure the quality of the results. These were applied to classify the prompt pattern used by the developer (addressing **RQ$_1$**).
(3) Third, we used the well-known SonarQube to compute quality metrics (specifically, the number of issues related to maintainability, security, and reliability) of the source code generated by the LLM in response to user requests.
(4) Finally, for each quality metric (our dependent variables), a statistical test was conducted to determine whether there were statistically significant differences in the metrics between the source code generated in response to different prompt patterns (addressing **RQ$_2$**). Since the characteristics of the data, we employed Kruskal-Wallis test as a non-parametric statistical test.

In terms of study design, we followed the guidelines by Wohlin et al. [51]. In terms of reporting, we adhered to the *ACM/SIGSOFT Empirical Standards*[1]; In particular, we leveraged the *"General Standard"*, and *"Repository Mining"* guidelines ensuring to have all the essential attributes and also some desirable attributes. For example, as described in the guidelines we summarized and discussed about the related works and also compared the contributions of the present works in relation to them.

3.2 Variables of the Study

The variables of the study were two: the prompt pattern used in the conversation (independent variable) and the quality metrics (dependent variable) of the generated source code.

[1]Available at: https://github.com/acmsigsoft/EmpiricalStandards

Prompt Patterns. Regarding the independent variable, it was operationalized using a categorical scale consisting of four prompt pattern categories and their combination, frequently discussed in the literature [19]:

- *Zero-shot* (ZS) [36]: It involves providing no examples within the prompt, relying solely on the model's pre-existing knowledge to generate responses.
- *Few-shot* (FS) [32]: It includes a small set of examples within the prompt to help guide the model's understanding and response generation.
- *Chain-of-Thought* (CoT) [47]: it uses prompts that encourage step-by-step reasoning, aimed at enhancing the model's logical consistency and detail in responses.
- *Personas* [22]: It implements a specific, consistent character or tone in the prompt, fostering contextualized responses aligned with particular roles or perspectives.

The selection of these categories was deliberate, as we recognize the existence of additional prompt patterns. First, the decision was made to complement the work of Wu et al. [53], who relied on a different scale developed during the early phases of LLM development, which has since evolved into the scale used in this study. Second, the chosen patterns are the basic patterns present in literature [19], and the most easily usable also by non-expert practitioners due to their simple structure. Given the preliminary nature of this work, we argue that focusing on these prominent patterns is more reasonable than using a broader, yet less representative, set.[2]

Quality Metrics. Regarding the dependent variable, we operationalized three proxies of code quality such as (1) *number of maintainability issues*; (2) *number of reliability issues*; and (3) *number of security issues*. The choice of these proxies come from three main considerations. First and foremost, these metrics capture a complementary and broad spectrum of quality attributes, offering insights into the multi-faceted nature of code quality. Indeed, maintenance, reliability, and security provide critical perspectives on how code performs over time, both in terms of robustness and ease of future adaptations [21, 37]: as such, their inclusion

[2]As a methodological note, initially, we attempted to include all identified prompt patterns in our scope. However, due to (1) excessive similarity between some patterns and (2) the limited occurrence of others in the dataset, the results were unsatisfactory. After multiple iterations with different configurations, we decided to focus on the four key prompt patterns reported in the article.

allowed us to assess software quality in a more comprehensive manner, addressing not only immediate functionality but also the longer-term aspects essential for sustainable and resilient code. In the second place, the combination of these metrics aligns with industry standards, such as *ISO/IEC 25002*,[3] and has been widely supported in software quality literature for evaluating factors that affect the sustainability, stability, and security of code over its lifecycle [6, 29, 43]. Last but not least, these proxies align with practical considerations in the industry, where automated tools like SonarQube are commonly used to assess these attributes [7, 44].

3.3 Context of the Study and Data Preparation

The dataset analyzed in this study, named Dev-GPT [54], consists of conversations between developers and ChatGPT focused on various software development tasks. Dev-GPT was assembled using OpenAI's conversation-sharing feature. The dataset consists of 6 `json` files representing different data sources used to elicit conversations such as GitHub issues, pull requests, discussions, commits, code files, and threads on Hacker News. To capture the evolving nature of these interactions, data snapshots were taken at multiple intervals, each reflecting the state of the dataset at a particular time.

The dataset includes separate `json` files for each source where the conversations had been taken. In total, Dev-GPT comprises 5494 conversation rounds, 29 788 total prompts containing 13 988 individual code snippets written across 113 different programming languages and frameworks.

Our initial intention was to use the dataset as-is. However, a preliminary manual review by the first two authors revealed that the dataset contained duplicate conversations and, more critically, included generic conversations that fell outside the scope of software development. For example, one of the user prompts found in the dataset was the following:

> *"Does USB-C without Thunderbolt support two 4k @60Hz monitors?"*

To ensure that the dataset was reliable, relevant, and representative of software development conversations, we undertook a preliminary data preparation process. We began by merging the various files that make up Dev-GPT, removing duplicate conversations and unnecessary attributes. Next, we applied a topic classification to filter the conversations, retaining only those that were pertinent to software engineering and development. Given the large volume of prompts, we used a LLM-based classification to expedite this process, relying on ChatGPT 4o-mini.[4] Since the accuracy of this classification is critical to the quality of our analysis, we also involved 10 human experts recruited from our network to validate the LLM-generated classifications. A statistically significant sample of 380 prompts was divided among the experts, each of whom reviewed a subset of the prompts using a simple web application we developed (available in the online appendix [4]). This application allowed users to upload a `json` file containing prompts for sequential review, following the same instructions given to ChatGPT. The experts confirmed all classifications made by the LLM, increasing our confidence in the filtering process used to retain only conversations relevant to software engineering and software development. Consequently, we used this filtered dataset for subsequent analyses.

3.4 Classification of Prompt Patterns

To address the research questions, it is necessary to analyze user interactions with ChatGPT, focusing on identifying and classifying instances of the four prompt patterns selected for the study within these interactions. To classify the prompt patterns in the filtered dataset, we developed an automated LLM-based classification mechanism. As in the data preparation stage, we validated this mechanism through manual analysis to ensure accuracy.

LLM-Based Prompt Pattern Classification. We aimed to use the LLAMA 3.1 70B[5] LLM, but some initial tentative showed that it had difficulties on some very long prompts that the dataset contains. Due to hardware limitations, we were not able to overcome this obstacle with other open-source models, and this led us to opt for ChatGPT-4o mini (snapshot *2024-07-18*). This model was chosen among the others due to a sufficiently large context window to process extensive prompts required for our experimentation. Comparisons were made to ChatGPT 4o, but the results did not diverge in a meaningful way, so we opted for using 4o mini given the lower cost per million tokens of input.

To improve the performance of the LLM, we adopted the self-refine approach, as described by Madaan et al. [31], where the model not only produces the initial outcome—in our case, the detection of prompt patterns—but also generates feedback and refines its results based on them. To put the self-refine approach in practice, we needed to develop three different prompts for each step of the process[6]: the initial classification step, the feedback step, and the refinement step. To increase the results quality and make the process reliable, those prompts were built using *prompt engineering techniques*, like Meta-Prompting [56], Personas [22] and Emotion Prompting [27]. After drafting the initial prompt, we conducted an *iterative prompt refinement process*, testing its performance on a small subset of the dataset. This approach ensured that the responses adhered to the specified task while improving the prompt to minimize model hallucinations. A notable adjustment was made to the feedback prompts for classification tasks, which now explicitly constrain the model to reason concisely within a fixed word limit. This change addressed issues in the initial prompts, which were often overly verbose, repetitive, and prone to inferring user intentions that were not explicitly stated. Concretely, consider the following user prompt contained in the dataset (*Conversation_ID=4, first user prompt*):

User Prompt

```
import cors from 'cors';
import axios from 'axios';
import fs from 'fs';
import express from 'express';
```

[3]The ISO/IEC 25002 standard: https://www.iso.org/standard/78175.html.
[4]For the sake of space limitation, more details on the prompt used in this stage are available in our online appendix [4].
[5]https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_1
[6]All the prompts used are available in the online appendix [4].

```
import https from 'https';

const options = {
    key: fs.readFileSync('path-to-key')
        ,
    cert: fs.readFileSync('path-to-cert
        ')
};

const app = express();
let promptResponse = {};
```

In this prompt, the user provides code (we only reported a part of it) and seeks information about its usage (without asking for explicit help). We will take this prompt as an example to comprehend the functioning of the SELF-REFINE approach better. For this case, we were performing the task of classifying if a prompt employs a zero-shot or a few-shot prompting pattern. Following the SELF-REFINE, the prompt used were:

(1) *Initial Classification Prompt*: This generates the initial classification and accompanying reasoning by the LLM. This explicit reasoning is then used in the feedback step to generate constructive feedback. Considering the exemplary prompt, its initial classification was the following:

Initial Classification

Classification: *Few-shot*;
Reasoning: The prompt consists of multiple code snippets and detailed implementation examples of server-side functionality using Express, Axios, and other libraries. This indicates it is part of a broader instructional context, typical of few-shot prompts that provide examples or detailed instructions to clarify a coding-related query or task.

(2) *Feedback Prompt*: This prompt validates the classification and generates feedback. To reach a convergence to a final classification of all the prompts, this prompt is explicitly written to ask the LLM to write feedback if—when performing the classification task again—it has some suggestions and feedback to provide to the next execution of the task. If the LLM has no feedback or suggestions and so agrees with the classification, it has to return *"None"* as the answer. In the exemplary case, the feedback provided by the LLM was the following:

Feedback Provided

Feedback: The classification as "Few-shot" is misleading because the text does not provide indicative examples or different variations of prompts that typically characterize few-shot learning. Instead, it consists of detailed implementation code without direct contextual examples geared for specific tasks.

(3) *Refinement Prompt*: This is used when feedback is generated in the previous step. This prompt is developed to instruct the LLM to perform the classification task again, using the instructions of Step #1 but with the additional input provided by the feedback. During the refinement process, the LLM produced the following classification for the exemplary case, which is actually the correct one:

Refinement Classification

Classification: *Zero-shot*;
Reasoning: The prompt consists solely of detailed implementation code without any concrete examples or variations to guide the desired outputs or tasks. There are no explicit instructions or comparative prompting typical of few-shot learning. Thus, it fits the characteristics of a zero-shot prompt, where the task is presented without demonstration.

In a preliminary analysis, we observed that the SELF-REFINE approach did not consistently converge to a final answer across all elements in the dataset. To address this, we implemented two stopping conditions: (1) when the feedback step produces no new suggestions, and (2) when a fixed number of iterations is reached. During experimentation, we found that by the fifth iteration, the agreement rate between the feedback step and the initial classifications declined significantly, indicating diminishing returns from further iterations. Based on this trend, we set the maximum number of iterations to 5 to optimize the balance between accuracy and efficiency in the classification process.

Automated Classification Validation. While the automated classification allowed us to streamline the data collection process, potential misclassifications could impact the reliability of the conclusions of the study. To assess the extent of these misclassifications, we conducted a manual evaluation of the automated mechanism. This evaluation aimed to ensure that the classifications met a high standard of accuracy, addressing the known limitations of LLMs in sometimes producing inaccurate information. For this purpose, two authors acted as *independent inspectors*, each reviewing a representative sample of the classifications to assess their validity. A random sample of 10% of the total classifications produced by the LLM was selected for evaluation—we deemed this sample sufficiently large to assess the automated classifier. The two inspectors independently analyzed the conversations in the sample, labeling each conversation according to the corresponding prompt pattern. To mitigate potential confirmation bias [34], they conducted their evaluations without prior knowledge of the classifications assigned by the automated mechanism and without any discussion of specific cases.
Following their independent assessments, the two inspectors convened for a focused, two-hour in-person meeting to discuss their findings. During this session, they addressed any disagreements and reached consensus on all classifications. This collaborative step resulted in a manually-curated oracle, which could be compared against the classifications made by the automated classifier. The comparison revealed a 97% match between the automated and manual classifications, indicating a high level of accuracy

in the automated mechanism. This high level of agreement indicated that the automated classification mechanism was largely accurate, providing confidence in the validity of our conclusions and suggesting a limited margin of error.

3.5 Extraction of Source Code Quality Metrics

To evaluate the impact of prompt patterns on code quality, we computed software quality metrics for the source code generated by the prompts defined in the dataset using SonarQube. We decided to use SonarQube because (1) the industry uses it since it is a well-known tool used in those contexts to analyze code quality, and (2) it has been used in various software engineering research papers to assess code quality [13, 25, 26] as a proof of its reliability. Starting with the source code snippets generated by ChatGPT in the dataset, we created individual code files, assigning file extensions based on the programming language indicated in the dataset. We then executed SonarQube on each file, retrieving results via the SonarQube Web API. The quality metrics obtained for each snippet were recorded in the dataset. For snippets that encountered errors during compilation or analysis, we annotated these cases accordingly to exclude them from the final set of successfully analyzed files, ensuring accurate results.

3.6 Data Analysis

To address **RQ$_1$**, we examined the distribution of prompt patterns identified by the automated classifier within the filtered dataset. We measured the frequency of each pattern individually (e.g., only Zero-Shot or only Chain-of-Thought) as well as in combination with other patterns (e.g., Zero-Shot with Chain-of-Thought, Few-Shots with Personas).

To answer **RQ$_2$**, we conducted statistical analyses by formulating the following null and alternative hypotheses:

- **Null Hypothesis:** There are no significant differences in the number of ($H1_0$) maintainability, ($H2_0$) reliability, and ($H3_0$) security issues in the source code generated by ChatGPT based on the prompt pattern used.
- **Alternative Hypotehsis:** There are significant differences in the number of ($H1_a$) maintainability, ($H2_a$) reliability, and ($H3_a$) security issues in the source code generated by ChatGPT based on the prompt pattern used.

To assess differences across multiple groups, we initially selected ANOVA for our analysis. However, as the assumptions for ANOVA were not met, we instead employed the non-parametric Kruskal-Wallis test with Dunn's post hoc test to evaluate the hypotheses, using a significance threshold of $\rho = 0.05$. The statistical analysis was conducted using JASP.[7]

4 ANALYSIS OF THE RESULTS

Following the experimentation process stated in the previous section, in this section we will delve into the main findings and answer the posed research questions.

[7]JASP website: https://jasp-stats.org

Table 1: Ranking of prompt patterns usage.

Rank	Prompt Pattern	#Occurrences
1	Zero-shot	10 034
2	Zero-shot with CoT	713
3	Few-shot	576
4	Zero-shot with Personas	334
5	Few-shot with Personas	134
6	Zero-shot with CoT and Personas	107
7	Few-shot with CoT	94
8	Few-shot with CoT and Personas	49

4.1 RQ$_1$—Prompt Patterns Prominence

The complete, filtered dataset included 3188 conversations and a total of 27 065 user prompts directed to ChatGPT.

Table 1 presents the final distribution, revealing a strong preference for Zero-Shot prompting, with 10 034 instances, far surpassing other patterns such as Zero-Shot with Chain-of-Thought (713 instances) and Few-Shot (576 instances). This preference for Zero-Shot prompting likely reflects developers' inclination toward simplicity and efficiency, as it requires minimal setup and leverages the model's built-in capabilities.

This finding may be due to various reasons. First, Zero-Shot prompts are highly adaptable across a variety of coding tasks, allowing developers to quickly gauge the model's capabilities without the need for extensive customization, which is particularly beneficial in exploratory phases of development. Second, this pattern has a low barrier to entry, which may be attractive to developers unfamiliar with advanced prompting techniques, as more complex patterns like Chain-of-Thought or Personas require a higher level of understanding and structuring to maximize their effectiveness. Consequently, Zero-Shot prompting may be preferred as it aligns well with the time-efficient, agile workflows often seen in software development, where minimal setup enables rapid iterations and fast prototyping.

Another possible interpretation of this preference is that developers may rely on the model's pre-trained knowledge and reasoning capabilities to produce quick, actionable insights or initial drafts without extensive guidance, trusting that the model will yield satisfactory results. However, the lower use of advanced patterns, such as Chain-of-Thought or Personas, could suggest an underutilization of techniques that may provide significant advantages in scenarios demanding nuanced reasoning or context-specific responses. Thus, while Zero-Shot prompting remains the default for everyday use, these findings also highlight an opportunity for further training on prompt engineering, which could help developers make better use of advanced patterns to address complex tasks effectively.

📊 RQ$_1$—Prompt Patterns Presence in Conversation

The most used prompt pattern in developers conversations is Zero-Shot as shown in Table 1 with 10 034 occurrences, Zero-Shot with CoT with 713 occurrences and Few-Shot with 576 occurrences.

Table 2: Descriptive Statistics of the code quality metrics.

	Maintainability	Reliability	Security
# of Occurrences	7624	7624	7624
Median	0.000	0.000	0.000
Mean	0.413	0.095	0.002
Std. Deviation	1.855	0.472	0.047
Minimum	0.000	0.000	0.000
Maximum	30.000	12.000	2.000

4.2 RQ$_2$—Prompt Patterns and Code Quality

We generated a total of 8748 code files from the selected conversations for evaluation by SonarQube. Of these, 369 Java files were excluded from analysis due to SonarQube license limitations. Out of the remaining 8379 files, 755 failed to pass the compilation stage and were subsequently excluded from further analysis. To prevent misinterpretation of these uncompiled files as error-free, we annotated their corresponding `json` entries with a flag, clearly indicating that these files were not analyzed.

The metrics calculated on the final pool of 7583 files are presented in Table 2. The *'Security'* dimension was the least frequent in the dataset, with only 50 occurrences, indicating few security issues across the analyzed code snippets. In contrast, *'Maintainability'* had the highest number of issues, totaling 3400 across all files, followed by *'Reliability'*, with 750 issues. These overall metrics suggest that maintainability is the most frequently affected quality dimension, which may reflect the general coding practices and limitations of prompt patterns in sustaining code maintainability.

Detailed results by prompt pattern are shown in Tables 3, 4, and 5. For maintainability, the ZS configuration shows the highest number of occurrences (6534), with a mean of 0.433 and a substantial standard deviation of 1.953, indicating notable variability in maintainability issues within this pattern. This high variability suggests that while many ZS-generated snippets are maintainable, some exhibit significant maintainability concerns. In contrast, the FS-CoT-Personas configuration, though less frequent (7 occurrences), exhibits the highest mean maintainability issues (1.286) with a lower standard deviation (1.604), indicating a smaller but more concentrated set of issues in this pattern. The median for maintainability remains 0 across all configurations, implying that the majority of code snippets are free from maintainability issues, which is consistent across reliability and security metrics. For reliability, mean values are generally low, ranging from 0.000 to 0.429, with FS-CoT-Personas showing the highest mean and a slightly elevated standard deviation of 1.134, indicating some inconsistency but generally fewer issues than maintainability. Security issues are particularly sparse, with mean values close to zero for all configurations and a maximum value of 2.000 only in FS-CoT-Personas, highlighting the rarity of significant security issues in the considered code snippets.

These distributions indicate that, while the FS-CoT-Personas configuration may correlate with slightly higher incidences of all issue types, most prompt patterns, particularly ZS, result in minimal issues across reliability and security. This analysis suggests that while there are some differences across prompt patterns, the overall impact on code quality metrics is limited, with most configurations yielding generally issue-free code, especially regarding reliability and security. From a statistical standpoint, two key decisions were made based on the observed distributions: (1) the FS-CoT-Personas configuration was excluded from the analysis due to its low occurrence count, and (2) given the substantial number of outliers, we performed the analysis twice—once including all values and once excluding outliers. This dual approach ensured a thorough and consistent evaluation, as the outliers lacked a discernible pattern or explanation. Notably, the results of the no-outliers analysis mirrored those of the full dataset. The findings of the statistical test, including outliers, are detailed in Table 6, with a summary of key results provided below.

- *'Maintainability'*: The test produced a statistic of 3.801 with 6 degrees of freedom, yielding a ρ-value of 0.704. This indicates no statistically significant differences in maintainability across treatments. The effect size (Rank $\epsilon^2 = 4.996 \times 10^{-4}$) indicates a negligible effect.
- *'Reliability'*: The test statistic is 11.583 with 6 degrees of freedom, and a ρ-value of 0.072. While this ρ-value approaches significance, it is greater than 0.05, indicating no significant difference across treatments. The effect size of 0.002 suggests a minimal impact.
- *'Security'*: The test statistic is 2.144 with 6 degrees of freedom, with ρ-value=0.906. This indicates a lack of significant differences across treatments. The Rank ϵ^2 of 2.818×10^{-4} reflects a negligible effect.

RQ$_2$—Prompt Patterns and Source Code Quality

The statistical test revealed no statistically significant differences among the prompt patterns for any of the issue variables, resulting in the rejection of all three alternative hypotheses and the confirmation of null hypotheses.

5 DISCUSSIONS AND IMPLICATIONS

This section provides a contextual analysis of our findings and highlights potential directions for future research.

5.1 On the Practitioner's Use of Prompt Patterns

The results of **RQ$_1$** indicate a clear preference for Zero-Shot prompting among developers, with significantly higher usage than more complex patterns like Few-Shot, Chain-of-Thought, or Personas. As observed in the analysis of the results, this trend may be attributed to the *simplicity, immediacy, and adaptability of Zero-Shot prompting*, which aligns with the efficiency demands of developer workflows, particularly in the exploratory and prototyping phases. By minimizing setup time and requiring no additional context, Zero-Shot prompts allow developers to quickly gauge model capabilities across a wide range of tasks, making it an appealing choice for practitioners focused on speed and agility. Another key factor influencing this preference may be the *current knowledge and familiarity level within the developer community regarding advanced prompt engineering techniques.* As a relatively novel area, prompt engineering is still gaining traction, and many practitioners may be less familiar with patterns that involve higher levels of structuring, such as Chain-of-Thought or Personas. This lack of familiarity, combined

Table 3: Descriptive statistics for Maintainability issues.

MAINTAINABILITY	ZS	ZS-CoT	FS	ZS-Personas	FS-Personas	ZS-CoT-Personas	FS-CoT-Personas	FS-CoT
# of Occurrences	6534	441	390	129	21	32	7	63
Median	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000
Mean	0.433	0.263	0.236	0.488	0.190	0.281	1.286	0.365
Std. Deviation	1.953	0.881	0.827	2.020	0.512	0.683	1.604	1.406
Minimum	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Maximum	30.000	10.000	7.000	14.000	2.000	3.000	4.000	10.000

Table 4: Descriptive statistics for Reliability issues.

RELIABILITY	ZS	ZS-CoT	FS	ZS-Personas	FS-Personas	ZS-CoT-Personas	FS-CoT-Personas	FS-CoT
# of Occurrences	6534	441	390	129	21	32	7	63
Median	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Mean	0.101	0.059	0.051	0.109	0.000	0.125	0.429	0.032
Std. Deviation	0.491	0.318	0.291	0.419	0.000	0.554	1.134	0.177
Minimum	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Maximum	12.000	3.000	4.000	2.000	0.000	3.000	3.000	1.000

Table 5: Descriptive statistics for Security issues.

SECURITY	ZS	ZS-CoT	FS	ZS-Personas	FS-Personas	ZS-CoT-Personas	FS-CoT-Personas	FS-CoT
# of Occurrences	6534	441	390	129	21	32	7	63
Median	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Mean	0.002	0.000	0.000	0.000	0.000	0.000	0.286	0.000
Std. Deviation	0.045	0.000	0.000	0.000	0.000	0.000	0.756	0.000
Minimum	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Maximum	1.000	0.000	0.000	0.000	0.000	0.000	2.000	0.000

Table 6: Results of the Kruskal-Wallis statistical test.

Fixed Factor	Statistic	df	ρ-value	Rank ϵ^2
Maintainability	3.801	6	0.704	$4.996 * 10^{-4}$
Reliability	11.583	6	0.072	0.002
Security	2.144	6	0.906	$2.818 * 10^{-4}$

with the additional effort required to set up these prompts effectively, may explain why developers gravitate toward Zero-Shot prompts by default, even if other techniques could potentially offer enhanced results for tasks requiring deeper reasoning or contextual awareness. Thus, the findings support previous research indicating that simplicity and usability often drive the adoption of LLM tools in software engineering tasks [38].

These findings highlight an opportunity for further *dissemination* and *education*. As prompt engineering continues to evolve, increasing awareness of these techniques could enable developers to select patterns more strategically, tailoring them to the complexity and specificity of their tasks.

> 👉 **Implication 1**: **Researchers** should develop a standardized catalog of prompt patterns for code generation tasks, capturing not only the patterns themselves but also guidelines for their effective use in specific scenarios. **Educators** can play a key role in disseminating this knowledge by integrating prompt engineering strategies into software engineering curricula, thereby equipping practitioners with a broader toolkit to tackle diverse coding challenges more effectively.

5.2 Prompt Patterns vs Source Code Quality

In addressing $\mathbf{RQ_2}$, our analysis reveals that different prompt patterns do not produce statistically significant differences in maintainability, reliability, or security metrics. Although there is some variability in descriptive statistics, particularly within Few-Shot, Chain-of-Thought, and Personas configurations, the Kruskal-Wallis test yielded negligible effect sizes across all metrics, suggesting that prompt patterns alone may not be a decisive factor in influencing code quality.

One explanation for these findings is that ChatGPT are inherently designed to respond effectively to basic prompt structures, making them relatively insensitive to finer adjustments introduced by more structured prompt patterns. Additionally, as prior work highlights [48], users' unfamiliarity with advanced prompt engineering techniques may contribute to the limited usage of complex patterns, as observed in $\mathbf{RQ_1}$. This low adoption of advanced patterns could, in turn, restrict researchers' ability to fully understand how these patterns might impact code quality, particularly in complex or high-stakes coding tasks. Therefore, further evaluations of complex prompt patterns may be worthwhile, especially as practitioners' familiarity with prompt engineering continues to grow.

☞ **Implication 2**: Findings suggest that **practitioners** can often achieve satisfactory quality results in code generation tasks using simple prompting techniques, such as Zero-Shot prompting. However, for **researchers**, this study highlights the need for additional evaluations of complex prompt patterns to better understand their impact on code quality in specialized contexts.

5.3 Evaluating Prompt Patterns and Quality Implications in Dev-GPT Code Generation

Our findings suggest that most code generated by ChatGPT is relatively free from major quality issues. However, maintainability issues are more prevalent than reliability or security concerns, indicating that ChatGPT may sometimes struggle with structural or stylistic aspects that support maintainable code.

On the one hand, these observations imply that the current set of quality metrics may not fully capture the dimensions of how different prompt patterns impact the quality of generated code. While simple prompts appear to yield satisfactory results for the considered tasks, more complex scenarios may require refined, context-sensitive metrics that better associate prompt patterns with specific quality outcomes. By developing qualitative metrics, researchers could gain a more accurate understanding of how prompt engineering influences code quality, especially in cases where advanced prompt patterns might play a significant role. On the other hand, the dataset itself could benefit from additional diversity in task types and complexity. Expanding Dev-GPT to include more varied coding scenarios would allow for a deeper evaluation of how advanced prompt patterns function across different contexts. Such diversification would enhance the study of how prompt engineering affect code quality, supporting both routine and complex development needs.

☞ **Implication 3**: The **research community** should prioritize qualitative analyses and diversified datasets in prompt engineering research. Developing context-sensitive metrics and datasets with varied coding tasks would enable an improved understanding of code quality in LLM outputs, benefiting both standard and specialized software development tasks.

6 THREATS TO VALIDITY

This study presents several threats to validity that we considered as a result of our design [52].

Threats to Internal Validity. Threats in this category concern the extent to which observed effects can be attributed to the variables studied rather than other factors. In this study, code quality assessments may be affected by the limitations of SonarQube, potentially impacting the consistency of maintainability, reliability, and security metrics. Additionally, biases may be introduced through the iterative self-refinement process with the LLM if feedback loops fail to converge. This risk was mitigated by implementing a maximum iteration limit and conducting manual verification by domain experts.

Threats to Construct Validity. Threats in this category refer to the accuracy with which study measures capture the concepts they intend to represent. In this research, categorizing prompt patterns into specific types—Zero-shot, Few-shot, Chain-of-Thought, and Personas—may oversimplify how prompt patterns affect code generation, as nuances in pattern design and implementation are not fully captured. Patterns were selected based on prevalent literature to address this. Furthermore, employing AI tools to support the classification process may have introduced some inaccuracies, as LLMs are known for occasional hallucinations. However, automatic analysis was supplemented with a manual review conducted by both the authors and external experts, ensuring robust results and mitigating potential risks.

Threats to External Validity. These threats concern the generalizability of findings beyond the study's specific conditions. The use of a single dataset (Dev-GPT) and a specific LLM model (ChatGPT-4o mini) may limit the applicability of results to other datasets or models. In this respect, future work should consider additional datasets and alternative LLMs to validate the results across different contexts.

Threats to Conclusion Validity. Threats in this category examine whether the statistical analyses accurately reflect relationships between variables. In this study, the Kruskal-Wallis test, a non-parametric alternative to ANOVA, helps to address potential issues related to normality and variance assumptions in the data. However, this test's reliance on rank-based analysis may still introduce limitations, particularly in detecting subtle effects. Care was taken to set significance thresholds to minimize the risk of Type I and II errors, thereby enhancing the robustness of the study's findings.

7 CONCLUSIONS

This study explored the relationship between prompt patterns and the quality of source code. An automated prompt classification mechanism, powered by ChatGPT, was employed to (1) isolate prompts specifically related to software engineering and (2) identify the presence of four distinct prompt pattern types. The generated code from ChatGPT, as documented in the Dev-GPT dataset, was assessed for maintainability, reliability, and security using the widely recognized tool SonarQube. The collected data supported an empirical analysis to investigate the correlation between prompt patterns and the aforementioned code quality metrics. The results indicated that there is no statistically significant relationship between the analyzed prompt patterns and code quality across the evaluated dimensions. As part of our future research we aim to deepen the analysis by considering more dimensions of code quality such as functional correctness or code smells.

ACKNOWLEDGMENT

This work has been partially supported by the *Qual-AI* national research project, which has been funded by the MUR under the PRIN 2022 program (Code: D53D23008570006).

REFERENCES

[1] Seif Abukhalaf, Mohammad Hamdaqa, and Foutse Khomh. 2023. On Codex prompt engineering for OCL generation: an empirical study. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 148–157.

[2] Anisha Agarwal, Aaron Chan, Shubham Chandel, Jinu Jang, Shaun Miller, Roshanak Zilouchian Moghaddam, Yevhen Mohylevskyy, Neel Sundaresan, and Michele Tufano. 2024. Copilot Evaluation Harness: Evaluating LLM-Guided Software Programming. *ArXiv* abs/2402.14261 (2024). https://doi.org/10.48550/arXiv.2402.14261

[3] Alexandre Agossah, Frédérique Krupa, Matthieu Perreira Da Silva, and Patrick Le Callet. 2023. Llm-based interaction for content generation: A case study on the perception of employees in an it department. In *Proceedings of the 2023 ACM International Conference on Interactive Media Experiences*. 237–241.

[4] Anon. [n. d.]. Online Appendix. ([n. d.]). https://figshare.com/s/91396667ee475a4b0a0b

[5] Simon Arvidsson and Johan Axell. 2023. Prompt engineering guidelines for LLMs in Requirements Engineering. (2023).

[6] Hala Assal and Sonia Chiasson. 2018. Security in the software development lifecycle. In *Fourteenth symposium on usable privacy and security (SOUPS 2018)*. 281–296.

[7] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 470–481.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[9] Bowen Cao, Deng Cai, Zhisong Zhang, Yuexian Zou, and Wai Lam. 2024. On the Worst Prompt Performance of Large Language Models. *arXiv preprint arXiv:2406.10248* (2024).

[10] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology* 15, 3 (2024), 1–45.

[11] Qi Cheng, Liqiong Chen, Zhixing Hu, Juan Tang, Qiang Xu, and Binbin Ning. 2024. A novel prompting method for few-shot ner via llms. *Natural Language Processing Journal* 8 (2024), 100099.

[12] Fiona Draxler, Daniel Buschek, Mikke Tavast, Perttu Hämäläinen, Albrecht Schmidt, Juhi Kulshrestha, and Robin Welsch. 2023. Gender, age, and technology education influence the adoption and appropriation of LLMs. *arXiv preprint arXiv:2310.06556* (2023).

[13] Dario Amoroso D'Aragona, Fabiano Pecorelli, Maria Teresa Baldassarre, Davide Taibi, and Valentina Lenarduzzi. 2023. Technical Debt Diffuseness in the Apache Ecosystem: A Differentiated Replication. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 825–833. https://doi.org/10.1109/SANER56733.2023.00095

[14] Ionut Daniel Fagadau, Leonardo Mariani, Daniela Micucci, and Oliviero Riganelli. 2024. Analyzing Prompt Influence on Automated Method Generation: An Empirical Study with Copilot. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 24–34.

[15] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53.

[16] Sebastian Farquhar, Jannik Kossen, Lorenz Kuhn, and Yarin Gal. 2024. Detecting hallucinations in large language models using semantic entropy. *Nature* 630, 8017 (2024), 625–630.

[17] Guhao Feng, Bohang Zhang, Yuntian Gu, Haotian Ye, Di He, and Liwei Wang. 2024. Towards revealing the mystery behind chain of thought: a theoretical perspective. *Advances in Neural Information Processing Systems* 36 (2024).

[18] Jon Hagar and Satoshi Masuda. 2024. Prompt Engineering Impacts to Software Test Architectures for Beginner to Experts. In *2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 116–121.

[19] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* (2023).

[20] Ranim Khojah, Mazen Mohamad, Philipp Leitner, and Francisco Gomes de Oliveira Neto. 2024. Beyond Code Generation: An Observational Study of ChatGPT Usage in Software Engineering Practice. *arXiv preprint arXiv:2404.14901* (2024).

[21] John Knight. 2012. *Fundamentals of dependable computing for software engineers*. CRC Press.

[22] Aobo Kong, Shiwan Zhao, Hao Chen, Qicheng Li, Yong Qin, Ruiqi Sun, Xin Zhou, Enzhi Wang, and Xiaohang Dong. 2024. Better Zero-Shot Reasoning with Role-Play Prompting. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. 4099–4113.

[23] Eva Krapp, Robin Neuhaus, Marc Hassenzahl, and Matthias Laschke. 2024. In a Quasi-Social Relationship With ChatGPT. An Autoethnography on Engaging With Prompt-Engineered LLM Personas. In *Proceedings of the 13th Nordic Conference on Human-Computer Interaction*. 1–16.

[24] Jahnavi Kumar and Sridhar Chimalakonda. 2024. Code Summarization without Direct Access to Code-Towards Exploring Federated LLMs for Software Engineering. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 100–109.

[25] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimaki, Savanna Lujan, and Fabio Palomba. 2023. A critical comparison on six static analysis tools: Detection, agreement, and precision. *Journal of Systems and Software* 198 (2023), 111575.

[26] Valentina Lenarduzzi, Alberto Sillitti, and Davide Taibi. 2020. A survey on code analysis tools for software maintenance prediction. In *Proceedings of 6th International Conference in Software Engineering for Defence Applications: SEDA 2018 6*. Springer, 165–175.

[27] Cheng Li, Jindong Wang, Yixuan Zhang, Kaijie Zhu, Wenxin Hou, Jianxun Lian, Fang Luo, Qiang Yang, and Xing Xie. 2023. Large language models understand and can be enhanced by emotional stimuli. *arXiv preprint arXiv:2307.11760* (2023).

[28] Youjia Li, Jianjun Shi, and Zheng Zhang. 2024. An Approach for Rapid Source Code Development Based on ChatGPT and Prompt Engineering. *IEEE Access* (2024).

[29] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A systematic mapping study on technical debt and its management. *Journal of Systems and Software* 101 (2015), 193–220.

[30] Mingxing Liu, Junfeng Wang, Tao Lin, Quan Ma, Zhiyang Fang, and Yanqun Wu. 2024. An empirical study of the code generation of safety-critical software using llms. *Applied Sciences* 14, 3 (2024), 1046.

[31] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems* 36 (2024).

[32] Ben Mann, N Ryder, M Subbiah, J Kaplan, P Dhariwal, A Neelakantan, P Shyam, G Sastry, A Askell, S Agarwal, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* 1 (2020).

[33] David OBrien, Sumon Biswas, Sayem Mohammad Imtiaz, Rabe Abdalkareem, Emad Shihab, and Hridesh Rajan. 2024. Are Prompt Engineering and TODO Comments Friends or Foes? An Evaluation on GitHub Copilot. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[34] Margit E Oswald and Stefan Grosjean. 2004. Confirmation bias. *Cognitive illusions: A handbook on fallacies and biases in thinking, judgement and memory* 79 (2004), 83.

[35] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.

[36] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[37] Miguel A Revilla. 2007. Correlations between internal software metrics and software dependability in a large population of small C/C++ programs. In *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*. IEEE, 203–208.

[38] Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended abstracts of the 2021 CHI conference on human factors in computing systems*. 1–7.

[39] Alberto D Rodriguez, Katherine R Dearstyne, and Jane Cleland-Huang. 2023. Prompts matter: Insights and strategies for prompt engineering in automated software traceability. In *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*. IEEE, 455–464.

[40] Daniel Russo. 2024. Navigating the complexity of generative ai adoption in software engineering. *ACM Transactions on Software Engineering and Methodology* (2024).

[41] Yuya Sasaki, Hironori Washizaki, Jialong Li, Dominik Sander, Nobukazu Yoshioka, and Yoshiaki Fukazawa. 2024. Systematic Literature Review of Prompt Engineering Patterns in Software Engineering. In *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 670–675.

[42] Agnia Sergeyuk, Yaroslav Golubev, Timofey Bryksin, and Iftekhar Ahmed. 2025. Using AI-based coding assistants in practice: State of affairs, perceptions, and ways forward. *Information and Software Technology* 178 (2025), 107610.

[43] Edith Tom, Aybüke Aurum, and Richard Vidgen. 2013. An exploration of technical debt. *Journal of Systems and Software* 86, 6 (2013), 1498–1516.

[44] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering* 25 (2020), 1419–1457.

[45] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R Lyu. 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*. 382–394.

[46] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. 2023. Prompt Tuning in Code Intelligence: An Experimental Evaluation. *IEEE Transactions on Software Engineering* 49, 11 (2023), 4869–4885. https://doi.org/10.1109/TSE.2023.3313881

[47] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[48] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).

[49] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. 2024. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. In *Generative AI for Effective Software Development*. Springer, 71–108.

[50] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. 2024. *ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design.* Springer Nature Switzerland, Cham, 71–108. https://doi.org/10.1007/978-3-031-55642-5_4

[51] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. 2012. *Experimentation in software engineering*. Vol. 236. Springer.

[52] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. 2012. *Experimentation in software engineering*. Vol. 236. Springer.

[53] Liangxuan Wu, Yanjie Zhao, Xinyi Hou, Tianming Liu, and Haoyu Wang. 2024. ChatGPT Chats Decoded: Uncovering Prompt Patterns for Superior Solutions in Software Development Lifecycle. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 142–146.

[54] Tao Xiao, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. 2024. Devgpt: Studying developer-chatgpt conversations. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 227–230.

[55] Yongda Yu, Guoping Rong, Haifeng Shen, He Zhang, Dong Shao, Min Wang, Zhao Wei, Yong Xu, and Juhong Wang. 2024. Fine-tuning large language models to improve accuracy and comprehensibility of automated code review. *ACM transactions on software engineering and methodology* (2024).

[56] Yifan Zhang. 2023. Meta prompting for agi systems. *arXiv preprint arXiv:2311.11482* (2023).

[57] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910* (2022).