Understanding Prompt Quality and Its Relation to LLM-based Code Generation

Antonio Della Porta, Stefano Lambiase, Valeria Pontillo, João F. Ferreira, Akond A. Rahman, Chaiyong Ragkhitwetsagul, Fabio Palomba

Abstract—Prompt engineering has emerged as a key practice to guide Large Language Models (LLMs) in code generation tasks, under the assumption that better prompts yield better outputs. Yet little is known about which characteristics of prompts actually drive variations in output quality. In this paper, we take a first step toward addressing this gap by investigating how measurable properties of prompts relate to the quality of generated code. We conducted an empirical study of real-world, single-turn interactions between developers and ChatGPT collected from GitHub. Prompts were characterized along readability and structural dimensions, while output quality was measured as the conceptual consistency to committed code, perceived usefulness, and correctness. Regression analysis shows that readability metrics are strong predictors of usefulness and correctness, whereas structural features such as the number of sentences and task type significantly affect conceptual consistency. Overall, our findings provide evidence that prompt quality is multi-dimensional and that measurable readability-related and structural prompt properties can serve as indicators of the correctness, usefulness, and conceptual consistency of LLM-generated code in software engineering.

Index Terms—Prompt Quality; Prompt Engineering; Empirical Software Engineering; LLM; Generative AI.

1 Introduction

Large Language Models (LLMs) are reshaping software engineering, powering tasks from code generation [1], [2] and summarization [3], [4] to code review, debugging, and testing [5]–[11]. Despite these advances, LLMs may threaten software quality, particularly in code generation [12]. Recent studies show that they can (i) hallucinate, producing code that diverges from developers' intent [13]; (ii) inject security weaknesses into otherwise correct-looking programs [14]; and (iii) reduce code quality and maintainability [15], [16], raising concerns about software quality assurance in development workflows [17]–[24]. These issues may arise either from the stochastic nature of LLMs and their training data [25], [26], or from the formulation of *prompts*, i.e., the requests provided by developers [27], [28]. This latter has received increasing attention under the label of *prompt engineering*, i.e., the study and practice of crafting prompts to guide LLMs toward more accurate and reliable outputs [29].

Much of the existing research in this field has focused on the effectiveness of prompting strategies, proposing approaches such as *prompt patterns* [30], iterative refinement methods [31], and reasoning-oriented techniques [32]. These studies provide valuable evidence that prompt design matters, yet they typically assess quality *indirectly*, by evaluating whether a given strategy improves task outcomes. This pragmatic perspective is useful: much like software testing, it reveals whether a specific input works under particular conditions. Nonetheless, current approaches consider prompt quality only through outcomes, treating it as a *by-product of output evaluation* rather than a construct with its own measurable characteristics. As a result, the **intrinsic properties of prompts themselves** remain underexplored. This leaves open a more fundamental issue: *What, in itself, makes a prompt "good"?* Without a principled understanding of this concept, it is hard to know whether existing strategies truly optimize the aspects of a prompt that matter for model performance, to develop systematic methods for improving prompts beyond trial-and-error, or to compare alternative strategies on common grounds. As such, the lack of a principled notion of prompt "goodness" represents a key gap in current research. In this paper, we take a first step toward addressing this gap. Our conjecture is that, since the ultimate goal of a "good" prompt is to produce a "good" output, its definition should be based on the idea that any measure of prompt quality must at least correlate with variations in the final output quality of LLMs.

◎ Objective of the Work

Building on these considerations, this paper presents **an empirical investigation aimed at identifying prompt-related metrics that correlate with variations in the quality of code generated by LLMs**. Our ultimate objective is to determine which characteristics can serve as candidates for defining a more concrete set of measures for prompt quality.

Antonio Della Porta and Fabio Palomba are with the Software Engineering (SeSa) Lab of the University of Salerno, Italy (e-mails: {adellaporta, fpalomba}@unisa.it).
Stefano Lambiase is with the Aalborg University, Denmark (e-mail: stla@cs.aau.dk).
Valeria Pontillo is with the Gran Sasso Science Institute, Italy (e-mail: valeria.pontillo@gssi.it).
João F. Ferreira is with the INESC-ID/IST, University of Porto, Portugal (e-mail: joao@joaoff.com).
Akond A. Rahman is with the Auburn University, Alabama (e-mail: akond@auburn.edu)
Chaiyong Ragkhitwetsagul is with Mahidol University, Thailand (e-mail: chaiyong.rag@mahidol.ac.th)

To this end, we focused on code generation for maintenance and evolution tasks and constructed a novel dataset capturing interactions between developers and LLMs on GitHub. We then identified and refined a candidate set of prompt quality metrics, organizing them along two dimensions: the *readability* of the request and its *structural* characteristics (e.g., the type of task performed and the composition of the prompt). Output quality was assessed using both similarity to developers' final code [33] and the ICE-Score [34], a metric known to align closely with human judgments. Finally, we analyzed the relationship between prompt metrics and output quality using regression models.

Based on our analysis, we found that the considered prompt quality metrics do indeed influence LLM outcomes, but their effects vary across dimensions of quality. Structural features were linked to the similarity between generated and developer-adapted code, while readability emerged as the strongest predictor of both usefulness and correctness. Other aspects, such as vocabulary richness and task type, showed only limited or inconsistent associations. These findings suggest that readability and structure in prompts are central to shaping the quality of LLM-generated code, providing initial evidence toward a more principled understanding of prompt quality.

To sum up, our article provides four main contributions:

1) A set of **prompt quality metrics**, elicited through a structured, multi-stage process involving experts, and organized along two dimensions: readability and structural characteristics;
2) An **empirical study** showing that prompt metrics correlate with variations in LLM-generated code quality;
3) A **dataset of developer–LLM interactions for maintenance and evolution tasks**, enabling the study of prompt quality in realistic scenarios;
4) A **publicly available replication package** [35], which supports reproducibility and further research.

2 Related Work

Our paper is related to prior research on prompt quality and applications of LLMs for software engineering (SE), which we discuss in the following subsections.

2.1 Prior Research on Applying LLMs to SE

Since the advent of LLMs, there has been a strong interest in their application to SE tasks. Research has investigated code generation for domain-specific languages such as Ansible and Kubernetes configuration scripts [1], [36], for object-oriented programming [2], [37], and for tasks like code summarization and specification generation [3], [4], [38]–[40]. Beyond generation, LLMs have been applied to quality assurance, including code review [5], automated debugging [6], testing [7], [8], static code analysis [41], configuration validation [42], and detection of security or reliability issues [43]–[45]. Another active line of work has focused on bug-related tasks, including localization [46], [47], repair [48], [49], and replay [50]. Complementary studies have examined developer perceptions of LLMs [51]–[53] and their role in supporting reproducibility of SE research [54].

Another line of research is that of Lambiase et al., who investigated various factors associated with the adoption of LLMs by software engineers [55], [56] and provided a comprehensive literature review on their use in conversational form [57]. Similarly, Tavantzis et al. conducted research focusing on how LLMs are adopted and for which specific purposes, trying to provide an empirically-grounded framework contribution [58], [59].

More focused on the prompt side, Jishan et al. [60] surveyed practitioners on the impact of user experience and prompt characteristics, finding that users tend to prefer results from longer and more structured prompts. This observation informed our catalog of metrics, which explicitly includes prompt length to test whether this perception holds in practice. Similarly, Fang et al. [61] introduced PCQA, a framework that treats prompts as conditioning factors when evaluating AI-generated content. Although their work focuses on multimodal generation, the underlying principle–that prompt characteristics play an important role in shaping output quality and should therefore be incorporated into its assessment–aligns with our motivation of treating prompt characteristics as measurable variables and empirically studying their relationship with output quality.

2.2 Prior Research on Prompt Analysis

Prior studies have examined how different aspects of prompt design influence LLM behavior in software engineering and beyond. Rawte et al. [62] linked hallucination rates to linguistic nuances such as readability, formality, and concreteness, showing that small stylistic differences can affect reliability. Santana Jr. et al. [63] evaluated prompting techniques (e.g., zero-shot, few-shot, decomposition, self-criticism) across multiple SE tasks, revealing that both strategy and language characteristics shape solution quality. Puerto et al. [64] showed that code-oriented prompting can elicit conditional reasoning in text-plus-code models, underscoring the role of structure in guiding depth of reasoning. Karmaker and Feng [65] proposed a TELeR, a general-purpose taxonomy for prompt design based on four properties, i.e., turn, expression, level of detail, and role, later applied in SE tasks such as test case generation [66], [67]. While such taxonomies provide a valuable conceptual framework for describing how prompts are structured, several of their dimensions (e.g., level of detail or role) rely on qualitative and context-dependent aspects, such as sub-task decomposition or evaluation criteria, that are not directly operationalizable as quantitative variables and would require manual annotation or heuristic encoding. Moreover, TELeR explicitly models multi-turn interaction settings, whereas our study focuses on single-turn prompts to isolate the effect of prompt characteristics on output quality and avoid confounding factors introduced by iterative interactions. For these reasons, we select a set of prompt-oriented metrics that are (i) compatible with single-turn interactions and (ii) fully operationalizable and automatically measurable, enabling large-scale empirical analysis of their impact on output quality in code generation tasks. Finally, Zhan et al. [68] highlighted the lexical sensitivity of LLMs, where minor lexical variations can produce disproportionately large differences in output quality. Overall, these studies establish that prompt design tangibly affects model behavior. The dimensions they explored, such as

readability, lexical variation, and prompt patterns, informed our design and are directly operationalized as variables in our study. Building on these works, our contribution moves beyond individual case analyses toward a structured, metrics-based investigation that systematically quantifies how prompt characteristics relate to multiple dimensions of output quality.

2.3 Prior Research on Evaluating LLM-Generated Code

A complementary line of work focuses on how the quality of LLM-generated code is assessed, providing direct insights for the selection of output quality metrics. The literature converges on three broad families of evaluation approaches. The first relies on *execution-based correctness*, where generated code is run against test suites to verify functional behavior; representative examples include HumanEval [12] and BigCodeBench [69], which have become de facto standards for benchmarking code generation. The second family adopts *similarity-based metrics*, where generated code is compared against a reference (typically human-written) using lexical or embedding-based measures; cosine similarity over learned representations [33] is a widely adopted instance, valued for its insensitivity to surface-level differences in length or naming. The third family encompasses *human-centered and judge-based evaluations*, which capture qualitative dimensions that execution and similarity cannot, such as perceived usefulness, readability, and alignment with developer expectations; this includes empirical studies on developer perception [51], [52] and, more recently, LLM-as-a-judge approaches [70] such as the ICE-Score [34], which has been validated against execution-based correctness across multiple programming languages. These three families are complementary rather than mutually exclusive: each captures a distinct facet of what it means for generated code to be "good".[1]

Research Gap

Researchers have not yet fully addressed how readability-related and structural properties of prompts translate into concrete software quality outcomes. Prior work has shown that such properties influence model behavior, but systematic, metrics-based evidence on their relationship with correctness, usefulness, and similarity of generated code is still lacking.

3 Objective, Variables, and Research Questions

The objective of this study is to identify prompt-related metrics that correlate with the quality of code generated by LLMs. This section presents the *conceptual foundations* of the study: the elicitation of prompt-quality dimensions and candidate metrics, the conceptualization of the resulting constructs, and the formulation of the research hypotheses. As a first step toward this objective, we had to elicit a candidate set of prompt-related metrics that could plausibly correlate with the quality of LLM-generated code. Such an elicitation process was grounded in a two-phase structured procedure, overviewed in Figure 1. This section presents the conceptual foundations of the study.

Phase 1: Four-Day Workshop. From December 2 to 5, 2024, five of the authors of this paper participated in a four-day Shonan Meeting on *"Anti-patterns and Defects: Synergies, Challenges, and Opportunities"*,[2] which focused on reshaping the concept of software quality in the era of LLMs. The event gathered about 25 internationally recognized researchers in software quality and AI-enabled systems. The workshop was organized around discussion-driven sessions, where participants rotated through small focus groups to generate innovative ideas. These results were subsequently shared and refined in plenary exchanges. Within this setting, the focus group involving the authors held three one-hour sessions where they developed a preliminary study design, iteratively improved through feedback from the other attendees. Building on this, the group also drafted a candidate set of prompt-quality metrics, i.e., the prospective independent variables of the study, and prompt-output indicators, i.e., the prospective dependent variables. In addition, they defined inclusion and exclusion criteria to guide their evaluation. These criteria were defined to ensure that the selected variables (both prompt-quality metrics and prompt-output indicators) were (i) coherent with each other, (ii) non-overlapping, and (iii) diverse enough to capture complementary perspectives. Both the initial metrics and the criteria were further refined through plenary discussion. As a result, we obtained a consolidated study design, which forms the basis of this submission, along with a preliminary catalog of candidate prompt-quality metrics and a set of criteria to guide subsequent refinement.

Phase 2: Discussion and Refinement. Building on the preliminary catalog of candidate metrics and the inclusion/exclusion criteria defined during the workshop, the authors organized a series of five bi-weekly meetings between February and May 2025. Each meeting, held online via Google Meet due to the geographical distribution of participants, lasted about one hour and focused on iteratively reviewing and refining the metric set. A shared document, available in the replication package [35], served as the central platform where new proposals were recorded and debated. Throughout this phase, the initial set of criteria was retained without changes, as it was collectively deemed sufficiently robust to guide the selection process. Metrics were systematically assessed against these criteria, and those failing to meet them were excluded. To further increase confidence in the metric set, the team complemented the expert-driven elicitation process with a literature-informed assessment of candidate metrics. The

1. As described in more detail in Section 4.2, this observation directly informed our study: rather than selecting a single evaluation paradigm, we operationalized output quality through three dimensions that draw on all three families: *conceptual consistency*, computed via cosine similarity to capture alignment with developer intent; *usefulness*, capturing the practical value perceived by developers; and *correctness*, capturing functional adequacy. The use of ICE-Score for the latter two reflects a deliberate methodological choice, as it provides a deterministic, reference-free judgment that scales to a dataset of developer-authored prompts where execution-based testing of arbitrary code is not feasible.

2. Shonan Meeting No. 207: https://shonan.nii.ac.jp/seminars/207/.

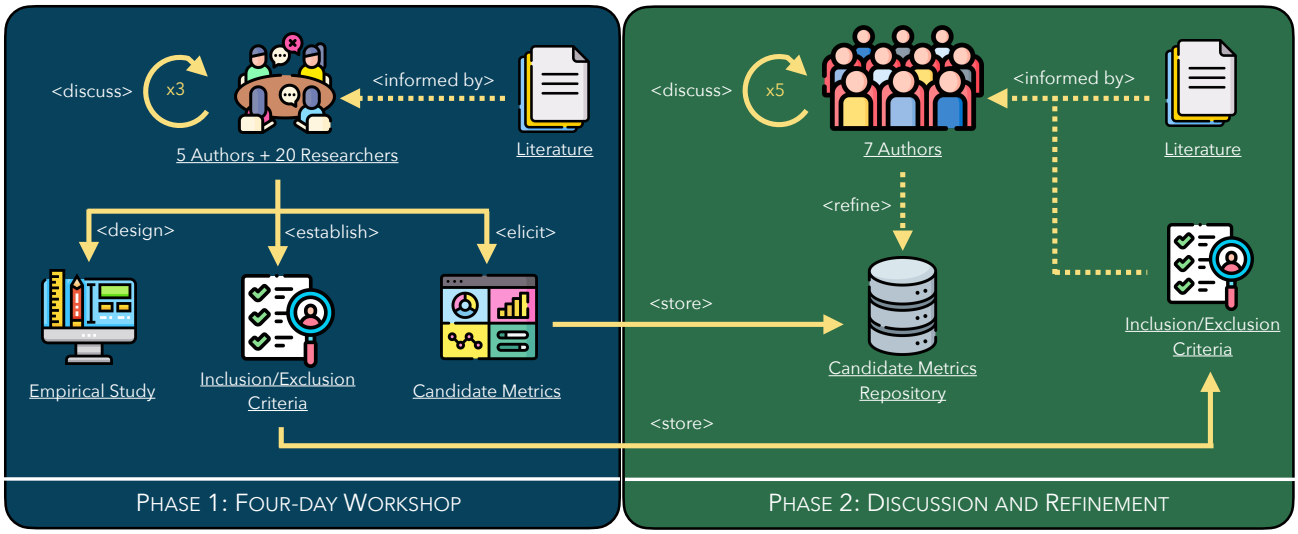

Fig. 1: Independent and Dependent Metrics Elicitation Process.

goal of this activity was not to conduct a full systematic literature review, but rather to ensure that the selected metrics were grounded in recent peer-reviewed research and reflected dimensions already discussed in the emerging literature on prompt engineering, LLM evaluation, and AI-assisted software engineering. More specifically, the fifth and sixth authors independently explored recent peer-reviewed literature following a shared lightweight scoping-review protocol [71]. Consistent with these guidelines, the process involved four main activities: (i) identification of the evidence sources, (ii) screening and selection of relevant studies, (iii) data extraction, and (iv) synthesis of the extracted evidence. For the identification phase, rather than performing a broad database-driven search based on generic keywords - which, given the breadth and rapid evolution of the area, would have produced a very large and potentially noisy set of results - we adopted a venue-driven screening strategy centered on flagship publication venues in Natural Language Processing and Software Engineering. The selected venues were manually screened by inspecting the proceedings and papers published in the considered timeframe. During the screening phase, papers were evaluated according to explicit inclusion and exclusion criteria aligned with the objectives and methodological constraints of our study. In particular, studies were included if they: (i) investigated prompt characteristics, prompt engineering strategies, or evaluation of LLM-generated artifacts; (ii) proposed or employed measurable prompt-related variables; and (iii) relied on metrics applicable without dynamic code execution, consistently with our focus on prompt-oriented and execution-independent properties. Conversely, papers were excluded if they focused exclusively on model architectures, training procedures, benchmark construction without prompt analysis, or evaluation approaches requiring execution-based validation only. For the data extraction phase, the authors collected candidate metrics, their definitions, and their operationalizations from the selected studies. Finally, during the synthesis phase, the extracted evidence was discussed in the bi-weekly meetings, where the full author team jointly evaluated the relevance, scope, redundancy, and applicability of the identified metrics with respect to the evolving candidate metric set. The exploration covered major venues in both Natural Language Processing and Software Engineering, including the Conference on Empirical Methods in Natural Language Processing (EMNLP), the Annual Meeting of the Association for Computational Linguistics (ACL), the North American Chapter of the Association for Computational Linguistics (NAACL), the International Conference on Learning Representations (ICLR), the ACM/IEEE International Conference on Automated Software Engineering (ASE), the International Conference on Software Engineering (ICSE), the ACM International Conference on the Foundations of Software Engineering (FSE), IEEE Transactions on Software Engineering (TSE), ACM Transactions on Software Engineering and Methodology (TOSEM), and Empirical Software Engineering (EMSE), covering publications from 2024 and 2025. These venues were selected because they are broadly recognized as flagship publication venues in NLP and Software Engineering and regularly publish peer-reviewed research on prompt engineering, LLM evaluation, generative AI systems, and AI-assisted software development. The two explorations were complementary in scope. The NLP-oriented exploration focused on prompt characteristics and evaluation metrics applicable without dynamic code execution, yielding five publications that informed the selection of readability- and structure-related metrics. In parallel, the SE-oriented exploration focused on studies involving LLM-generated natural-language artifacts and their evaluation in contexts where execution-based validation is not feasible. This complementary exploration identified one additional study proposing measurable properties for evaluating generated textual artifacts, several of which were subsequently incorporated into the refinement process. Although the exploration identified additional discussions on prompt engineering and LLM

evaluation, we did not observe additional measurable metrics beyond those ultimately retained in our final metric set. This increased our confidence that the selected variables adequately capture the most commonly investigated and operationalizable dimensions of prompt quality discussed in the recent literature.

The two-phase methodological process was designed to elicit a principled set of candidate metrics for assessing prompt quality. We acknowledge that prompt quality is a broad and rapidly evolving construct, and therefore, absolute completeness cannot be claimed; this would hold even in the case of a systematic search, given the fast-moving and exploratory nature of prompt engineering and LLM research. Nevertheless, the combination of an expert-driven elicitation process, explicit inclusion and exclusion criteria, and iterative refinement until saturation increases our confidence that the selected metrics adequately capture the most salient and diverse dimensions relevant to code generation. Future work may expand this catalog as the research community converges on additional perspectives. To support such efforts, the replication package includes the artifacts produced during the refinement process, offering a basis for further augmentation of the catalog and replication of the study [35]. In total, 15 prompt quality metrics and 4 output quality metrics were initially discussed. During the process, these were refined into a final set of 8 prompt quality metrics and 3 output quality metrics, which are presented in the following sections.

3.1 Dependent Variable—Characterizing the Quality of the Output

Based on the two-phase process just described, we operationalize output quality through three complementary dimensions: *conceptual consistency*, *usefulness*, and *correctness*.

Conceptual consistency. We define conceptual consistency as the degree to which the code generated by the LLM is reflected in the developer-adapted code that was committed on GitHub. The notion is adapted from the conceptual coherence of classes in object-oriented programming, which measures how well the elements of a class relate to a single, unified concept. More specifically, this dimension captures whether the generated and the developer-adapted code share the same underlying concepts and functionality, beyond surface-level textual or syntactic similarity. Prior research emphasizes that alignment between generated code and developer intent is essential for practical adoption. For example, Fang et al. [61] highlight that prompts condition the evaluation of generated content, underscoring that semantic alignment is a critical measure of quality. Similarly, studies on LLM-assisted programming tasks (e.g., code generation [2], [37] and code repair [48], [49]) show that developers prefer outputs that closely match their expected implementation, as this reduces integration effort and accelerates workflows. Moreover, recalling our initial consideration, it is reasonable to assume that a greater similarity between the two codes, i.e., generated and developer-adapted, reflects a greater ability of the LLM to meet the user's initial desires. Thus, we adopt semantic similarity as a proxy for alignment with user intent. To operationalize this notion of similarity, we adopt cosine similarity [33], a widely used metric in software engineering and natural language processing for capturing semantic closeness between artifacts. Cosine similarity is particularly suited here because it accounts for the overlap in terms while being insensitive to differences in code length. This property makes it a suitable proxy for comparing generated and developer-adapted code, where implementations may differ in size since developers may refine LLM outputs to meet the specific contextual demands of their systems.

Usefulness. Jishan et al. [60] found that practitioners' satisfaction depends not only on accuracy but also on how applicable and usable outputs are in context. This observation aligns with findings from perception studies [51], [52], which report that developers often judge outputs based on their ability to provide actionable help, even if the information is incomplete or approximate. We therefore include usefulness as a core dimension of quality, capturing the practical value of outputs beyond their formal correctness. In our work, this characteristic was operationalized using the ICE-Score [34], an *LLM-as-a-judge* approach [70] in which a model evaluates usefulness directly from the task description and the generated output. In this respect, an important consideration concerns the accuracy and reliability of the adopted evaluation approach. ICE-Score is explicitly designed to minimize hallucination and bias, unlike naïve LLM-as-a-judge approaches. It enforces deterministic inference by setting the temperature parameter to 0, uses fixed, explicit evaluation criteria, and relies on a structured decision template, thereby constraining the model to a bounded-scoring task rather than open-ended generation. In addition, it operates in a reference-free manner, preventing bias toward specific solutions or superficial similarity. Its validation [34] demonstrates a correlation with human preference and execution-based correctness across several programming languages and tasks, providing strong evidence of its robustness and reliability. The choice of using the ICE-score was also motivated by practical and methodological considerations: indeed, evaluating the usefulness and correctness of long, multi-function code snippets requires fine-grained, line-by-line inspection, which would be difficult to conduct manually at scale and could introduce fatigue-related variability, threatening reliability and repeatability. By adopting a deterministic and standardized evaluation process, we ensure full reproducibility across the dataset.

Correctness. Correctness remains a foundational dimension. Research in LLM-based debugging and testing consistently shows that outputs which appear relevant may still fail to execute or meet requirements [6], [8], [9]. For this reason, correctness is often explicitly measured in evaluations of LLMs across SE tasks [7], [38]. We again operationalized correctness using the ICE-Score [34]. Among its dimensions, the emphasis here was on whether the generated code fulfills the intended functionality. ICE-Score is particularly suitable for correctness because it provides a scalable judgment of functional adequacy without requiring code execution, while still reflecting how developers typically assess whether code *"does what it is supposed to do"*. Recent work has also adopted ICE-

Score specifically as a correctness measure. For instance, Payoungkhamdee et al. evaluated Program-of-Thought prompting for reasoning in cross- and multilingual settings using ICE-Score's correctness evaluation [72]. Similarly, Li et al. [73] proposed a ranking method to select stylistically consistent fine-tuning data and validated it using ICE-Score correctness. This prior usage further supports our choice, and ICE-Score's empirical validation [34] reports strong and consistent correlation with execution-based correctness across multiple programming languages, outperforming traditional string-based and neural evaluation metrics.

These dimensions build on insights from prior work: similarity addresses alignment with user intent, usefulness captures contextual applicability, and correctness ensures functional reliability. The concrete metrics used to operationalize these variables are detailed in Section 4.2.[3]

3.2 Independent Variable—Characterizing the Quality of the Prompt

Through the elicitation procedure, we selected seven independent variables covering both *readability-related* and *structural* properties of prompts.

Readability-related variables. This category was designed to capture the readability and expressiveness of prompts. Prior work [60], [62], [63], [68] indeed suggests that prompts written in a more readable and well-articulated manner are more likely to guide models toward producing useful outputs. We identified four specific metrics capturing these readability-related aspects:

- *Prompt Readability.* Clearer prompts reduce ambiguity, making it easier for the model to interpret user requests. Prior work shows that readability correlates with reduced hallucinations and better performance [62], [63]. In particular, Santana Jr. et al. [63] explicitly measured readability with the Flesch Reading Ease metric and observed its effect on model performance. We likewise operationalized this dimension using Flesch Reading Ease, which evaluates how easy a text is to read based on sentence length and syllable count [74].
- *Prompt Linguistic Complexity.* Higher linguistic complexity could hinder the model's ability to parse prompts effectively. Excessive complexity risks confusing the model or diluting the intent behind the request. Santana Jr. et al. [63] explicitly measured the linguistic complexity using the Gunning Fog Index, a readability metric estimating the years of formal education required to understand a text [75]. We adopted the same metric in our study, where higher values indicate greater linguistic complexity and potentially lower prompt interpretability for the model.
- *Vocabulary Richness.* A diverse vocabulary can enrich semantic meaning and improve LLM outputs, but may also introduce terms not well represented in training data. Empirical evidence links lexical diversity to better performance in language models [68]. We operationalized this dimension using the *Yule's K* [76] metric, which measures the lexical richness of a prompt independently from its length.
- *Lexical Difficulty.* Beyond overall complexity, the presence of uncommon or advanced vocabulary can affect how reliably a model interprets a prompt. Recent work has shown that prompt vocabulary specificity has a measurable, non-monotonic impact on LLM performance across domains [77], and tokenization studies suggest that rare words are more likely to be fragmented into multiple sub-word units, which has been linked to degraded model performance [78]. We operationalized this dimension using the number of *Difficult Words*, defined as the count of words in the prompt that have two or more syllables and do not appear in the Dale–Chall list of common words [79].
- *Prompt Length.* Longer prompts provide richer context, but may also introduce noise. Practitioners often perceive well-structured, longer prompts as more effective [60]. To operationalize this metric, by counting the *number of sentences* of the natural language request made in the prompt.

Structural variables. This category was designed to capture explicit cues and contextual features embedded in the prompt. Prior work [64], [80]–[83] has shown that structural elements can shape how models interpret and respond to instructions. As such, we identified three specific metrics capturing these structural aspects.

- *Inclusion of Code Snippets.* Providing code examples anchors the model's reasoning to a specific syntax and style, reducing ambiguity and guiding the model toward concrete, context-aware interpretations. Studies show that including code snippets enhances model reasoning and output quality [64]. We operationalized this dimension with a *boolean value* that indicates whether the *prompt contains code examples or snippets*.
- *Use of Prompt Patterns.* Structural patterns such as zero-shot, few-shot, and chain-of-thought explicitly guide LLM reasoning. These approaches have proven effective across diverse software engineering tasks [80], [82], [83]. To operationalize the presence of prompt patterns, we relied on Hou et al. [84], who identify eight categories. We excluded those incompatible with our single-turn setting (Auto-CoT [85], SCoT [86], MoT [87], and CoC [88]) or designed for prompt generation (APE [89]). The metric was thus defined over the three remaining patterns, Zero-Shot, Few-Shot, and Chain-of-Thought, and coded as a *binary variable*, where 1 indicates that the prompt follows one of these patterns and 0 otherwise.
- *Type of Task Performed.* Different software engineering tasks vary in complexity and context. Accounting for task type is therefore essential, since prompt effectiveness may depend on the underlying activity. To operationalize this dimension, each prompt was classified according to the type of maintenance and evolution task it represented, following the standard categories

3. We deliberately exclude intrinsic properties of the generated code, such as cyclomatic complexity or length, from our notion of output quality. Under our intent-based framing (Section 1), a simple prompt may legitimately require complex code and vice versa; the complexity of the output, on its own, does not indicate whether the developer's intent was met. Code complexity remains a relevant property for concerns such as maintainability, which we leave to future work.

defined in the ISO/IEC 14764:2022 standard [81], and coded as a *categorical variable* where the categories are the ones defined in the standard.

Together, these eight variables capture complementary dimensions of prompt formulation. Readability-related factors determine how clearly intent is communicated, while structural features shape reasoning and contextual adaptation. This dual perspective aligns with recent taxonomies of prompt design [65] and frameworks for prompt-aware evaluation [61], providing a robust foundation for analyzing how prompt design impacts output quality in code generation. Section 4.2 describes how readability and structural variables have been operationalized in our study.

3.3 Research Questions and Hypotheses

Following the definition of the variables, we formulate three research questions, each corresponding to one dimension of output quality: conceptual consistency, usefulness, and correctness. The research questions frame the overall objectives of the study and define the phenomena under investigation. In addition, we introduce corresponding hypotheses, which provide the statistical operationalization of the research questions for the regression analyses [90]. More specifically, the hypotheses formalize whether a given prompt-oriented metric is significantly associated with a given output-quality dimension. Since the current body of knowledge on prompt quality in software engineering is still limited and does not provide sufficiently consistent theoretical or empirical evidence to support directional expectations for all variables, we formulate non-directional hypotheses, i.e., hypotheses of the form $H_0 : \beta_j = 0$ versus $H_A : \beta_j \neq 0$, where β_j represents the coefficient associated with a given prompt-oriented metric in the regression model. This choice allows us to investigate whether significant associations exist without imposing a priori assumptions on the direction of the effects.

> **RQ$_1$: On the Relationship Between Prompt-Oriented Metrics and Conceptual Consistency.** *Which prompt-oriented metrics are associated with the LLM-generated code that closely matches the code a developer ultimately uses?*

> **RQ$_2$: On the Relationship Between Prompt-Oriented Metrics and Code Usefulness.** *Which prompt-oriented metrics are associated with the LLM-generated code that developers consider useful?*

> **RQ$_3$: On the Relationship Between Prompt-Oriented Metrics and Code Correctness.** *Which prompt-oriented metrics are associated with the LLM-generated code that is correct and functions as intended?*

For each research question described above, we developed a pair of hypotheses, i.e., null (H_0) and alternative (H_A), to guide the empirical investigation. Let Y_1, Y_2, and Y_3 denote the dependent variables: Y_1 = Alignment between committed code and LLM-generated code (**RQ$_1$**); Y_2 = Usefulness of LLM-generated code (**RQ$_2$**); and Y_3 = Correctness of LLM-generated code (**RQ$_3$**). Let X_1–X_8 denote the independent variables: X_1 = Prompt Readability; X_2 = Prompt Linguistic Complexity; X_3 = Difficult Words; X_4 = Vocabulary Richness; X_5 = Prompt Length; X_6 = Inclusion of Code Snippets; X_7 = Use of Prompt Patterns; and X_8 = Type of Task Performed. Our hypotheses are defined as follows:

$$RQ_1 \begin{cases} \mathbf{H1_{j,0}}: & \text{corr}(Y_1, X_j) = 0, \quad j = 1, \ldots, 8 \\ \mathbf{H1_{j,A}}: & \text{corr}(Y_1, X_j) \neq 0, \quad j = 1, \ldots, 8 \end{cases}$$

$$RQ_2 \begin{cases} \mathbf{H2_{j,0}}: & \text{corr}(Y_2, X_j) = 0, \quad j = 1, \ldots, 8 \\ \mathbf{H2_{j,A}}: & \text{corr}(Y_2, X_j) \neq 0, \quad j = 1, \ldots, 8 \end{cases}$$

$$RQ_3 \begin{cases} \mathbf{H3_{j,0}}: & \text{corr}(Y_3, X_j) = 0, \quad j = 1, \ldots, 8 \\ \mathbf{H3_{j,A}}: & \text{corr}(Y_3, X_j) \neq 0, \quad j = 1, \ldots, 8 \end{cases}$$

4 Research Method

Building on the variables and hypotheses defined in Section 3, this section describes the empirical procedures adopted to test them. We first detail the construction of the dataset (Section 4.1), then specify how each independent and dependent variable was measured (Section 4.2), and finally outline the statistical models used to analyze the data (Section 4.3). To address our **RQ**s and test the hypotheses, we conducted an empirical study on a dataset of real-world, developer-oriented prompts we have built. We extracted only single-turn conversations written in English that generated code in order to facilitate the identification of the factors influencing the output. For **RQ$_1$**, the developer-adapted code was retrieved from the GitHub commit link associated with each prompt. Both the generated code and the corresponding developer-adapted were embedded to compute their semantic similarity [33]. For **RQ$_2$** and **RQ$_3$**, the generated code was evaluated using the ICE-Score [34] to obtain usefulness and correctness values, respectively. For each research question, we performed a statistical analysis to examine the relationships between the prompt-oriented metrics and each of the three output quality dimensions.

4.1 Data Collection and Preparation

We chose to analyze real-world conversations between developers and ChatGPT to ensure that the prompts reflected real usage scenarios. In line with this goal, we did not apply any preprocessing to the code snippets when computing conceptual consistency, so as to preserve the authenticity of the interactions as they occur in practice. This allows us to capture how developers naturally phrase requests and provide context, unlike manually crafted datasets that risk overly simplified wording and a lack of contextual information. Our focus on ChatGPT was motivated by two reasons: first, to align our methodology with previous research efforts, which have also relied on ChatGPT-based datasets [91]–[93]; and second, because only a limited number of LLM platforms make shared conversations publicly available, with ChatGPT being the most prominent source. In practice, conversations involving LLMs other

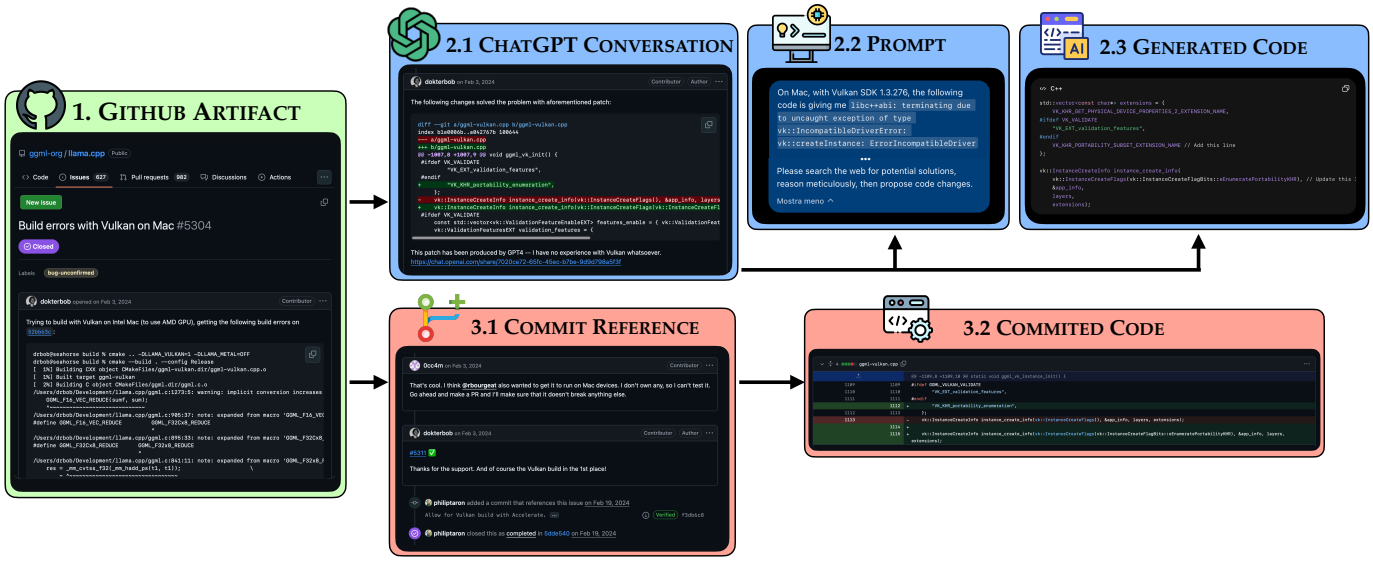

Fig. 2: Overview of the extraction process

Issue

ID: `IS_1361_1`

CONVERSATION TEXT

On Mac, with Vulkan SDK 1.3.276, the following code is giving me `libc++abi: terminating due to uncaught exception of type vk::IncompatibleDriverError: vk::createInstance: ErrorIncompatibleDriver`.

Please search the web for potential solutions, reason meticulously, then propose code changes.

Commit

ID: `CO_165_1`

CONVERSATION TEXT

Look at the readme of my project.

Create a Python script which pretends to be `Continue.dev`, sends a POST request just like in my Research section, but this time send to destination `127.0.0.1:11434` (my load balancer). Slightly change the request to use `stream: false` in the JSON.

Pull Request

ID: `PR_46_1`

CONVERSATION TEXT

Your job is to provide documentation for the code below. Please follow best code documentation practices. Also, add comments where necessary. You may delete/modify the line containing `// needs better documentation`.

Fig. 3: Representative examples of developer–LLM conversations drawn from the three sources analyzed in our dataset: GitHub **Issues**, **Commits**, and **Pull Requests**.

than CHATGPT accounted for less than 2% of the collected data; for example, attempts to include conversations from GEMINI yielded very few usable instances, many of which were no longer accessible. As a consequence, including such data would have resulted in a severely unbalanced variable and would not have supported meaningful comparative analysis. While this limits our ability to empirically assess generalizability across models, existing benchmarks (e.g., HumanEval [12], BigCodeBench [69]) show that state-of-the-art LLMs exhibit highly similar code-generation capabilities, suggesting that our findings are likely to extend beyond CHATGPT.

In exploring available resources, we considered existing datasets of developer–LLM interactions, but found that they were not directly suitable for our study. For instance, DevGPT [91] contains prompts and LLM responses from various developer interactions. It also includes the generated code and the link to the commit where the conversation was shared, thus containing all the information required for our analysis. However, its last update in October 2023 raised concerns about its ability to capture more recent trends in LLM usage. Prompt-Set [94] contains prompts extracted from source code files where developers performed tasks with LLM assistance, providing insights into how LLMs are integrated into real workflows. However, it does not include the actual output produced by the LLM. Without access to the generated output, none of the output quality metrics defined can be computed, making it impossible to proceed with the intended analysis.

For these reasons, we created a new dataset that satisfies our requirements in terms of data composition and is sufficiently recent to capture current patterns of developer–LLM interactions. To construct it, we replicated the data collection methodology used by Xiao et al. [91] to develop Dev-GPT, detailed in Figure 2. Specifically, we searched GitHub issues, commits, and pull requests for links to shared ChatGPT conversations. Whenever such a link was found, we recorded both the GitHub source URL (e.g., the commit containing

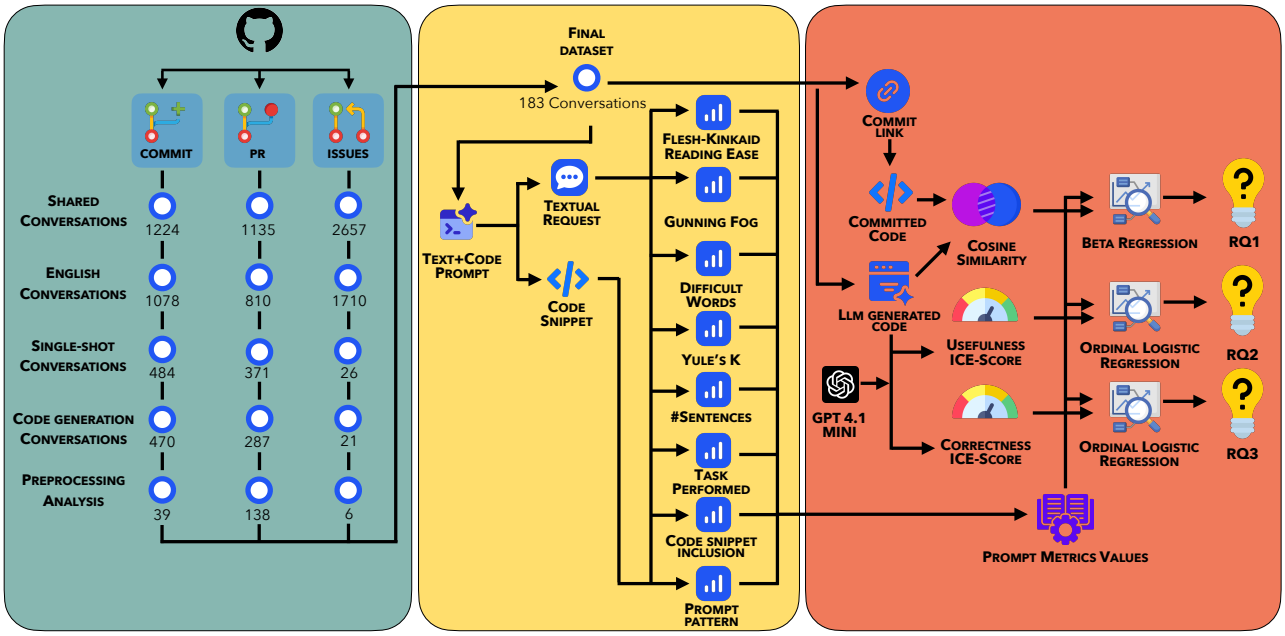

Fig. 4: Overview of the Research Method.

the conversation) and the conversation URL itself. We then extracted each user–LLM exchange from the conversation into separate files, organized by source. In this way, we could rely on a time-updated set of conversations while maintaining methodological consistency. In doing so, we also addressed a potential content bias identified by Della Porta et al. [95] in the way Dev-GPT was originally built: the dataset included general-purpose conversations from non-GitHub sources (e.g., blogs, forums, or Q&A websites), which diluted the focus on developer-oriented interactions. To avoid this issue, we restricted our replication to GitHub issues, commits, and pull requests, ensuring that the collected conversations were authored and shared by developers in a project-related context. GitHub was chosen as it provides publicly available, developer-authored interactions in a rich project context, ensuring that the collected prompts are both authentic and relevant to software development. We performed the extraction of conversation links on the 18th of March 2025, and from these conversations, we extracted the textual content exchanged between developers and the LLM, which was then processed to form the dataset used in our analysis.

Figure 3 presents some example of the conversations gathered in the dataset, meanwhile Figure 4 summarizes the results of the data collection and filtering process. As shown, we gathered a total of 5,016 raw conversations from GitHub (1,224 from commits, 1,135 from pull requests, and 2,657 from issues). Afterward, we applied a series of filtering criteria to restrict the analysis to relevant types of prompts. First, we restricted the dataset to *English-language prompts* to ensure consistency in textual metrics and to simplify manual inspection, as all authors are proficient in English: this filter reduced the dataset to 3,598 conversations. Then, we selected only *single-turn conversations*, i.e., a single interaction between a developer and the LLM, narrowing the dataset to 881 items. This restriction excludes multi-turn conversations, where developers may request successive adaptations, and thus may affect the generalizability of the study. Nonetheless, this choice was essential to *isolate* the effects of the prompt characteristics under study and thereby ensure internal validity. Including multi-turn interactions could have introduced bias by conflating prompt quality with conversational dynamics, since improvements in later turns may depend as much on the developer's iterative refinements as on the original prompt, making it difficult to attribute outcomes to prompt characteristics alone.

Subsequently, we filtered out conversations that did not produce code, retaining only those in which *the LLM effectively generated code*, ensuring the presence of code needed to compute our output quality metrics; this left 778 items. Finally, an initial exploratory analysis revealed that a single contributor accounted for more than 80% of the conversations. In this setting, observations are not independent across the contributor dimension, and retaining all instances from highly represented contributors would have potentially skewed aggregate statistics and limited the representativeness of the results by disproportionately reflecting the prompting style and interaction patterns of a single developer. To mitigate this issue, we applied a preprocessing step aimed at limiting the disproportionate influence of heavily represented contributors. Specifically, we computed the mean number of conversations per contributor and constrained over-represented contributors to this threshold (10 conversations). This ensured a more balanced distribution across contributors while reducing redundancy and potential bias in the subsequent statistical analysis. While alternative approaches, such as mixed-effects models with contributor as a random effect, could account for contributor-level variability, the high within-contributor similarity observed in the over-represented cases motivated a data-level intervention to directly control for this source of bias.

TABLE 1: Output quality metrics (dependent variables) used in the study.

Metric	Scale	Description
Cosine Similarity	Continuous, range $[0, 1]$	Measures the cosine of the angle between two embedding vectors; widely used to quantify semantic similarity between texts or code. In our case, the comparison is computed between the generated code and the committed code.
ICE-Score (Usefulness)	Ordinal, range $[0, 4]$	An *LLM-as-a-judge* metric in which a language model evaluates the perceived usefulness of the generated code for the given task, without requiring a reference implementation or test suite.
ICE-Score (Correctness)	Ordinal, range $[0, 4]$	An *LLM-as-a-judge* metric in which a language model assesses whether the generated code fulfills the task requirements and behaves as intended; does not involve program execution or static analysis.

After this adjustment, the dataset stabilized at a final set of 183 conversations (39 from commits, 138 from pull requests, and 6 from issues), extracted from 94 distinct GitHub repositories. The complete list of repositories is available in the replication package [35]. We verified the adequacy of this sample size through a power analysis and an events-per-variable assessment (see Section 4.3), both of which indicate that the dataset is sufficient for the planned statistical analyses.

4.2 Measurement Process

Once the dataset of conversations was finalized, we moved to the operationalization of the independent and dependent variables required for the analysis. In this section, we describe the procedures adopted to compute each metric.

4.2.1 Dependent Variable Measurement

To test our hypotheses, we focused on three dimensions of output quality: conceptual consistency, usefulness, and correctness. The measures employed are summarized in Table 1. To compute cosine similarity, we first processed the code linked to each GitHub commit. For every commit associated with a developer–LLM conversation, we extracted only the added lines of code and stored them in a temporary file for subsequent embedding computation. From the same conversations, we also collected the code generated by the LLM in response to the developer's prompt. Both the committed code and the LLM-generated code were embedded using `Qwen3 8B`, which at the time of consulting (20th July 2025) was ranked as the top-performing open-source model on the MTEB leaderboard.[4]. We then proceeded to measure the usefulness and correctness of the generated code using the ICE-Score metric [34]. Although the original formulation of ICE-Score employed `GPT-4`, in our study we opted for `GPT-4.1-mini`.[5] This decision was made to mitigate known issues in LLM-as-a-judge settings, in particular bias arising from model identity, whereby a model acting as a judge may systematically assign higher scores to its own outputs (*self-bias*) or to those generated by models from the same family (*family-bias*), even when controlling for their underlying quality. This effect can confound evaluation results by inflating scores due to model identity rather than actual performance differences [96]. Importantly, the usage of `GPT-4.1-mini` also ensured methodological consistency across our pipeline, as the same model had already been employed in earlier steps of our study (e.g., prompt preprocessing and code/text separation). In line with the guidelines for SE studies using LLMs by Baltes et al. [97], we report that the LLM temperature was set to 0, and for prompting we used the original prompt setup reported by Zhuo et al. [34].

4.2.2 Independent Variables Measurement

The measures employed to operationalize the independent variables are summarized in Table 2. To compute the independent variables, we first separated the natural language task description in each prompt from any embedded code snippets. This step ensures that textual prompt metrics are computed exclusively on the natural language content, without being influenced by code snippets. The separation was performed using an LLM-based annotation with `GPT 4.1-mini` , which classified each line as either code or natural language. Code snippets were then replaced with placeholders identified by unique IDs, allowing us to preserve their position within the prompt while excluding them from the computation of readability and structural metrics. Since many prompts exceeded the model's context window, we divided each prompt into batches of 20 lines and asked the model to classify each line of the batch as either code or natural language. To facilitate automation, we used the structured output[6] option provided in the API to receive responses in a JSON format, which could be easily parsed. When the model failed to produce a valid structured output, the request was repeated until a correct result was obtained. All annotations were subsequently verified by two authors during a dedicated session to ensure quality. The script and prompts used for this step are available in the replication package [35]. We opted for this semi-automated approach because separating code from natural language is a relatively simple task that can be reliably automated, making a fully manual procedure unnecessarily time-consuming. This separation was essential to ensure that readability metrics and lexical measures, such as the number of sentences, were applied exclusively to the textual part of the request, avoiding distortions introduced by code tokens.

After completing this step, we measured the three readability variables on the natural language text of each prompt. The Flesch Reading Ease, the Gunning Fog Index, the Difficult Words, and Lexical Richness were measured using the `textstat`[7] Python library. The library implements the original formulations of these readability metrics

4. https://huggingface.co/spaces/mteb/leaderboard
5. https://platform.openai.com/docs/models/gpt-4.1-mini
6. https://platform.openai.com/docs/guides/structured-outputs
7. https://textstat.org/

TABLE 2: Prompt-oriented metrics (independent variables) used in the study.

Metric	Scale	Formula / Definition
Flesch Reading Ease	Continuous, $[0, 100]$	$\mathrm{RE} = 206.835 - 1.015 \cdot (\text{words}/\text{sentences}) - 84.6 \cdot (\text{syllables}/\text{words})$
Gunning Fog Index	Continuous, $[0, +\infty)$	$\mathrm{FI} = 0.4 \cdot \left((\frac{\text{words}}{\text{sentences}}) + 100 \cdot (\frac{\text{complex words}}{\text{words}}) \right)$ words = total words, complex words = not in Dale–Chall list [79]
Difficult Words	Discrete, $\{0, 1, 2, \dots\}$	$\mathrm{DW} = \sum_{w \in \text{Text}} 1[\,\sigma(w) \geq 2 \wedge w \notin \mathcal{D}\,]$ $\sigma(w)$ = syllable count of w, $\mathcal{D}$ = Dale–Chall list [79]
Yule's K	Continuous, $[0, +\infty)$	$\mathrm{K} = 10^4 \cdot \frac{\sum_{i=1}^{N} i^2 V_i - N}{N^2}$ N = total tokens, V_i = types occurring i times
Number of Sentences	Discrete, $\{0, 1, 2, \dots\}$	$S = \lvert\{\, s \in \text{Text} \mid s \text{ is a sentence} \,\}\rvert$
Task Performed	Nominal	Task $\in$ {Additive, Adaptive, Corrective, Perfective, Preventive}
Code Snippet Inclusion	Boolean, $\{0, 1\}$	$\mathrm{CodeIncl} = \begin{cases} 1 & \text{if a code snippet is present,} \\ 0 & \text{otherwise.} \end{cases}$
Prompt Pattern	Nominal	Pattern $\in$ {Zero-Shot, Few-Shot, Chain-of-Thought}

as defined in the literature. In particular, the constant coefficients used in the formulas correspond to the standard values originally derived by the metric authors through empirical calibration studies on English texts. For the Flesch Reading Ease, the constants were calibrated by relating sentence length and syllable count to human reading-difficulty judgments [74], while for the Gunning Fog Index, the scaling coefficients were introduced to map sentence complexity and the proportion of complex words to U.S. grade-level equivalents [75]. These constants were not modified or recalibrated in our study. Taking advantage of the earlier separation between text and code, we also computed two structural features of the prompts: the number of sentences in the textual part and the presence of code snippets. The *number of sentences* was obtained using the NLTK sentence tokenizer,[8] trained on English text, to approximate the level of detail and contextual information provided by the developer. The code snippet metric, on the other hand, captured whether a prompt included one or more code snippets.

To identify the *task performed* in the prompt, following the ISO/IEC 14764:2022 standard [81], we classified the tasks in 5 main types: Additive, Adaptive, Corrective, Perfective, and Preventive.[9] We decided not to use automatic classification approaches for this task. While automated methods could, in principle, speed up the process, they risk misclassifying subtle cases that require domain knowledge and contextual understanding. A manual procedure was therefore more appropriate to ensure accurate labeling and to maintain confidence in the validity of subsequent analyses. The classification was carried out independently by the seven authors of the paper, with the procedure ensuring that at least two different reviewers reviewed each prompt. In cases where the two reviewers disagreed, an external reviewer was involved to provide clarification and resolve the conflict. If the reviewers could not identify a maintenance and evolution task, the prompt was excluded from the analysis. All reviewers engaged in this step had a strong background in software engineering, which helped guarantee the reliability of the classifications. Finally, for each prompt, we identified the presence of *prompt patterns*. Each prompt of the dataset was independently reviewed by six authors, with at least two reviewers per prompt (details in the replication package [35]). Disagreements were resolved by the first author, who was selected due to their deeper expertise in the area. As with the task classification, we opted for a manual approach rather than automation, since prompt patterns are often implicit and require contextual interpretation. For instance, a prompt containing code could be misclassified as Few-Shot when the code is not an example but rather the subject of the request itself.

8. https://www.nltk.org/

9. Details on the guidelines used for the classifications are available in the replication package [35].

4.3 Data Analysis

To answer **RQ$_1$**, we employed a *beta regression model*, since the dependent variable (*cosine similarity*) is a continuous proportion bounded in the interval $[0, 1]$. Beta regression is particularly suited for this type of outcome because it allows modeling of non-normal, heteroskedastic distributions typical of proportion data, while respecting the bounded support of the variable. This type of model relies on several assumptions. First, the outcome must be restricted to the open interval $(0, 1)$, since proportions at the exact boundaries are not supported. Second, predictors should not exhibit multicollinearity, as highly correlated covariates can distort coefficient estimates and inflate standard errors. Third, observations are assumed to be independent, meaning that each prompt–output pair is treated as a distinct and unrelated data point.

To answer **RQ$_2$** and **RQ$_3$**, we applied *ordinal logistic regression (OLR)* models, since both *usefulness* and *correctness* are ordinal scores ranging from 0 to 4. OLR is an appropriate choice in this context, as it accounts for the ordered nature of the response categories without assuming equal spacing between them, unlike linear regression. OLR models also impose specific assumptions. The most important is the *proportional odds assumption* which requires that the relationship between predictors and the odds of being in higher versus lower categories of the outcome is constant across all thresholds. In addition, predictors should not be affected by multicollinearity, ensuring stable and interpretable coefficients, and the independence of observations is assumed, meaning

TABLE 3: Beta Regression Model (Note. *** $p < .001$, ** $p < .01$, * $p < .05$).

Predictor	Estimate	Std. Error	z-value	p-value	Sign.
Flesch Reading Ease	0.001	0.005	0.254	0.799	
Gunning Fog Index	0.030	0.023	1.297	0.195	
Difficult Words	-0.021	0.009	-2.265	0.024	*
Number of Sentences	0.112	0.034	3.288	0.001	**
Yule's K	-0.001	0.001	-0.974	0.330	
Task Performed: Additive	-0.672	0.220	-3.049	0.002	**
Task Performed: Corrective	-0.941	0.176	-5.348	<0.001	***
Task Performed: Preventive	-0.750	0.264	-2.840	0.005	**

that the responses of one participant or prompt–output pair should not influence those of another.[10]

Since our analyses are performed at the conversation level, each developer–LLM conversation constitutes one independent observation in the dataset. We therefore treat the number of conversations ($N = 183$) as the effective sample size for all regression models, rather than counting individual prompts or turns within a conversation, which would violate the independence assumption. To justify that $N = 183$ is adequate, we conducted two complementary checks. First, for the regression model with the seven independent variables discussed in Section 4.2.2, we aimed to detect a medium effect size ($f^2 = 0.15$) with power 0.80 and $\alpha = 0.05$. A G*Power analysis [100], [101] yields a minimum required sample size of $N = 103$ observations. Second, for the ordinal logistic regression models, we cross-validated adequacy using the widely adopted events-per-variable heuristic (at least 10 observations per predictor per response level). With seven predictors and five ordered response categories (0–4), this implies a conservative requirement of $N \geq 70$ conversations. Our dataset of 183 conversations exceeds both thresholds, indicating that the sample size is appropriate for the model complexity and sufficient to detect medium effects.

5 Analysis of the Results

In this section, we report the results of our analysis, organized by the three research questions defined in Section 3.3. The statistical analysis was performed using the R programming language, and all the scripts used for the analysis can be found in the replication package [35].

Before conducting the analysis, we examined the distribution of all the metrics used in the study. In this stage, we decided to remove the *Prompt Pattern* metric, since the manual classification revealed that 90% of the prompts (164 out of 183) were Zero-shot. This strong imbalance, also observed in previous work [95], would have heavily affected the statistical models and biased the results. For this reason, we excluded the metric from further analysis and did not test the associated hypotheses ($H1_{7,0}$, $H2_{7,0}$, and $H3_{7,0}$).

10. For the categorical *task type* predictor, we adopted *Perfective* as the reference category, as it is the most frequent category in our dataset. Selecting the modal category as the baseline is a standard practice in regression modeling, as it yields more stable contrast estimates against a larger reference subsample [98]. We note, however, that this choice does not affect the overall model fit, the predicted values, or the global significance of the factor: a different baseline would simply re-express the coefficients of the remaining categories without altering the substantive conclusions [98], [99].

Another point to mention is that the *adaptive* category of the *task performed* type is not present in the final version of the dataset. The label was occasionally proposed during initial labeling but never retained after adjudication, and is therefore absent from the analysis.

5.1 RQ_1: On the Relationship Between Prompt-Oriented Metrics and Conceptual Consistency

For **RQ_1**, we investigated the relationship between prompt-oriented metrics and the conceptual consistency between LLM-generated code and the developer-adapted code that was commited on the repository. To this end, we fitted a beta regression model with cosine similarity as the dependent variable and the readability and structural predictors as independent variables.

Before fitting the model, we rescaled the cosine similarity values to avoid boundary issues, since some raw values were exactly equal to 1. Following Smithson and Verkuilen [102], we applied a standard transformation that maps values into the open interval $(0, 1)$, to ensure compatibility with beta regression. Diagnostic tests based on simulated residuals (Kolmogorov–Smirnov uniformity test, dispersion test, and outlier test) indicated no violations of distributional assumptions (all $p > 0.05$). Variance inflation factors (VIF) were below the common threshold of 5.

The results are reported in Table 3. Among the readability metrics, neither Flesch Reading Ease nor Gunning Fog Index showed a significant association with cosine similarity ($p = 0.799$ and $p = 0.195$, respectively). Yule's K was also non-significant ($p = 0.330$). In contrast, two structural features displayed significant effects: the number of difficult words in the prompt was negatively associated with similarity ($\beta = -0.021$, $p = 0.024$), while the number of sentences was positively associated ($\beta = 0.112$, $p = 0.001$).

Regarding task type, we used perfective tasks as the baseline. Compared to this baseline, additive ($\beta = -0.672$, $p = 0.002$), corrective ($\beta = -0.941$, $p < 0.001$), and preventive ($\beta = -0.750$, $p = 0.005$) tasks were all significantly associated with lower cosine similarity.

Overall, the model explained a moderate portion of the variance (pseudo $R^2 = 0.159$) and met the distributional assumptions based on residual diagnostics. In light of these findings, we reject the null hypotheses associated with difficult words $H1_{3,0}$, the number of sentences $H1_{5,0}$, and task type $H1_{8,0}$, thereby supporting their corresponding alternative hypotheses.

TABLE 4: Ordered Logistic Regression (OLR) Model for Usefulness (Note. *** $p < .001$, ** $p < .01$, * $p < .05$).

Predictor	Estimate	Std. Error	t-value	odds ratio	p-value	Sign.
Flesch Reading Ease	0.033	0.011	2.990	1.033	0.003	**
Gunning Fog Index	0.211	0.058	3.651	1.235	<0.001	***
Difficult Words	-0.006	0.021	-0.261	0.995	0.794	
Number of Sentences	-0.023	0.077	-0.301	0.977	0.763	
Yule's K	0.001	0.002	0.375	1.001	0.708	
Task Performed: Additive	0.629	0.537	1.170	1.876	0.242	
Task Performed: Corrective	-0.380	0.427	-0.891	0.684	0.373	
Task Performed: Preventive	1.934	1.091	1.773	6.920	0.076	

Answer to RQ_1. *The analysis showed that structural aspects of prompts affect conceptual consistency. A higher number of sentences in a prompt was positively correlated with greater similarity to developer code, while increased lexical diversity exhibited a negative correlation. In addition, task type influenced outcomes, with perfective tasks showing the strongest alignment.*

5.2 RQ_2: On the Relationship Between Prompt-Oriented Metrics and Code Usefulness

For $\mathbf{RQ_2}$, we investigated the relationship between prompt-oriented metrics and the perceived usefulness of LLM-generated code, as measured by ICE-Score. We fitted an *Ordinal Logistic Regression* model (OLR) with usefulness (0–4) as the dependent variable, and readability and structural predictors as independent variables.

All assumptions for ordered logistic regression were checked. Variance inflation factors (VIF) were within acceptable thresholds (all < 5). The proportional odds assumption was tested using the nominal test, and no significant violations were observed, supporting the validity of the OLR specification. The results are presented in Table 4. Two readability metrics emerged as significant predictors: higher Flesch Reading Ease scores were associated with an increased probability of usefulness being rated higher (odds ratio = 1.033), while higher Gunning Fog Index values showed an even stronger positive effect (odds ratio = 1.235). In contrast, structural metrics such as difficult words, number of sentences, and Yule's K were not significant predictors. Regarding task type, additive and corrective tasks showed positive coefficients compared to the perfective baseline, with corrective tasks approaching significance ($p = 0.084$), while preventive tasks were positively associated but non-significant. Overall, the model showed modest explanatory power (pseudo $R^2 = 0.107$). In light of these findings we reject the null hypotheses associated with Flesch Reading Ease $H2_{1,0}$ and Gunning Fog Index $H2_{2,0}$, thereby supporting their corresponding alternative hypotheses.

Answer to RQ_2. *Prompt readability influenced usefulness: clearer and more complex prompts were associated with higher usefulness ratings of generated code. Structural features and most task types showed no significant association. Preventive tasks suggested a positive trend but did not reach statistical significance.*

5.3 RQ_3: On the Relationship Between Prompt-Oriented Metrics and Code Correctness

For $\mathbf{RQ_3}$, we investigated the relationship between prompt-oriented metrics and the correctness of LLM-generated code. Correctness was modeled using ordered logistic regression with readability and structural predictors as independent variables. Initial tests revealed violations of the proportional odds assumption for the number of difficult words and sentences. To address this, these predictors were removed and the model re-estimated. The final model is reported in Table 5. Both readability metrics—Flesch Reading Ease and Gunning Fog Index—were significantly and positively associated with correctness, indicating that prompts that are clearer and structurally more complex tended to yield more correct code. Yule's K did not show a significant effect. Task type was also not significant, although corrective tasks showed a marginal negative trend relative to the perfective baseline.

Overall, the model showed an explanatory power of pseudo $R^2 = 0.148$. In light of these findings, we reject the null hypotheses associated with Flesch Reading Ease $H3_{1,0}$ and Gunning Fog Index $H3_{2,0}$, thereby supporting their corresponding alternative hypotheses.

Answer to RQ_3. *Readability metrics were the strongest predictors of correctness. Higher Flesch Reading Ease and Gunning Fog scores correlated with a higher likelihood of correct code. Other predictors showed no significant effects.*

6 Discussion, Implications, and Limitations

This section discusses the implications of our results.

6.1 Interpretation of the Results

Our results demonstrate that prompt-oriented metrics influence different dimensions of output quality.

For *conceptual consistency*, the relevant predictors were structural rather than readability-based (Table 3). The number of difficult words negatively influenced similarity, while the number of sentences had a positive effect. This suggests that prompts articulated across multiple sentences improve alignment with the code developers commit, whereas excessive lexical variety reduces it. Importantly, all task types (i.e., additive, corrective, and preventive) had significant negative effects relative to perfective tasks. This shows that prompts asking the model to add new functionality, correct faults, or prevent potential issues lead to less similar outputs than prompts framed around perfective modifications.

TABLE 5: Ordered Logistic Regression Model for Correctness (Note. *** $p < .001$, ** $p < .01$, * $p < .05$). Two measures, i.e., the number of difficult words and sentences, were excluded due to assumption violations.

Predictor	Estimate	Std. Error	t-value	odds ratio	p-value	Sign.
Flesch Reading Ease	0.035	0.012	2.896	1.036	0.004	**
Gunning Fog Index	0.206	0.060	3.416	1.230	0.001	**
Yule's K	-0.002	0.002	-1.056	0.998	0.291	
Task Performed: Additive	-0.053	0.495	-0.107	0.949	0.915	
Task Performed: Corrective	-0.681	0.394	-1.730	0.506	0.084	
Task Performed: Preventive	0.328	0.699	0.470	1.389	0.638	

For *usefulness*, readability proved decisive again (Table 4). Flesch Reading Ease and Gunning Fog Index had significant positive effects, confirming that prompts which combine clear and readable phrasing with more articulated technical content are more likely to result in outputs considered useful. Task type did not have statistically significant effects, although preventive prompts showed a tendency to increase usefulness compared to perfective ones.

For *correctness*, both Flesch Reading Ease and Gunning Fog Index showed significant positive effects (Table 5). Prompts that combined high readability with more articulated and lexically sophisticated content were more likely to lead to correct solutions, suggesting that prompts containing detailed technical information, when expressed in a clear and well-structured manner, help the model generate functionally accurate code. As for task type, corrective prompts showed a negative trend compared to perfective ones, although this effect was only marginally significant. This indicates that when the task is framed as correcting existing code, the model tends to produce fewer correct outputs. A possible explanation is that corrective tasks, such as bug fixing, are inherently more complex than perfective tasks, such as extending or generating code, which makes them harder for LLMs to handle. While developers cannot change the nature of their tasks, this finding highlights that, currently, achieving correctness is more difficult in bug-fixing scenarios. Vocabulary richness, by contrast, had no significant effect, indicating that lexical variety does not enhance the correctness of software engineering prompts.

These findings highlight that **prompt quality is inherently multi-dimensional**. Readability metrics emerge as key drivers of correctness and usefulness, whereas similarity is more strongly influenced by structural features and task framing. Furthermore, measuring task-type effects against perfective prompts highlights how the way a problem is posed, whether as correction, addition, or prevention, shapes alignment with developer expectations.

6.2 Implications of the Study

Overall, prompt quality emerges as a multi-dimensional and context-sensitive construct, shaped by the interplay between prompt formulation, task characteristics, and the interaction and validation strategies adopted by developers. Future research should therefore move toward task-aware and quality-aware frameworks that jointly consider correctness, usefulness, and conceptual consistency, treating readability, structural organization, and task-specific interaction practices as complementary determinants of LLM performance in software engineering.

6.2.1 Readability as a Targeted Lever for Output Quality

Readability metrics act selectively on output quality. Flesch Reading Ease and Gunning Fog Index strongly influenced correctness and usefulness, but not conceptual consistency, meaning that readability helps the model generate accurate and practically usable code without necessarily bringing the output closer to what developers ultimately commit. For such a reason, developers should not treat readability as a guarantee of alignment with their final implementation, but rather as a means to reduce ambiguity and increase the likelihood that the generated code is correct, useful, and easy to adapt before integration.

For practitioners, since easier-to-read prompts were associated with better outcomes, prompts should be clear and explicitly articulated rather than syntactically dense. Such a guideline translates into four practices: (i) stating the task explicitly in the first sentence; (ii) using short sentences, each expressing a single requirement; (iii) separating goal, context, constraints, and expected behavior into distinct sentences; and (iv) avoiding deeply nested constructions that embed requirements within subordinate clauses. As an illustrative example, consider the request *"The function that is shown below appears to behave in an unexpected manner in the specific situation where the list it receives as a parameter does not contain any elements at all, and the appropriate resolution would presumably be to return an empty list in that case. The behavior for lists that do contain elements has to remain completely unchanged, and the efficiency of the implementation on considerably larger inputs should also be preserved as it is."* (Flesch Reading Ease: 21.6; Gunning Fog Index: 22.7; Correctness: 3; Usefulness: 0). Applying the four practices, the same requirements can be expressed as *"Fix the function below. It fails when the input list is empty. In that case it should return an empty list. Do not change how it handles lists that are not empty. Keep the code fast for large inputs."* (Flesch Reading Ease: 94.8; Gunning Fog Index: 3.1; Correctness: 3; Usefulness: 3). The two prompts differ only in surface form, yet only the readable one yielded code that a developer could adopt without further intervention. Readability therefore acts on usefulness even when correctness is within reach of both formulations.

For researchers, these results pave the way for quality-aware prompt engineering support, comprising (i) prompt quality analyzers that detect readability and structural issues (e.g., long sentences or implicit requirements); (ii) prompt refactoring techniques that transform poorly structured prompts into clearer, multi-sentence formulations; and (iii) predictive models to estimate how changes in prompt design may affect correctness or usefulness.

> **Implication 1.** *Readability should be treated as a design mechanism to improve correctness and usefulness, rather than as a proxy for prompt quality. Developers should structure prompts explicitly by stating the task up front, using short, single-purpose sentences, and separating context, constraints, and expected behavior into distinct statements.*

6.2.2 Prompts as Structured Artifacts

Conceptual consistency was mainly driven by sentence count and task framing, while readability and lexical richness had little influence, indicating that prompt quality is not only a matter of clear sentences but also of how requirements are decomposed and organized. Indeed, prompts expressed in multiple sentences, each introducing a specific requirement, constraint, or piece of context, tend to produce code more closely aligned with the final implementation, whereas high lexical variability (e.g., frequent synonyms or stylistic variation) may introduce ambiguity and reduce alignment. Hence, developers should decompose complex requests into sequences of simple, well-scoped statements and use consistent terminology, rather than compressing multiple requirements into a single, dense formulation.

For practitioners, these findings suggest that task framing should be explicitly tailored to the activity being performed, rather than relying on generic prompt formulations. Since perfective requests led to higher alignment than additive, corrective, or preventive ones, each task type appears to require its own level of explicitness. Starting from a minimally specified prompt such as *"Update this function"*, developers should make explicit (i) what is wrong and what the correct behavior should be for corrective tasks (e.g., *"Fix this function. It fails when the input list is empty. The function should return an empty list in this case"*); (ii) where the new functionality should be integrated and under which constraints for additive tasks; (iii) the scenarios or issues to be avoided for preventive tasks; and (iv) the intended improvement for perfective tasks.

For researchers, these findings call for expanding prompt-quality models beyond readability and lexical properties to incorporate structural dimensions, i.e., information decomposition, requirement ordering, and task framing. Such dimensions should be (i) operationalized into measurable constructs (e.g., metrics capturing how requirements are distributed across sentences or whether failure conditions are stated); (ii) embedded into structure-aware prompt patterns that enforce progressive specification; and (iii) supported by static analyzers and refactoring techniques that detect missing specification elements or reorganize requirements. From this perspective, prompts resemble software design artifacts, whose quality depends not only on how clearly each instruction is written but also on how the instructions are structured

> **Implication 2.** *Prompt quality models should incorporate structural properties at the prompt level, such as requirement decomposition, ordering, and task framing, in addition to readability and lexical metrics. Developers should organize prompts as sequences of well-scoped requirements, while researchers should model and support how multiple instructions are structured, prioritized, and connected.*

6.2.3 The Context-Dependent Nature of Prompt Quality

Additive, corrective, and preventive prompts all reduced conceptual consistency compared to perfective prompts, and corrective prompts also tended to reduce correctness, indicating that certain software engineering tasks are systematically harder for LLMs and therefore more sensitive to how the prompt is formulated.

For practitioners, these findings suggest that prompt quality cannot be defined independently of the task being performed, and developers should adapt their interaction strategy accordingly: validating corrective outputs through testing or manual inspection before integration, verifying that preventive outputs actually enforce the intended constraints, and checking that additive outputs do not introduce unintended side effects. In all such cases, an iterative strategy that progressively refines and validates outputs may be preferable to relying on a single response, whereas perfective tasks may already yield satisfactory results with simpler interactions.

For researchers, prompt quality should be modeled as a function of the task and the associated validation requirements, for example, through task-specific benchmarks that reflect varying difficulty and risk across tasks (e.g., correctness in bug fixing vs. robustness in preventive tasks) and through the study of iterative prompting and validation workflows. More generally, these findings call for integrating prompt engineering with established software engineering practices, e.g., testing, verification, and quality assurance, and evaluating prompt quality by how effectively the generated output supports downstream activities such as validation, debugging, and integration into the codebase.

> **Implication 3.** *Prompt quality is context-dependent and should be evaluated together with the task and the associated interaction and validation strategy. Developers should adapt how they use and verify LLM outputs depending on the task (e.g., prioritizing testing and inspection for corrective and preventive tasks), while researchers should model prompt quality as a function of task-specific requirements and validation workflows.*

6.3 Threats to Validity

Our empirical study may be subject to several threats to validity, which we discuss in this section.

6.3.1 Conclusion Validity

The main threats in this category stemmed from the statistical methods applied during the analysis. We employed *Beta Regression* and *Ordinal Logistic Regression*, as they fit our research design and the characteristics of the variables. We verified all model assumptions and examined the relevant statistical measures to ensure that the analysis was conducted correctly and with rigor. Moreover, the moderate explanatory power of our models ($0.107 <$ pseudo-$R^2 < 0.159$) indicates that prompt-oriented metrics account for only part of the variability in output quality. Consequently, although several relationships were statistically significant, their practical significance should be interpreted with caution. Our findings identify prompt characteristics

associated with output quality, while future replications and controlled experiments are needed to establish the practical magnitude of these relationships.

Another potential threat comes from the non-determinism of LLMs, as repeated executions of the same prompt may yield different outputs. This variability could, in principle, affect the dependent variables and the stability of our findings. To mitigate this, we relied on a fixed dataset of already-executed conversations, ensuring that the analysis was performed on stable outcomes rather than re-generated outputs. In other words, while results may differ across generations, our study reflects the actual code that developers obtained from the LLM in practice. Future replications on alternative datasets or regenerated outputs are welcome to further assess the robustness of our findings.

Our study identifies statistical associations between prompt characteristics and output quality, but it does not establish causality. This limitation is inherent to observational studies on real-world data, where multiple confounding factors may exist. To mitigate the risk of overstating our findings, we carefully framed results in correlational terms, verified robustness across multiple model specifications, and excluded variables that violated model assumptions. In other terms, we view this work as a first step toward theory building: our findings generate hypotheses that can be tested through controlled experiments. In this sense, we envision future research pursuing such designs to move from correlations to stronger causal claims.

6.3.2 Construct Validity

Our data was mined from GITHUB open-source repositories, which may have raised concerns about quality; to mitigate this threat, we applied filtering and corrective steps to ensure data reliability. We further adopted both numerical and categorical measurements. For the former, we operationalized well-established metrics from the literature. For the latter, we relied not only on categories recognized in prior work (e.g., ISO standards) but also implemented a rigorous multi-step, multi-rater categorization process involving the entire research team. For example, each dataset entry was labeled by at least two authors independently, and disagreements were resolved by a third author. Another potential threat came from errors in data preprocessing and filtering. We addressed this by inspecting the scripts used for data collection and processing and by manually validating a sample of the dataset. Finally, we did not preprocess code (e.g., comment removal, normalization) before computing conceptual consistency between LLM-generated and developer-adapted code snippets, which may influence similarity scores. This was deliberate: our dataset consists of real developer prompts, and applying transformations would break correspondence with what was actually submitted to the model, harming ecological validity. To prevent code from contaminating textual prompt metrics, code segments were separated from the natural language portion and replaced with placeholder IDs before computing readability and structural features.

6.3.3 Internal Validity

To mitigate this risk, we followed a structured two-stage process to elicit and refine the variables used in the study (Section 3). In the first stage, candidate variables were identified through an expert workshop and consolidated using explicit inclusion/exclusion criteria. In the second stage, these variables were iteratively reviewed and refined across multiple author meetings, with systematic checks against the literature until saturation was reached. This procedure ensured that the final set of independent and dependent variables captured complementary and non-overlapping dimensions of prompt and output quality. While it remains possible that some relevant variables were overlooked, the resulting set is principled and transparent, thereby increasing confidence in the internal validity of our findings and offering a solid foundation for future replications with extended metrics.

6.3.4 External Validity

Our results may have been limited by the dataset we constructed. We adopted a thorough data collection process and applied multiple filtering and corrective steps to ensure balance and representativeness. Another limitation was that we only considered prompts written in English, which restricted the applicability of our findings to prompts formulated in other languages. Moreover, because our dataset was based solely on publicly available ChatGPT conversations from GITHUB repositories, the findings may not have fully generalized to proprietary or private development contexts. In addition, the focus on maintenance and evolution tasks meant that results might not generalize to other software engineering activities, such as design or prototyping. Likewise, the restriction to single-turn conversations excluded the more common multi-turn interactions, limiting generalizability to real-world collaborative use cases. However, this exclusion was a principled choice, motivated by the need to guarantee internal validity: by isolating a single prompt–response exchange, we avoided confounding effects introduced by iterative developer refinements across turns, which would have made it difficult to attribute outcomes to prompt characteristics alone and could have compromised the reliability of the statistical analysis. Another potential limitation came from the evaluation of usefulness and correctness through ICE-Score with `GPT-4.1-mini` as the judging model, which may have introduced model-specific biases; we mitigated this by selecting a strong and widely adopted evaluator and by ensuring methodological transparency to enable replication with alternative judges. Finally, temporal validity also had to be considered: our dataset was collected in early 2025, and both LLM technology and prompting practices evolve rapidly. We addressed this by basing our analysis on a recent snapshot of real-world interactions, while making our dataset and scripts publicly available to facilitate replications in different contexts and timeframes.

7 CONCLUSION

This study examined whether prompt-oriented metrics can explain the quality of LLM-generated code. We analyzed the conceptual consistency, correctness, and usefulness of outputs in relation to prompt readability and structure. Readability strongly predicted correctness and usefulness,

while conceptual consistency was shaped by structural factors: more sentences improved alignment, whereas greater lexical variety and non-perfective task framings reduced it. Corrective prompts also tended to lower correctness compared to perfective ones. Overall, these results indicate that prompt quality is inherently multi-dimensional.

Our results inform a future research agenda, which aims to explore multi-turn interactions, adopt causal designs to establish stronger relationships, and develop tools that provide developers with real-time feedback on prompt readability and structure. Another promising direction concerns the code snippets embedded within prompts. While our study focused on the natural language component by replacing code snippets with placeholders before computing readability metrics, recent evidence suggests that the code component also affects LLM behavior. For example, Pan et al. [103] showed that code formatting influences token consumption without substantially improving model performance. These findings suggest that comprehensive prompt quality models should jointly characterize both the natural language and code components of developer prompts.

ACKNOWLEDGMENT

Della Porta and Palomba acknowledge the support of the project *FAIR* (Code: PE0000013), funded under the NRRP MUR program by the European Union – NextGenerationEU, and the project *Qual-AI* (Code: D53D23008570006), funded under the MUR PRIN 2022 program. Pontillo acknowledges the support of the European HORIZON-KDT-JU-2023-2-RIA research project MATISSE "Model-based engineering of Digital Twins for early verification and validation of Industrial Systems" (grant 101140216-2, KDT232RIA_00017). Rahman acknowledges the support of the U.S. National Science Foundation (NSF) Award #2310179 and Award #2312321. Ferreira was supported by national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) under projects UID/50021/2025 (DOI: 10.54499/UID/50021/2025), UID/PRR/50021/2025 (DOI: UID/PRR/50021/2025), and project SafeIaC (DOI: 10.54499/2023.18089.ICDT). The authors would like to acknowledge Shonan Meeting 207 *"Anti- patterns and Defects: Synergies, Challenges, and Opportunities"* for the inspiration and bringing researchers together to work on this research (https://shonan.nii.ac.jp/docs/No.207.pdf).

REFERENCES

[1] K. G. Srivatsa, S. Mukhopadhyay, G. Katrapati, and M. Shrivastava, "A survey of using large language models for generating infrastructure as code," in *Proceedings of the 20th International Conference on Natural Language Processing (ICON)*, J. D. Pawar and S. Lalitha Devi, Eds. Goa University, Goa, India: NLP Association of India (NLPAI), Dec. 2023, pp. 523–533. [Online]. Available: https://aclanthology.org/2023.icon-1.48/

[2] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating large language models in class-level code generation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639219

[3] T. Ahmed and P. Devanbu, "Few-shot training llms for project-specific code-summarization," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3559555

[4] Y. Virk, P. Devanbu, and T. Ahmed, "Calibration of large language models on code summarization," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3729400

[5] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, and X. Peng, "Exploring the potential of chatgpt in automated code refinement: An empirical study," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623306

[6] K. H. Levin, N. van Kempen, E. D. Berger, and S. N. Freund, "Chatdbg: Augmenting debugging with large language models," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3729355

[7] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660783

[8] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623343

[9] S. Zhou, J. Wang, H. Ye, H. Zhou, C. L. Goues, and X. Luo, "Lwdiff: an llm-assisted differential testing framework for webassembly runtimes," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 153–164.

[10] A. Parziale, G. Voria, V. Pontillo, G. Catolino, A. De Lucia, and F. Palomba, "Toward systematic counterfactual fairness evaluation of large language models: The caffe framework," *arXiv preprint arXiv:2512.16816*, 2025.

[11] A. Della Porta, G. Recupito, S. Lambiase, D. Di Nucci, and F. Palomba, "Unlocking code simplicity: The role of prompt patterns in managing llm code complexity," in *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering-Companion (SANER-C)*. IEEE, 2025, pp. 140–143.

[12] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, pp. 21 558–21 572, 2023.

[13] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, L. Zhang, Z. Li, and Y. Ma, "Exploring and evaluating hallucinations in llm-powered code generation," *arXiv preprint arXiv:2404.00971*, 2024.

[14] S. Hamer, M. d'Amorim, and L. Williams, "Just another copy and paste? comparing the security vulnerabilities of chatgpt generated code and stackoverflow answers," in *2024 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2024, pp. 87–94.

[15] M. L. Siddiq, S. Dristi, J. Saha, and J. C. Santos, "The fault in our stars: Quality assessment of code generation benchmarks," in *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2024, pp. 201–212.

[16] M. T. Jamil, S. Abid, and S. Shamail, "Can llms generate higher quality code than humans? an empirical study," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 2025, pp. 478–489.

[17] V. Pontillo, F. Palomba, and F. Ferrucci, "Static test flakiness prediction: How far can we go?" *Empirical Software Engineering*, vol. 27, no. 7, p. 187, 2022.

[18] V. Pontillo, L. Martins, I. Machado, F. Palomba, and F. Ferrucci, "An empirical investigation into the capabilities of anomaly detection approaches for test smell detection," *Journal of Systems and Software*, vol. 222, p. 112320, 2025.

[19] G. Recupito, G. Giordano, F. Ferrucci, D. Di Nucci, and F. Palomba, "When code smells meet ml: on the lifecycle of ml-specific code smells in ml-enabled systems," *Empirical Software Engineering*, vol. 30, no. 5, p. 139, 2025.

[20] G. Giordano, G. Festa, G. Catolino, F. Palomba, F. Ferrucci, and C. Gravino, "On the adoption and effects of source code reuse on defect proneness and maintenance effort," *Empirical Software Engineering*, vol. 29, no. 1, p. 20, 2024.

[21] G. Giordano, G. Sellitto, A. Sepe, F. Palomba, and F. Ferrucci, "The yin and yang of software quality: On the relationship between design patterns and code smells," in *2023 49th Euromicro conference on software engineering and advanced applications (SEAA)*. IEEE, 2023, pp. 227–234.

[22] L. Martins, V. Pontillo, H. Costa, F. Ferrucci, F. Palomba, and I. Machado, "Test code refactoring unveiled: where and how does it affect test code quality and effectiveness?" *Empirical Software Engineering*, vol. 30, no. 1, p. 27, 2025.

[23] A. Afeltra, A. Cannavale, F. Pecorelli, V. Pontillo, and F. Palomba, "A large-scale empirical investigation into cross-project flaky test prediction," *IEEE Access*, vol. 12, pp. 131 255–131 265, 2024.

[24] V. Pontillo, D. Amoroso d'Aragona, F. Pecorelli, D. Di Nucci, F. Ferrucci, and F. Palomba, "Machine learning-based test smell detection," *Empirical Software Engineering*, vol. 29, no. 2, p. 55, 2024.

[25] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, "On the dangers of stochastic parrots: Can language models be too big?" in *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*, 2021, pp. 610–623.

[26] Z. Zhang, C. Wang, Y. Wang, E. Shi, Y. Ma, W. Zhong, J. Chen, M. Mao, and Z. Zheng, "Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 481–503, 2025.

[27] A. Keluskar, A. Bhattacharjee, and H. Liu, "Do llms understand ambiguity in text? a case study in open-world question answering," in *2024 IEEE International Conference on Big Data (BigData)*. IEEE, 2024, pp. 7485–7490.

[28] A. Della Porta, G. Voria, A. Abbate, R. Sulipano, S. Lambiase, G. Catolino, and F. Palomba, "Toward measuring prompt quality: A preliminary investigation on prompt smells," in *2026 IEEE International Conference on Software Analysis, Evolution and Reengineering-Companion (SANER-C)*. IEEE, 2026, pp. 293–300.

[29] Y. Sasaki, H. Washizaki, J. Li, D. Sander, N. Yoshioka, and Y. Fukazawa, "Systematic literature review of prompt engineering patterns in software engineering," in *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2024, pp. 670–675.

[30] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design," in *Generative AI for Effective Software Development*. Springer, 2024, pp. 71–108.

[31] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang *et al.*, "Self-refine: Iterative refinement with self-feedback," *Advances in Neural Information Processing Systems*, vol. 36, pp. 46 534–46 594, 2023.

[32] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[33] P. Xia, L. Zhang, and F. Li, "Learning similarity with cosine similarity ensemble," *Information sciences*, vol. 307, pp. 39–52, 2015.

[34] T. Y. Zhuo, "Ice-score: Instructing large language models to evaluate code," in *Findings of the Association for Computational Linguistics: EACL 2024*, 2024, pp. 2232–2242.

[35] A. Della Porta, S. Lambiase, V. Pontillo, J. F. Ferreira, A. A. Rahman, C. Ragkhitwetsagul, and F. Palomba, "Understanding prompt quality and its relation to llm-based code generation—replication package," https://doi.org/10.6084/m9.figshare.30109213.

[36] Z. Namrud, K. Sarda, M. Litoiu, L. Shwartz, and I. Watts, "Kubeplaybook: A repository of ansible playbooks for kubernetes auto-remediation with llms," in *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '24 Companion. New York, NY, USA: Association for Computing Machinery, 2024, p. 57–61. [Online]. Available: https://doi.org/10.1145/3629527.3653665

[37] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2023.

[38] L. Ma, S. Liu, Y. Li, X. Xie, and L. Bu, "Specgen: Automated generation of formal program specifications via large language models," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 16–28.

[39] G. Voria, F. Casillo, C. Gravino, G. Catolino, and F. Palomba, "Recover: Toward requirements generation from stakeholders' conversations," *IEEE Transactions on Software Engineering*, 2025.

[40] G. Voria, B. Scala, L. Todisco, C. Venditto, G. Giordano, G. Catolino, and F. Palomba, "Fair and square? evaluating fairness of llm-generated synthetic datasets," *Information and Software Technology*, p. 107980, 2025.

[41] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639117

[42] X. Lian, Y. Chen, R. Cheng, J. Huang, P. Thakkar, M. Zhang, and T. Xu, "Large language models as configuration validators," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 1704–1716.

[43] C. Wang, J. Liu, X. Peng, Y. Liu, and Y. Lou, "Boosting static resource leak detection via llm-based resource-oriented intention inference," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 2905–2917.

[44] N. Zahan, P. Burckhardt, M. Lysenko, F. Aboukhadijeh, and L. Williams, "Leveraging large language models to detect npm malicious packages," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 2625–2637.

[45] M. Cicalese, A. Della Porta, S. Lambiase, E. Iannone, T. Hinrichs, R. Scandariato, and F. Palomba, "The language of security: How prompt syntax shapes secure code generation in open llms," *arXiv preprint arXiv:2508.13666*, 2026.

[46] S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W.-H. Chiang, Y. Lyu, H. Nguyen, and O. Tripp, "A deep dive into large language models for automated bug localization and repair," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660773

[47] F. Batole, D. OBrien, T. N. Nguyen, R. Dyer, and H. Rajan, "An llm-based agent-oriented approach for automated code design issue localization," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 1320–1332.

[48] M. Yuan, J. Chen, Z. Xing, A. Quigley, Y. Luo, T. Luo, G. Mohammadi, Q. Lu, and L. Zhu, "Designrepair: Dual-stream design guideline-aware frontend repair with large language models," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 2483–2494.

[49] J. Xu, Y. Fu, S. H. Tan, and P. He, "Aligning the objective of llm-based program repair," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 2548–2560.

[50] S. Feng and C. Chen, "Prompting is all you need: Automated android bug replay with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3608137

[51] R. Choudhuri, D. Liu, I. Steinmacher, M. Gerosa, and A. Sarma, " How Far Are We? The Triumphs and Trials of Generative AI in Learning Software Engineering ," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2024, pp. 2270–2282. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/3597503.3639201

[52] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3491101.3519665

[53] J. H. Klemmer, S. A. Horstmann, N. Patnaik, C. Ludden, C. Burton, C. Powers, F. Massacci, A. Rahman, D. Votipka, H. R. Lipford, A. Rashid, A. Naiakshina, and S. Fahl, "Using ai assistants in software development: A qualitative study on

security practices and concerns," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 2726–2740. [Online]. Available: https://doi.org/10.1145/3658644.3690283

[54] J. T. Liang, C. Badea, C. Bird, R. DeLine, D. Ford, N. Forsgren, and T. Zimmermann, "Can gpt-4 replicate empirical software engineering research?" *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660767

[55] S. Lambiase, G. Catolino, F. Palomba, F. Ferrucci, and D. Russo, "Exploring individual factors in the adoption of llms for specific software engineering tasks," *arXiv preprint arXiv:2504.02553*, 2025.

[56] ——, "Investigating the role of cultural values in adopting large language models for software engineering," *ACM Transactions on Software Engineering and Methodology*, vol. 35, no. 1, pp. 1–43, 2025.

[57] S. Lambiase, G. Catolino, F. Palomba, and F. Ferrucci, "Motivations, challenges, best practices, and benefits for bots and conversational agents in software engineering: A multivocal literature review," *ACM Computing Surveys*, vol. 57, no. 4, pp. 1–37, 2024.

[58] T. Tavantzis, S. Lambiase, D. Russo, and R. Feldt, "From challenge to change: Design principles for ai transformations," *arXiv preprint arXiv:2512.05533*, 2025.

[59] T. Tavantzis and R. Feldt, "Unpacking organizational change in ai transformations of software engineering," in *2025 IEEE/ACM 18th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 2025, pp. 149–160.

[60] M. A. Jishan, M. W. Allvi, and M. A. K. Rifat, "Analyzing user prompt quality: Insights from data," in *2024 International Conference on Decision Aid Sciences and Applications (DASA)*. IEEE, 2024, pp. 1–5.

[61] X. Fang, W. Wang, X. Lv, and J. Yan, "Pcqa: A strong baseline for aigc quality assessment based on prompt condition," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 6167–6176.

[62] V. Rawte, P. Priya, S. Tonmoy, S. Zaman, A. Sheth, and A. Das, "Exploring the relationship between llm hallucinations and prompt linguistic nuances: Readability, formality, and concreteness," *arXiv preprint arXiv:2309.11064*, 2023.

[63] E. Santana Jr, G. Benjamin, M. Araujo, H. Santos, D. Freitas, E. Almeida, P. A. d. Neto, J. Li, J. Chun, and I. Ahmed, "Which prompting technique should i use? an empirical investigation of prompting techniques for software engineering tasks," *arXiv preprint arXiv:2506.05614*, 2025.

[64] H. Puerto, M. Tutek, S. Aditya, X. Zhu, and I. Gurevych, "Code prompting elicits conditional reasoning abilities in text+ code llms," *arXiv preprint arXiv:2401.10065*, 2024.

[65] S. K. Karmaker Santu and D. Feng, "TELeR: A general taxonomy of LLM prompts for benchmarking complex tasks," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 14 197–14 203. [Online]. Available: https://aclanthology.org/2023.findings-emnlp.946/

[66] M. M. Hassan, J. Salvador, S. K. K. Santu, and A. Rahman, "State reconciliation defects in infrastructure as code," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660790

[67] F. Qiu, P. Ji, B. Hua, and Y. Wang, "Chemfuzz: Large language models-assisted fuzzing for quantum chemistry software bug detection," in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, 2023, pp. 103–112.

[68] P. Zhan, Z. Xu, Q. Tan, J. Song, and R. Xie, "Unveiling the lexical sensitivity of llms: Combinatorial optimization for prompt enhancement," pp. 5128–5154, 2024.

[69] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul *et al.*, "Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions," *arXiv preprint arXiv:2406.15877*, 2024.

[70] J. Gu, X. Jiang, Z. Shi, H. Tan, X. Zhai, C. Xu, W. Li, Y. Shen, S. Ma, H. Liu *et al.*, "A survey on llm-as-a-judge," *arXiv preprint arXiv:2411.15594*, 2024.

[71] P. Ralph and S. Baltes, "Paving the way for mature secondary research: the seven types of literature review," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1632–1636.

[72] P. Payoungkhamdee, P. Tuchinda, J. Baek, S. Cahyawijaya, C. Udomcharoenchaikit, P. Manakul, P. Limkonchotiwat, E. Chuangsuwanich, and S. Nutanong, "Towards better understanding of program-of-thought reasoning in cross-lingual and multilingual environments," in *Findings of the Association for Computational Linguistics: ACL 2025*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds. Vienna, Austria: Association for Computational Linguistics, Jul. 2025, pp. 15 810–15 828. [Online]. Available: https://aclanthology.org/2025.findings-acl.817/

[73] Z. Li, Y. Hua, T. Vu, H. Zhan, L. Qu, and G. Haffari, "Scar: Data selection via style consistency-aware response ranking for efficient instruction-tuning of large language models," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2025, pp. 12 756–12 790.

[74] R. Flesch, "A new readability yardstick." *Journal of applied psychology*, vol. 32, no. 3, p. 221, 1948.

[75] R. Gunning, "The technique of clear writing," *(No Title)*, 1952.

[76] A. Miranda-García and J. Calle-Martín, "Yule's characteristic k revisited," *Language Resources and Evaluation*, vol. 39, no. 4, pp. 287–294, 2005.

[77] D. Schreiter, "Prompt engineering: How prompt vocabulary affects domain knowledge," *arXiv preprint arXiv:2505.17037*, 2025.

[78] S. Pawar, M. Apte, K. Jadhav, G. K. Palshikar, and N. Ramrakhiyani, "Broken words, broken performance: Effect of tokenization on performance of llms," in *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, 2025, pp. 372–385.

[79] E. Dale and J. S. Chall, "A formula for predicting readability: Instructions," *Educational research bulletin*, pp. 37–54, 1948.

[80] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.

[81] I. E. C. International Organization for Standardization, 2022. [Online]. Available: https://www.iso.org/standard/80710.html

[82] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[83] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[84] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.

[85] Z. Zhang, A. Zhang, M. Li, and A. Smola, "Automatic chain of thought prompting in large language models," *arXiv preprint arXiv:2210.03493*, 2022.

[86] J. Li, G. Li, Y. Li, and Z. Jin, "Structured chain-of-thought prompting for code generation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, pp. 1–23, 2025.

[87] J. Li, P. Chen, B. Xia, H. Xu, and J. Jia, "Motcoder: Elevating large language models with modular of thought for challenging programming tasks," *arXiv preprint arXiv:2312.15960*, 2023.

[88] C. Li, J. Liang, A. Zeng, X. Chen, K. Hausman, D. Sadigh, S. Levine, L. Fei-Fei, F. Xia, and B. Ichter, "Chain of code: Reasoning with a language model-augmented code emulator," *arXiv preprint arXiv:2312.04474*, 2023.

[89] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba, "Large language models are human-level prompt engineers," in *The eleventh international conference on learning representations*, 2022.

[90] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén *et al.*, *Experimentation in software engineering*. Springer, 2012, vol. 236.

[91] T. Xiao, C. Treude, H. Hata, and K. Matsumoto, "Devgpt: Studying developer-chatgpt conversations," in *Proceedings of the 21st international conference on mining software repositories*, 2024, pp. 227–230.

[92] K. Jin, C.-Y. Wang, H. V. Pham, and H. Hemmati, "Can chatgpt support developers? an empirical evaluation of large language models for code generation," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 167–171.

[93] Y. Tao, A. Agrawal, J. Dombi, T. Sydorenko, and J. I. Lee, "Chatgpt role-play dataset: Analysis of user motives and model naturalness," *arXiv preprint arXiv:2403.18121*, 2024.
[94] K. Pister, D. J. Paul, I. Joshi, and P. Brophy, "Promptset: A programmer's prompting dataset," in *Proceedings of the 1st International Workshop on Large Language Models for Code*, 2024, pp. 62–69.
[95] A. Della Porta, S. Lambiase, and F. Palomba, "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*, 2025, pp. 181–192.
[96] E. Spiliopoulou, R. Fogliato, H. Burnsky, T. Soliman, J. Ma, G. Horwood, and M. Ballesteros, "Play favorites: A statistical method to measure self-bias in llm-as-a-judge," *arXiv preprint arXiv:2508.06709*, 2025.
[97] S. Baltes, F. Angermeir, C. Arora, M. M. Barón, C. Chen, L. Böhme, F. Calefato, N. Ernst, D. Falessi, B. Fitzgerald, D. Fucci, M. Kalinowski, S. Lambiase, D. Russo, M. Lungu, L. Prechelt, P. Ralph, R. van Tonder, C. Treude, and S. Wagner, "Guidelines for empirical studies in software engineering involving large language models," 2025. [Online]. Available: https://arxiv.org/abs/2508.15503
[98] M. A. Hardy, *Regression with dummy variables*. Sage, 1993, no. 93.
[99] J. Fox, *Applied regression analysis and generalized linear models*. Sage publications, 2015.
[100] F. Faul, E. Erdfelder, A.-G. Lang, and A. Buchner, "G* power 3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences," *Behavior research methods*, vol. 39, no. 2, pp. 175–191, 2007.
[101] F. Faul, E. Erdfelder, A. Buchner, and A.-G. Lang, "Statistical power analyses using g* power 3.1: Tests for correlation and regression analyses," *Behavior research methods*, vol. 41, no. 4, pp. 1149–1160, 2009.
[102] M. Smithson and J. Verkuilen, "A better lemon squeezer? maximum-likelihood regression with beta-distributed dependent variables." *Psychological methods*, vol. 11, no. 1, p. 54, 2006.
[103] D. Pan, Z. Sun, C. Zhang, D. Lo, and X. Du, "The hidden cost of readability: How code formatting silently consumes your llm budget," *arXiv preprint arXiv:2508.13666*, 2025.

BIOGRAPHIES

Antonio Della Porta is a Ph.D student at the University of Salerno, Italy. His research interests include software engineering for AI, quantum software engineering, software maintenance and evolution, and empirical software engineering. He received the distinguished reviewer award on the Shadow PC of ICSE'26 and is part of the Program Committee of ICSE'27. For more information, see https://atdepo.github.io.

Stefano Lambiase is currently an Assistant Professor at Aalborg University, Copenhagen, Denmark. Before joining Aalborg University, he was a Postdoctoral Researcher with the Department of Computer Science, University of Salerno. His research interests include software engineering, with a particular emphasis on software project management, collaborative and socio-technical aspects, SE4AI, quantum software engineering, video game development, and empirical research methods. He has served on numerous program committees and reviewed more than 100 articles for leading software engineering venues. For more information, see https://stefanolambiase.github.io.

Valeria Pontillo is a postdoctoral researcher at the Gran Sasso Science Institute (GSSI), Italy. Her research interests include test code quality, predictive analytics, mining software repositories, software maintenance and evolution, empirical software engineering, and security aspects in software code. She serves and has served as a reviewer for international conferences and journals in the software engineering field. For more information, see https://valeriapontillo.github.io/.

João F. Ferreira received the Ph.D. degree in computer science from the University of Nottingham, U.K. He is an Associate Professor with the Faculty of Engineering, University of Porto, Portugal, and a Researcher with INESC-ID. His research interests are centred on software reliability, encompassing both empirical and formal methods applied to software engineering. For more information, see https://joaoff.com.

Akond A. Rahman is an Assistant Professor at Auburn University. His research interests include DevOps and Secure Software Development. He graduated with a PhD from North Carolina State University, an M.Sc. from University of Connecticut, and a B.Sc. from Bangladesh University of Engineering and Technology (BUET). He is the recipient of the Research Excellence Award from Auburn University in 2025, the CSC Distinguished Dissertation Award, and the CoE Distinguished Dissertation Award from NC State in 2020. He actively collaborates with industry practitioners from GitHub, WindRiver, and others.

Chaiyong Ragkhitwetsagul received the Ph.D. degree in computer science from the University College London. He is an Assistant Professor in the Faculty of Information and Communication Technology (ICT) at Mahidol University, Thailand. His research interests include software engineering for AI, AI for software engineering, empirical software engineering, coding proficiency, software dependencies, software visualization, code search, code similarity, code clone detection, and mining software repositories. For more information, see https://cragkhit.github.io.

Fabio Palomba is an Associate Professor at the University of Salerno. His research focuses on software maintenance and evolution, empirical software engineering, source code quality, and mining software repositories. He has received multiple distinguished and best paper awards at top-tier software engineering venues, serves on the editorial boards of leading journals in the field, and was awarded the 2023 IEEE/TCSE Rising Star Award. For more information, see https://fpalomba.github.io.