Towards Detecting Prompt Knowledge Gaps for Improved LLM-guided Issue Resolution

Ramtin Ehsani
Drexel University
Philadelphia, PA, USA
ramtin.ehsani@drexel.edu

Sakshi Pathak
Drexel University
Philadelphia, PA, USA
sp3856@drexel.edu

Preetha Chatterjee
Drexel University
Philadelphia, PA, USA
preetha.chatterjee@drexel.edu

***Abstract*—Large language models (LLMs) have become essential in software development, especially for issue resolution. However, despite their widespread use, significant challenges persist in the quality of LLM responses to issue resolution queries. LLM interactions often yield incorrect, incomplete, or ambiguous information, largely due to knowledge gaps in prompt design, which can lead to unproductive exchanges and reduced developer productivity.**

In this paper, we analyze 433 developer-ChatGPT conversations within GitHub issue threads to examine the impact of prompt knowledge gaps and conversation styles on issue resolution. We identify four main knowledge gaps in developer prompts: *Missing Context, Missing Specifications, Multiple Context*, and *Unclear Instructions*. Assuming that conversations within closed issues contributed to successful resolutions while those in open issues did not, we find that ineffective conversations contain knowledge gaps in 54.7% of prompts, compared to only 13.2% in effective ones. Additionally, we observe seven distinct conversational styles, with *Directive Prompting*, *Chain of Thought*, and *Responsive Feedback* being the most prevalent. We find that knowledge gaps are present in all styles of conversations, with *Missing Context* being the most repeated challenge developers face in issue-resolution conversations.

Based on our analysis, we identify key textual and code-related heuristics—*Specificity*, *Contextual Richness*, and *Clarity*—that are associated with successful issue closure and help assess prompt quality. These heuristics lay the foundation for an automated tool that can dynamically flag unclear prompts and suggest structured improvements. To test feasibility, we developed a lightweight browser extension prototype for detecting prompt gaps, that can be easily adapted to other tools within developer workflows.

***Index Terms*—issue resolution, large language models, prompt quality**

I. Introduction

Large language models such as ChatGPT, Gemini, and Claude have become crucial tools for software development. The 2023 JetBrains survey, which gathered responses from 26k developers across 196 countries, 77% (i.e., three out of four developers) use ChatGPT for daily tasks [1]. LLMs are transforming the way developers approach problem-solving, particularly in issue resolution [2], [3]. Developers seek guidance from these models to troubleshoot and refine solutions. By providing real-time feedback, helping to debug, and accelerating problem-solving, LLMs have become integral to the issue resolution process [4], [5].

Despite their popularity, several concerns remain regarding the quality of LLM responses to issue resolution-related queries. Recent studies have shown that these interactions often yield incomplete, ambiguous, or incorrect information [6]–[9]. LLM responses are highly sensitive to the information provided in the prompts [10], [11]. Therefore, *prompt knowledge gaps* (e.g., missing context, unclear instructions) play a critical role in shaping these interactions. These gaps can lead to irrelevant responses or even hallucinations [7]. Knowledge gaps in prompt design also lead to multiple back-and-forth interactions, resulting in unsuccessful conversational outcomes and decreased developer productivity [12]. These challenges highlight the need for a deeper understanding of how prompt gaps and styles affect LLM-driven issue resolution.

Previous research has examined gaps in developer-LLM conversations across different software engineering tasks, identifying factors that prolong conversations and require multiple prompts for developers to obtain helpful responses [11], [13], [14]. However, these studies do not identify the unique challenges and knowledge gaps specific to issue resolution, such as the need for precise problem descriptions and detailed error messages. A recent study analyzing 85 ChatGPT conversations related to issue resolution reveals 11 types of gaps in both developer prompts and ChatGPT's responses that contribute to longer exchanges [7]. Despite these findings, existing research has yet to systematically examine how knowledge gaps in developer prompts impact the effectiveness of LLM-driven issue resolution. While prompt engineering methods aim to refine LLM responses by optimizing phrasing, structure, and style [13], they often fail to address a deeper challenge: providing actionable, targeted guidance that empowers developers to create more effective prompts. As a result, effective issue resolution still heavily depends on developers' ability to identify the knowledge gap, and incorporate the necessary information in the prompts.

In this paper, we analyze 433 developer-ChatGPT conversations shared within GitHub issue threads to investigate the impact of *prompt knowledge gaps* and *conversation styles* on issue resolution. We focus on knowledge gaps, such as identifying missing or ambiguous content, that impact the effectiveness of developer-LLM discussions related to issue resolution. By identifying key textual and code-related heuristics that are related to these gaps, we explore patterns in conversations that are associated with successful issue closure on GitHub. Our objective is to identify actionable heuristics that can inform the design of a tool to dynamically flag unclear or incomplete prompts, and suggest improvements in the form of structured templates. Toward that goal, we investigate the following research questions:

RQ1: *How do prompt knowledge gaps and conversation styles influence the progression and effectiveness of developer-ChatGPT conversations in issue resolution?* We annotate each developer-ChatGPT conversation with four categories of knowledge gaps and seven styles of conversations. By analyzing the content and discourse of each conversation, we investigate their influence on the GitHub issue status (open vs. closed). We find that developers use different styles and techniques such as *Chain of Thought* and *Directive Prompting* to interact with ChatGPT, however, knowledge gaps persist across all styles. Developers struggle with providing the right context, with *Missing Context* emerging as the most common issue associated with unsuccessful conversations.

RQ2: *What heuristics can be used to automatically measure the prompt knowledge gaps?* Based on the results of RQ1, we design three categories of textual and code-related heuristics (*Contextual Richness, Specificity, and Clarity*) that capture the nuances of knowledge gaps in prompts when using ChatGPT for issue resolution. We found that providing short code snippets, additional information such as links to documentation, and error messages in the prompt, while maintaining the conversation on the same topic can lead to more effective issue resolution conversations. These heuristics provide a foundation for designing tools for automatic detection of prompt gaps.

To demonstrate the feasibility of using our RQ2 heuristics in an automated tool, we develop a lightweight prototype. Implemented as a browser extension, this prototype can be adapted to other tools within developer workflows. Our work takes the first step towards automated prompt knowledge gap detection in LLM-aided issue-resolution conversations. By providing targeted, actionable suggestions, this tool could help developers proactively enhance prompt quality. The key contributions in this paper are summarized as follows:

- We present a manually annotated dataset of developer-ChatGPT conversations focused on issue resolution, with annotations for prompt styles and knowledge gaps as the conversation progresses.
- We conduct a comprehensive analysis of these conversations, identifying common knowledge gaps and styles, and uncovering key heuristics that are strongly associated with the effectiveness of the interactions towards issue resolution.
- We develop the first prototype for automatically detecting knowledge gaps in prompts during ChatGPT-based issue resolution, offering tailored templates to improve prompt quality and enhance the likelihood of successful outcomes.

II. Dataset

We use the DevGPT dataset [15] for our analysis. This dataset contains developer-ChatGPT conversations that are publicly shared through links on platforms such as GitHub and Hackernews. These conversations are generated using OpenAI's web-browser platform of ChatGPT, which utilizes either GPT-3.5 or GPT-4. Since the focus of our study is issue resolution, we selected conversations shared within GitHub issue threads. These conversations contain a wide variety of queries directed at ChatGPT, including how-to questions, advanced programming guidance, inquiries about frameworks, and high-level design recommendations. [12], [16], [17]. The conversations in this dataset often serve as references for potential solutions or helpful context [12], and they are inherently related to the issues because developers intentionally share them as resources they believe might assist in resolving the problem. We further analyzed the dataset and observed that the queries and subtasks presented to ChatGPT vary from straightforward ones such as API usage and syntax fixes to more complex debugging and multi-threading issues. Simpler tasks required fewer interactions and were resolved more effectively, while complex ones were more challenging. For example, resolving syntax errors like "How do I fix a syntax error in Python?" was far more straightforward than diagnosing a segmentation fault in C. While task difficulty impacts ChatGPT's performance, our focus remains on assessing how providing sufficient detail within prompts influences issue resolution regardless of the difficulty of issues. In addition, the diversity of subtasks covered in these conversations allows us to generalize our findings to a wide range of issue-resolution challenges.

Each dataset entry contains the ChatGPT link, the associated GitHub issue, the full conversation comprising each prompt and its corresponding ChatGPT response, and the saved HTML content of the conversation. The original dataset consists of 636 entries. We filtered out duplicate entries and non-English conversations using Python's *lingua-py* library [18]. Code snippets and error messages were replaced with [CODE] and [ERROR] tags, respectively, and separated from the text. ChatGPT responses structure code snippets in quote blocks, allowing for easy replacement using RegEx, while developer prompts often do not. Following previous studies, to detect unstructured code and error messages in prompts, we used *GPT-4* [19]. One of the authors manually validated the dataset to ensure accuracy. Our final dataset comprises 433 developer-ChatGPT conversations shared within 400 unique GitHub issues.

III. Methodology

A. RQ1: How do prompt knowledge gaps and conversation styles influence the progression and effectiveness of developer-ChatGPT conversations in issue resolution?

We annotate 433 developer-ChatGPT conversations, focusing on two main aspects: prompt knowledge gaps (i.e., deficiencies in the content of prompts) and conversation styles (i.e., the techniques developers used to communicate with ChatGPT). By annotating the dataset according to these two aspects, we assess how knowledge gaps and conversation styles contribute to the effectiveness of issue resolution. We assume that conversations within closed issues likely contributed to successful resolutions, while those within open issues did not effectively aid in resolving the issues. This approach is the best available option for evaluating the relationship between prompt knowledge and issue resolution. Our study focuses on understanding how prompt quality is associated with the likelihood of issue resolution. Given this, the status of the issue (open or closed) provides a direct and practical measure of effectiveness.

We followed a qualitative content analysis approach [20], combining both deductive and inductive coding methods. We began with a set of predefined categories for prompt knowledge gaps and conversation styles, derived from existing taxonomies and literature. Using this strategy, two authors of this paper independently annotated an initial subset of conversations to capture prompt gaps and conversation styles. Through iterative coding and discussion, we refined the categories, leading to the creation of modified taxonomies for both prompt knowledge gaps and conversation styles for issue resolution. This inductive refinement allowed us to adapt our categories based on observed data patterns, enhancing the validity of our coding scheme. To ensure reliability, we calculated Cohen's Kappa scores at each stage, achieving strong inter-rater reliability in the final round, and further validated our consistency through a blinded sample check. Next, we discuss the evolution of the taxonomy and further details on the annotation procedure.

To identify the prompt knowledge gaps, we started with Mondal et al.'s prompt gap taxonomy consisting of a total of 11 categories: *Missing Specifications, Different Use Cases, Incremental Problem Solving, Exploring Alternative Approaches, Wordy Response, Additional Functionality, Erroneous Response, Missing Context, Clarity of Generated Response, Inaccurate/Untrustworthy Response, and Miscellaneous* [7]. To identify the conversation styles, we started with 18 categories: *Meta Language Creation, Output Automator, Persona, Visualization Generator, Template, Skeleton of Thought, Chain of Thought, Tree of Thought, Fact Check List, Meta-prompting, Reflection, Responsive Feedback, Question Refinement, Alternative Approaches, Cognitive Verifier, Refusal Breaker, Game Play*, and *Few-shot Learning*, drawn from the literature on interaction styles with LLMs [21]–[23]. Two authors independently reviewed an initial set of 50 ChatGPT conversations. We annotated the first prompt in each conversation to capture its initial gaps, while subsequent prompts were annotated based on new information the developers provided, allowing us to observe how knowledge gaps evolved through the conversation. After identifying prompt knowledge gaps, each conversation was categorized with a conversation style that reflected the overall interaction across all prompts. The inter-rater agreement for the first round of annotations was Cohen's Kappa of 0.62 for gaps and 0.48 for styles, indicating moderate agreement [24]. In the second round of annotation, the authors revisited the initial 50 conversations along with 50 additional ones. The final Cohen's Kappa agreement was 0.84 for gaps and 0.72 for styles, both reflecting strong inter-rater reliability [24]. Conflicts in the annotations were iteratively discussed and resolved collaboratively with both annotators contributing equally, leading to a refinement of the initial categories. Since, we had high inter-rater agreement after the second iteration, the rest of the dataset was split among the two researchers to complete the annotation independently.

Based on our iterative discussions, for conversation styles seven categories were discarded because they were not present in our dataset. These categories were *Meta Language Creation, Output Automator, Visualization Generator, Fact Check List, Cognitive Verifier, Refusal Breaker,* and *Game Play*. In addition, eight categories were merged into three categories because they represented the same style: *Responsive Feedback, Meta-prompting, Reflection, Question Refinement* into *Responsive Feedback*; *Template* and *Skeleton of Thought* into *Template*; and *Tree of Thought* and *Alternative Approaches* into *Tree of Thought*. The final set included six categories, plus one additional style (Directive Prompting) derived from our coding. The prompt knowledge gaps were consolidated into three main categories, with an additional gap (Unclear Instructions) emerging from our open coding process [25]. The categories discarded for prompt knowledge gaps were *Different Use Cases, Incremental Problem Solving, Exploring Alternative Approaches, Wordy Response, Additional Functionality, Erroneous Response, Clarity of Generated Response,* and *Inaccurate/Untrustworthy Response*.

Additionally, to ensure our annotations were not biased based on conversations' status (open vs. closed), we sampled 50 conversations after the annotation, hiding their status and redoing the annotation to see if we identified different gaps or styles in the conversations. In only 4 conversations (1 open and 3 closed) we identified additional gaps in prompts indicating that our annotations were consistent.

We now present the refined taxonomies. Prompt Knowledge Gap Categories: As summarized in Table I, we categorize prompt knowledge gaps into four groups: *Missing Context*, *Missing Specification*, *Unclear Instruction*, and *Multiple Context*. These gaps are essential for evaluating whether the developer provided enough information and clarity for ChatGPT to understand and resolve the issue.

TABLE I: Prompt Knowledge Gaps in Developer-ChatGPT Conversations

Category	**Description**
Missing Context	Lacks essential details, such as goals, previous attempts, or project info.
Multiple Context	Introduces multiple issues without clear separation, leading to confusion.
Unclear Instructions	Instructions are vague or open to multiple interpretations, leading to ineffective responses.
Missing Specification	Lacks critical technical information (e.g., programming language).

Context refers to the background information that developers provide to help ChatGPT understand the problem. We identified gaps in context by looking for prompts that lacked sufficient background information. A prompt was labeled as *Missing Context* if it did not provide essential details like the user's end goals, prior attempts to solve the issue, codes and error logs, or relevant project information [26]. On the other hand, a prompt was classified as *Multiple Context* when it introduced more than one distinct issue in the same conversation thread. This often leads to confusion in responses, as ChatGPT struggles to focus on one problem.

Instructions refer to the explicit steps or actions that developers want ChatGPT to perform. Unambiguous instructions are crucial for obtaining relevant and accurate responses. We analyzed prompts to identify instances where the instructions provided were unclear or open to multiple interpretations. Unclear instructions with grammatical issues, misspellings, or anything that hinders the understanding of the instruction is classified as *Unclear Instructions*. One such example is:*"noe to how to run all togathor and display in website"*.

Specification relates to the technical details and system requirements specific to the issue. Effective prompts should contain enough technical information, such as exact programming language, performance constraints, or versions of the frameworks to guide ChatGPT in generating precise solutions. We categorized prompts as having a *Missing Specification* gap if they lacked essential technical details necessary for providing a meaningful solution.

Conversation Style Categories: We identified seven conversation styles as follows: *Persona*, *Template*, *Chain of Thought*, *Tree of Thought*, *Responsive Feedback*, *Few-shot Learning*, and *Directive Prompting*.

Persona is a style where developers instruct ChatGPT to assume a specific role or perspective. By asking ChatGPT to "act as a cybersecurity expert" or "explain this as a mentor would," developers can tailor responses to align with their specific needs. This style is particularly useful when the developer requires expert-level advice or wants the response framed in a particular way [21].

Template is a style where developers provide a predefined structure for ChatGPT to follow in its output. Developers often use this style when they need the response to adhere to a specific format, such as a documentation template or structured report. By giving ChatGPT a template to follow, developers ensure consistency in responses, particularly when the output must follow a standardized format [21].

Chain of Thought breaks down complex tasks into logical, sequential steps. Instead of asking ChatGPT for an immediate solution, the developer prompts the model to think through the problem step by step. This style is especially effective for multifaceted problems where each step needs to be carefully considered [22], [23].

Tree of Thought expands on the Chain of Thought approach by encouraging ChatGPT to explore multiple possible solutions or pathways. In situations where there is more than one potential solution, this style allows developers to prompt ChatGPT to branch out and explore various scenarios or alternative strategies [22], [23].

Responsive Feedback is a style where developers provide feedback directly within the prompting process to refine ChatGPT's responses. For example, after receiving an initial output, the developer might give feedback such as "I like this part, but can you make it simpler?" This allows for iterative improvements and dynamic interaction, leading to more refined and tailored responses [21], [22].

Few-shot Learning is when developers provide a few examples within the prompt to illustrate their request. By including these examples, developers can help ChatGPT better understand the task and generate responses that align with their expectations [22], [23].

Directive Prompting is a style where we identified developers provide goals and the scope of issues to direct ChatGPT toward a specific outcome. This style is straightforward with developers knowing what they exactly want, which helps reduce ambiguity and ensures that the conversation stays focused on the desired solution.

B. RQ2: What heuristics can be used to automatically measure the prompt knowledge gaps?

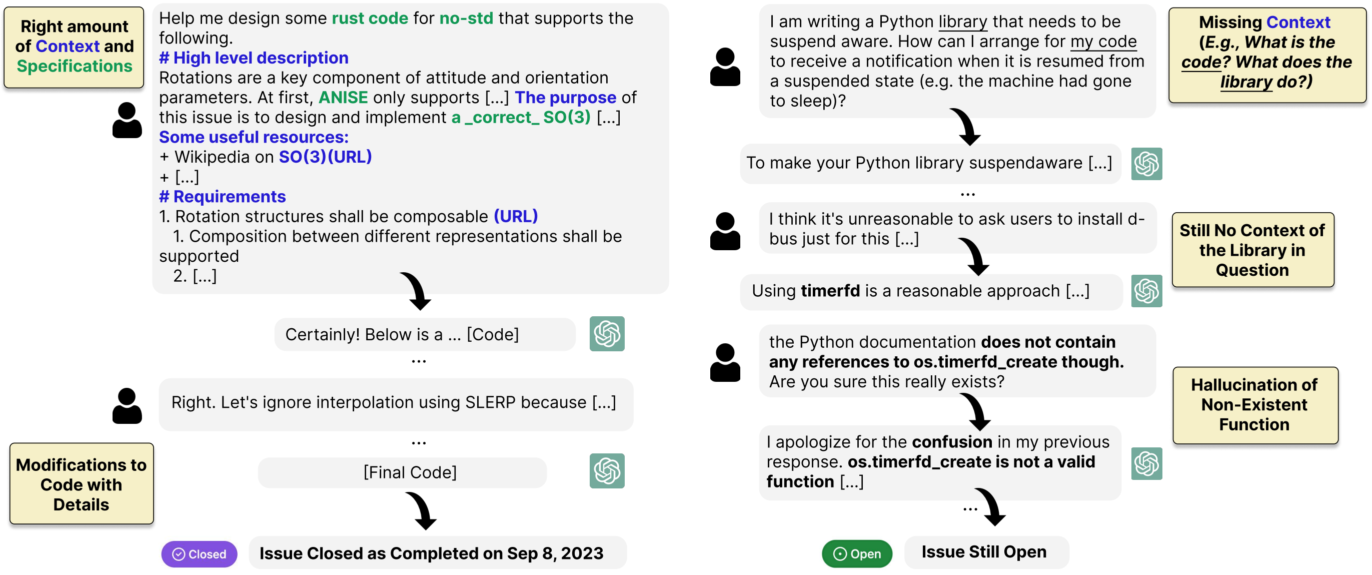
Based on the results of RQ1 and further analysis of the content of developer prompts, we design three categories of heuristics: *Specificity*, *Contextual Richness*, and *Clarity* (see Table II). These heuristics are coming from our analysis of what constitutes knowledge gaps in developer prompts and were derived using NLP and code-related metrics to directly correspond to the knowledge gaps identified in prompts. *Contextual Richness* helps identify *Missing Context* and *Multiple Context* by measuring the inclusion of code, external references, error messages, and other necessary information that developers may overlook. *Specificity* addresses *Missing Specification* by evaluating if the prompt contains enough technical details to frame the developers' requests. Finally, *Clarity* tackles *Unclear Instructions* by analyzing how cohesively instructions are structured, aiming to detect any ambiguous language that could lead to misunderstandings.

Additionally, to assess the impact of these heuristics on the effectiveness of issue resolution, we quantitatively evaluate them across conversations from open and closed issues using logistic regression models. This approach allows us to explore how these three heuristics are associated with successful outcomes, i.e., closed issues.

*1) **Specificity**:* Specificity assesses the degree of detail in a developer's prompts, focusing on how thoroughly technical requirements and specific requests are communicated. High specificity, such as indicating the programming language, library, or version, allows ChatGPT to better understand the problem and produce more accurate, relevant responses. We measure Specificity through three categories: *Technical Keywords*, *Conditional Phrasing*, *Information Emphasis*.

Technical Keywords: Inclusion of specific technical terms or commands can clarify the task and narrow the focus of the response. This could help ChatGPT align its output closely with the developer's needs. We use two metrics: the frequency of software-specific terms, and named entities. We calculate the *#Software-specific Terms* using a pre-compiled list of morphological terms and software-related terms [27], [28]. For *#Named Entities* we use Python's NLTK package [29].

Conditional Phrasing: This metric evaluates how developers structure prompts to clarify task-specific requirements.


Right amount of Context and Specifications
Modifications to Code with Details
Missing Context (E.g., What is the code? What does the library do?)
Still No Context of the Library in Question
Hallucination of Non-Existent Function

Help me design some rust code for no-std that supports the following.
# High level description
Rotations are a key component of attitude and orientation parameters. At first, ANISE only supports [...] The purpose of this issue is to design and implement a _correct_ SO(3) [...]
Some useful resources:
+ Wikipedia on SO(3)(URL)
+ [...]
# Requirements
1. Rotation structures shall be composable (URL)
1. Composition between different representations shall be supported
2. [...]

Certainly! Below is a ... [Code]

...

Right. Let's ignore interpolation using SLERP because [...]

...

[Final Code]

Closed Issue Closed as Completed on Sep 8, 2023

I am writing a Python library that needs to be suspend aware. How can I arrange for my code to receive a notification when it is resumed from a suspended state (e.g. the machine had gone to sleep)?

To make your Python library suspendaware [...]

...

I think it's unreasonable to ask users to install d-bus just for this [...]

Using **timerfd** is a reasonable approach [...]

the Python documentation **does not contain any references to os.timerfd_create though.** Are you sure this really exists?

I apologize for the **confusion** in my previous response. **os.timerfd_create is not a valid function** [...]

...

Open Issue Still Open

Fig. 1: Example of Open vs. Closed Conversations: Closed conversation provides Context and Specifications to ChatGPT vs. the missing Context in open conversation lead ChatGPT to hallucinate.

We calculate the frequency of #Constraints, #Modifiers, and #Subordinate Clauses using Python's NLTK. For instance, *Subordinate Clauses* introduce additional details or conditions that refine the request (e.g., "if the library is compatible with Python 3.8").

Information Emphasis: This metric captures how developers use repetition to maintain focus on key points. To capture this, we measure #Repeated N-grams (n=2, 3) within the prompts to identify instances where developers reinforce important information.

*2) **Contextual Richness**:* We assess how much contextual and background information developers provide in their prompts, as this is crucial for ChatGPT to understand the issue at hand. Capturing contextual richness in the text is challenging since it varies widely depending on the problem [30]. We categorize it into four key categories: *Information Density*, *Code Elements*, *References*, and *Verbosity*.

Information Density: The more unique and concentrated the information, the richer the context [31]. We measure this by calculating #Unique Words in the prompt and the ratio of distinct words to the total word count (#Unique Info).

Code Elements: Concrete artifacts, such as code snippets and error messages, add substantial context. We measure this using the number of #Code Snippets, #Error Message, and Mean Size Code Snippets included in the prompts. To measure #Code descriptions, we tokenize the code identifiers from the code snippets in each conversation and count the sentences mentioning these tokens [31], [32].

References: Referring to APIs or URLs that are relevant to the issue could provide additional context and knowledge to understand the problem. We use regular expressions to count #URLs within the prompt.

Verbosity: Verbosity reflects the extent to which developers elaborate on the problem. Using Python's package spaCy [33], we calculate the total number of prompts in a conversation, and measure the #Words and #Sentences.

*3) **Clarity**:* Unclear prompts are those that are vague, ambiguous, or open to multiple interpretations. Prompts must be clear enough for ChatGPT to accurately interpret them. To assess the clarity of prompts, we use two categories: *Readability* and *Ambiguity*.

Readability: We evaluate how easily the information can be read and understood. First, using Python's pyspellchecker [34], we count the number of misspelled words (#Misspellings). Using Python's spaCy [33], we identify incomplete sentences lacking a subject or object as #Incomplete Sentences [31]. Additionally, we compute two widely recognized readability metrics: the Flesch Reading Ease Score [35] and the SMOG Grade [36]. The Flesch score gauges how easy a text is to read, with higher scores indicating simpler content. The SMOG Grade estimates the years of education needed to understand a text. We calculate both using Python's py-readability-metrics package.

Ambiguity: To assess ambiguity, we analyze the #Unresolved References in prompts, focusing on instances of unclear pronoun usage. Using spaCy's NeuralCoref [33], we identify cases where pronouns lack a clear antecedent. Additionally, we use Natural Language Inference (NLI) to capture deeper contextual confusion. Using the RoBERTa-MNLI model [37], we measure how confidently it classifies relationships between sentences. This model generates an Entailment score (higher means less ambiguous), indicating if the text logically follows the context.

TABLE II: Heuristics to Capture Knowledge Gaps in Prompts and their Range in Conversations of Open and Closed Issues

Knowledge Gaps	Heuristic	Heuristic Categories	Metrics	Open Issues	Closed Issues
				Range (min<median<max)	Range (min<median<max)
Missing Specification	Specificity	Technical Keywords	#Software-specific Terms #Named Entities	0<8<289 0<2<635	0<8<185 0<2<91
		Conditional Phrasing	#Constraints #Modifiers #Subordinate Clauses	0<0<13 0<6<693 0<0<31	0<0<26 0<6<351 0<0<49
		Information Emphasis	#Repeated 2-grams #Repeated 3-grams	0<2<530 0<0<307	0<1<270 0<0<176
Missing Context	Contextual Richness	Code Elements	#Code Snippets Mean Size Code Snippets #Error Message #Code Descriptions	0<0<108 0<0<9257 0<0<47 0<0<669	0<0<77 0<0<3493 0<0<34 0<0<94
		Information Density	First Prompt Length #Unique Info #Unique Words	1<41<2297 2<11<56 3<39<1247	5<40<1727 1<11<96 1<38<598
Multiple Context		References	#URLs	0<0<16	0<0<10
		Verbosity	#Words #Sentences #Total Prompt Count	4<54<4837 1<3<299 1<2<42	1<53<2490 1<3<146 1<2<30
Unclear Instructions	Clarity	Readability	#Misspellings #Incomplete Sentences Flesch Reading Ease Score SMOG Grade	0<1<92 0<0<97 -44.2<66.7<102.6 0<7<17	0<1<14 0<0<50 -155.5<68.3<117.1 0<7<19
		Ambiguity	#Unresolved Reference Entailment	0<3<138 0.001<0.10<0.98	0<3<199 0.002<0.10<0.97

IV. Results and Discussion

A. RQ1: How do prompt knowledge gaps and conversation styles influence the progression and effectiveness of developer-ChatGPT conversations in issue resolution?

Out of 433 conversations, 262 were linked within closed issues while 171 were related to open issues. In total, open issues had 749 prompts, while closed issues had 849 prompts. Although there are more conversations in closed issues, open issues had a higher average number of prompts per conversation. This suggests that conversations in open issues tend to take longer to reach a solution.

Figure 2 shows the frequency of the seven conversation styles across open and closed issues. The predominant styles of conversation in both open and closed issue threads were *Directive Prompting, Chain of Thought, and Responsive Feedback*. Given the large sample size, we applied the independent t-test [38] (p-value<0.05), which is robust to minor deviations from normality. The t-test showed no significant difference in the styles employed across open and closed issues (p-value=0.48), indicating that developers maintain a consistent approach when framing their questions for issue resolution. We also conducted the Shapiro-Wilk test [39] to confirm that the data follows a normal distribution. Additionally, the Mann-Whitney U test [40] (p-value<0.05) showed no significant differences either (p-value=0.84). *Few-shot Learning* style of conversation was only noticed in open issues. This approach involved providing examples for ChatGPT to learn and generate relevant responses. For example, in a conversation about adding JSP support programmatically to the code, the developer provided an example of what a code with this support might look like.

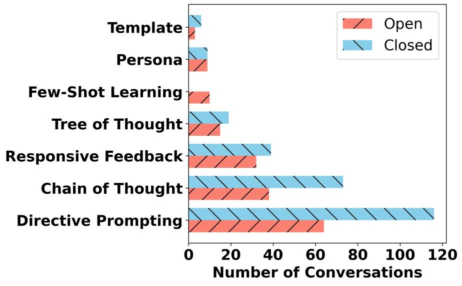


Fig. 2: Styles of Conversations in Closed Vs. Open Issues

We found a significant difference in the number of prompts with knowledge gaps: 410 in open issues compared to only 112 in closed ones. In open issues, the most common gap is *Missing Context* (n=262), followed by *Unclear Instructions* (n=66), *Multiple Context* (n=45), and *Missing Specification* (n=37). In closed issues, 742 of 849 prompts showed no gaps, but *Missing Context* (n=77) remained the most frequent gap.

Providing the right context for an issue is the biggest challenge developers face when interacting with ChatGPT. We observed numerous cases where ChatGPT struggled to grasp the necessary context due to *Missing Context*. For example, Figure 1 shows a conversation within an open issue where a developer asked ChatGPT how to make their Python library suspend-aware. However, they did not provide enough details about the library's functionality, the framework it was built on, and other critical information that was necessary to generate the correct answer. This resulted in ChatGPT hallucinating and using non-existent functions to compensate for the lack of information, leading to unhelpful answers to the developer's questions. Additionally, we observe that *Missing context* is a critical issue in all styles of open-issue conversations, showing that no matter what style developers use, the problem of providing the right context still persists.

Table III shows the number and percentage of gaps per

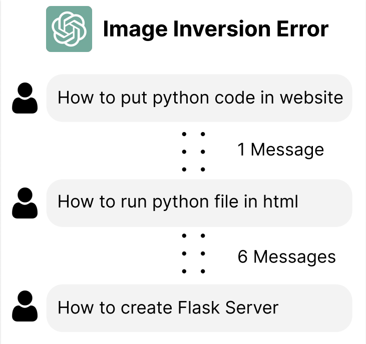


Image Inversion Error

How to put python code in website

1 Message

How to run python file in html

6 Messages

How to create Flask Server

Fig. 3: Multiple Context in a Conversation Linked to an Open Issue

TABLE III: Number and Percentage of Prompts with Knowledge Gaps Per Conversation Style in Open (OP.) and Closed (CL.) Issues

Style of Conversation	**Missing Context**		**Missing Specification**		**Unclear Instructions**		**Multiple Context**	
	OP.	**CL.**	**OP.**	**CL.**	**OP.**	**CL.**	**OP.**	**CL.**
Chain of Thought	**57 (7%)**	26 (3%)	**13 (2%)**	7 (<1%)	**43 (5%)**	3 (<1%)	**43 (6%)**	0
Directive Prompting	**68 (9%)**	29 (3%)	**16 (2%)**	10 (1 %)	**8 (1%)**	5 (<1%)	**2 (<1%)**	1 (<1%)
Responsive Feedback	**91 (12%)**	16 (2%)	**5 (<1%)**	2 (<1%)	**4 (<1%)**	3 (<1%)	0	**2 (<1%)**
Tree of Thought	**39 (5%)**	4 (<1%)	**2 (<1%)**	0	**10 (1%)**	0	0	0
Template	1 (<1%)	1 (<1%)	0	**1 (<1%)**	0	**1 (<1%)**	0	0
Persona	1 (<1%)	1 (<1%)	**1 (<1%)**	0	0	0	0	0
Few-shot Learning	5 (<1%)	-	0	-	1 (<1%)	-	0	-

conversation style for both open and closed issues. Since closed issues have a higher total prompt count, the higher number of knowledge gaps within each category for open issues result in higher percentages. Almost all conversation styles in open issues exhibit at least one type of gap, with *Chain of Thought* showing the highest number of gaps across all identified categories. Conversations adopting Chain of Thought prompting in closed conversations do not exhibit the gap of *Multiple Context* (n=0), indicating that developers focus each discussion with ChatGPT on a single topic. However, in open issues, conversations with Chain of Thought prompting frequently suffer from *Multiple Context* (n=43). In these conversations, developers often discuss their problems step by step, but unexpectedly shift the topic to something unrelated. For instance, as shown in Figure 3, problems such as embedding Python code in a website, running Python within HTML, and creating a Flask server are presented in one conversation thread. Changes in topics within the same conversation thread create confusion and lead to unclear assistance from ChatGPT.

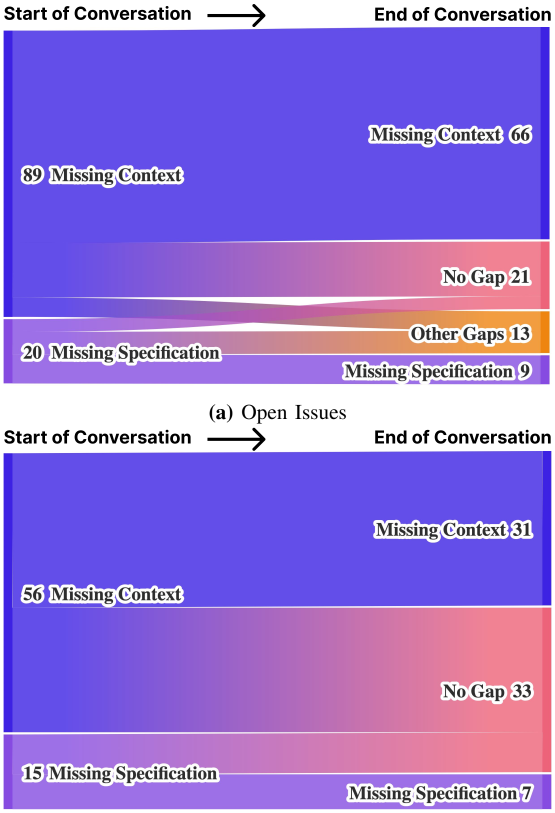
The other two conversation styles that are most influenced by prompt knowledge gaps across open and closed issues are Directive Prompting and Responsive Feedback. Overall, in closed issues, Directive Prompting is the style most affected by knowledge gaps, as it often lacks the context and background information needed to resolve complex issues. One other important difference between open and closed issues is the use of Responsive Feedback. We see noticeably fewer prompts with missing context in closed issues for this style. When providing feedback to ChatGPT, it is important to assess what ChatGPT is struggling with the most, and incorporate more context and information on the problem to get better answers from ChatGPT. For instance, in a conversation linked to a closed issue, a developer was trying to get help with writing code for SQLite database in Python to merge rows from different tables. After the first attempt, the developer was not satisfied with the answer provided by ChatGPT, but received the required answer at the next prompt by providing the schema in more detail: *"[...]This is the table scheme of favorites: CREATE TABLE favorites[...]"*.

Given that knowledge gaps are a problem in both open and closed issues—less in closed compared to open—we also looked into how developers deal with these gaps in their prompts as the conversations progress and how that determines the conversational outcome. We do this analysis for two of the gaps, *Missing context* and *Missing specification*, because these gaps can be addressed with additional rounds of interaction i.e, prompts in the same conversation. Figure 4 shows the results from this analysis. As shown in Figure 4a, out of the 89 open conversations with *Missing Context*, only 17 end with no gaps, while 72 conclude with either missing context or other gaps. In contrast, as shown in Figure 4b, among the 56 closed conversations that contain missing context gaps, 25 conclude with no gaps, suggesting that developers provided the necessary information and details by the end of the conversation. This shows a key difference between closed and open issues: while missing context is an issue in both, developers in closed issues tend to make more effort to provide the necessary context. We also observed the same pattern for *Missing Specifications* in conversations. Out of 15 closed conversations that contain missing specifications (Figure 4b), 8 conversations end with no gaps. For open conversations (Figure 4a), only 4 out of 20 conversations end with no gaps.

Discussion of RQ1 Findings

Style. Across open and closed issues, developers use various conversational styles to resolve issues with ChatGPT. However, prompt knowledge gaps persist across all styles. This indicates that developers face challenges in effectively presenting issues regardless of the chosen approach. In closed issues, Directive Prompting and Chain of Thought are the two most used styles. Chain of Thought allows developers to address gaps progressively by providing additional information, but it remains highly dependent on their ability to identify and articulate missing context both timely and effectively.

Gaps. Missing Context is the most significant gap impacting issue-resolution conversations with ChatGPT. Providing accurate and relevant information is essential to help ChatGPT understand the context; failure to do so often results in misunderstandings or hallucinations. While gaps are found in both open and closed issues, closed ones tend to manage them better through additional explanations and iterative exchanges. This highlights the importance of recognizing and addressing gaps proactively when interacting with LLMs.



(a) Open Issues

(b) Closed Issues

Fig. 4: Progression of Conversations with Prompt Knowledge Gaps

B. RQ2: What heuristics can be used to automatically measure the prompt knowledge gaps?

Table II presents the set of textual and code-related heuristics that we investigate to automatically measure prompt knowledge gaps. Using Logistic Regression, we analyze the association of these heuristics with issue resolution outcomes, where closed issues indicate successful resolution. We chose Logistic Regression for its interpretability and efficiency in modeling binary outcomes, making it suitable for our investigation into the factors that contribute to issue resolution (i.e., open vs. closed issues).

To identify highly associated independent variables, we first calculated the Variance Inflation Factor (VIF) for our heuristics. To ensure the accuracy of our analysis, we eliminated features with a VIF greater than 5 [41]. The features excluded were #Words, #Sentences, #Repeated 2-grams, #Modifiers, #SE Words, #Distinct Words, and #Named Entities. Additionally, to enhance the performance of our regression model, we applied an L1 penalty. Among various configurations and parameters tested, our best-performing regression model uses a Robust Scaler, an L1 penalty, and the liblinear solver with 1000 iterations. The mean Cross-Validated accuracy (CV=5) of our best-performing model is 62%. The coefficient values of our features are presented in Table IV. In our model, the top five features with the highest coefficients are the number of misspellings in the text, the Flesch Reading Ease score, the mean size of code snippets, the number of code snippets, and the entailment of the text. As shown in Table II, we provide a range (min, median, max) for each heuristic to show their variability across conversations. To further examine the statistical significance of these heuristics, we conducted a t-test, finding several metrics to be significant (p-value<0.05): #Named Entities, Mean Size of Code Snippets, #Code Description, #Total Prompt Count, and #Misspellings.

To make our model and the effects of the features more interpretable, and signify how important each heuristic is for issue resolution, we also use SHAP to provide insights into how these features affect the model [42]. The effects of each feature are shown in Figure 5. In this figure, high feature values are shown in red and low values in blue. For instance, lower values of First Prompt Length negatively impact the model (associated with open issues), while higher values positively impact it. High values of #Unresolved References negatively affect the model. The mean size of code snippets has a broader range than other features, making its impact less distinct in the figure due to the current x-axis limit. However, with expanded limits, high mean sizes show a strongly negative impact. Additional figures with varied x-axis limits are included in our replication package. The results of the SHAP in Figure 5 and feature analysis in Table IV are presented below.

Specificity. Effective conversations (i.e., conversations linked to closed issues) are associated with higher Information Emphasis (#Repeated 3-grams) and Conditional Phrasing (#Constraints). Repeated n-grams keeps ChatGPT's responses aligned with the conversation's goal, maintaining the general context. Constraints, reflecting detailed specifications, helps ChatGPT produce responses that are closely tailored to the developer's requests.

Contextual Richness. Providing high number of code snippets (#Code Snippets) while keeping their size small (Mean Size Code Snippets) is associated with effective issue resolution. Large code snippets can challenge ChatGPT's limited context window, leading to less accurate responses. Including more error messages (#Error Messages) also helps ChatGPT understand the problem context better. Additional features that are associated with effective resolution are, including references to external sources (#URLs), unique information (#Unique Info), and longer initial prompts (First Prompt Length). Even if ChatGPT cannot directly access external content, including them provides a clear indication of resources or tools relevant to the issue, which ChatGPT can factor into its response to suggest further actions. Using repeated sets of words to maintain the context of the conversations with ChatGPT while at the same time providing more unique information to keep the conversation going forward is another interesting observation from this analysis.

Clarity. A high number of misspellings (#Misspellings) is strongly associated with unresolved issues, highlighting their negative impact on ChatGPT conversations. High Flesch readability score (Flesch Reading Ease) and textual entailment (Entailment) further highlight the importance of clarity for effective issue resolution. Incomplete sentence structures

TABLE IV: Coefficient Values of Features in Regression Model

Feature Name	Regression Coefficient
#Misspellings	**-0.18062388**
Flesch Reading Ease	**0.06966411**
Mean Size of Code Snippets	**-0.06134951**
#Code Snippets	**0.05318818**
Entailment	**0.03103776**
#Unresolved Reference	-0.02926126
#Constraints	0.02143247
#URLs	0.01547335
First Prompt Length	0.01426245
#Code Descriptions	-0.01209851
#Incomplete Sentences	0.0116349
#Repeated 3-grams	0.00636866
#Unique Info	0.00597188
#Error Messages	0.00035634

(#Incomplete Sentences) were not found to negatively affect conversation outcomes.

Discussion of RQ2 Findings

Our analysis highlights the importance of different heuristics that could be leveraged to automatically improve developer prompts related to issue resolution queries. Effective conversations tend to be *contextually rich*, including relevant code snippets (but avoiding large files), unique details in descriptions, references, and error logs. They also exhibit *high specificity* through the addition of detailed requirements related to the issue. Effective conversations also maintain *clarity* by minimizing misspellings and ambiguity.

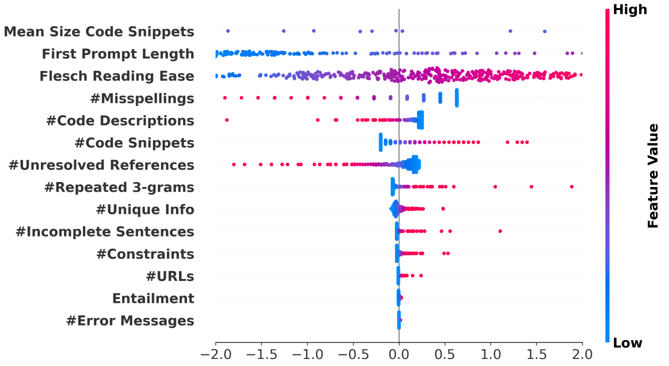


Fig. 5: Impact of Features on Model's Outcome Based on SHAP

V. Feasibility Study

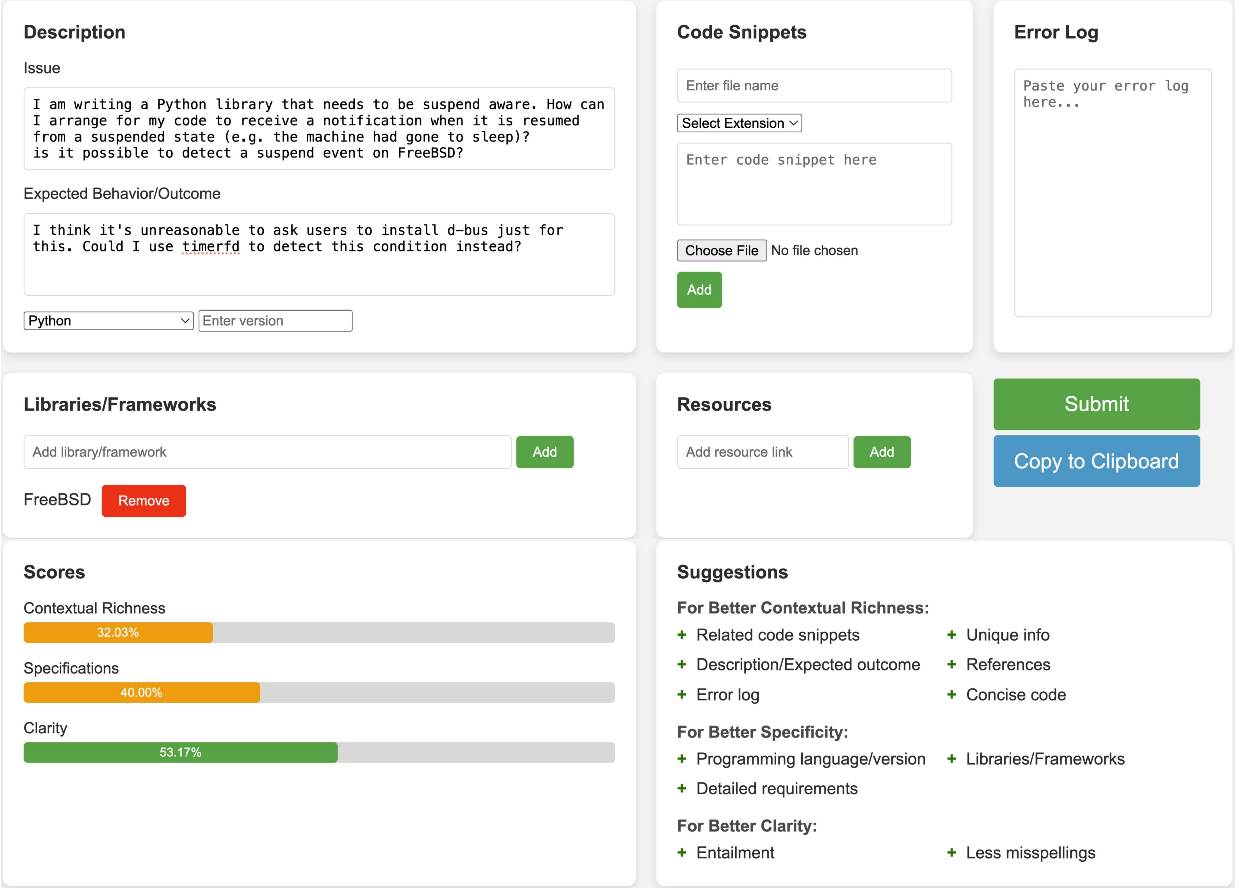
To explore whether the heuristics from RQ2 can be leveraged to develop a tool for detecting prompt knowledge gaps, we conduct a feasibility study. Our goal is to develop a lightweight tool that could be easily adapted into the developer workflows for issue resolution. Therefore, we develop a browser extension that can help users create tailored, detailed prompts for issue resolution. The frontend is developed using HTML, CSS, and JavaScript, with Flask powering the backend that uses our logistic regression models to evaluate the prompts to see what is missing from the prompt. (more details are provided in the replication package). A snapshot of our tool is shown in Figure 6. Users can enter the issue details in the **Description** field, including the expected outcome, programming language, and version. Code snippets can be added by uploading files or pasting them into the **Code Snippets** box. Error logs and stack traces can be entered under **Error Log**, relevant libraries or frameworks in **Libraries/Frameworks**, and additional resources to aid context in the **Resources** box.

Our tool offers developers a structured template designed to capture critical information essential for effective issue resolution, minimizing the risk of commonly observed knowledge gaps like Missing Context, Missing Specifications, or Unclear Instructions. Once the template is completed, the tool automatically evaluates the input against predefined heuristics, generating a score for each. Based on these scores, developers can iteratively refine their inputs to achieve higher scores. Once satisfied, they can copy the optimized, structured prompt for use in their issue-resolution conversations with LLMs. To illustrate an example, we present the tool's performance using the open and closed conversations shown in Figure 1. We extract the information from each conversation, populate the corresponding fields in the UI, and run the tool to display individual analyses for each case. The generated scores are the mean average of the features included in each heuristic.

Closed Conversation. In this conversation, the user provides a detailed issue description, including specific requirements, a code snippet, and relevant online resources, along with libraries that the response should incorporate. After running the tool, it scores 54.07% for contextual richness, 80% for specificity, and 65.86% for clarity. These scores suggest that the prompt is likely to have effective results but still has room for improvement. The analysis indicates that increasing the number of code snippets, and unique information inside the description can improve its context score, while improving the entailment in the description by having sentences that logically follow the same structure could enhance its clarity. Additionally, there are six misspellings that impact the clarity of the description.

Open Conversation. In this conversation (as shown in Figure 6), the user provides only a brief description of the issue, along with the programming language and framework. The tool's analysis shows lower scores: 32.03% for contextual richness, 40% for specificity, and 53.17% for clarity. The tool correctly identifies the gaps in the contextual richness of the prompt and low specifications in the requirements. Although clarity is generally good, adding relevant code snippets, resources, and a more detailed description all are required for this prompt to be more effective. These suggestions, along with the scores, are provided by the tool to guide targeted improvements.

This tool showcases the potential of leveraging heuristics to automatically identify prompt knowledge gaps in issue-resolution conversations, empowering developers to proactively enhance prompt quality. As the first step toward automating gap detection in LLM-mediated interactions, this tool lays the groundwork for advancing the quality of such conversations. However, its current capabilities are limited to predefined heuristics and may not capture all nuances of real-world developer interactions. In addition, it needs more user feedback studies to fully demonstrate its utility. Future enhancements, including additional features and more refined metrics, could further improve its effectiveness. To encourage



Issue

```
I am writing a Python library that needs to be suspend aware. How can
I arrange for my code to receive a notification when it is resumed
from a suspended state (e.g. the machine had gone to sleep)?
is it possible to detect a suspend event on FreeBSD?
```

Expected Behavior/Outcome

```
I think it's unreasonable to ask users to install d-bus just for
this. Could I use timerfd to detect this condition instead?
```

Python

FreeBSD

Scores

Contextual Richness 32.03%

Specifications 40.00%

Clarity 53.17%

Suggestions

For Better Contextual Richness:

- + Related code snippets
- + Description/Expected outcome
- + Error log
- + Unique info
- + References
- + Concise code

For Better Specificity:

- + Programming language/version
- + Detailed requirements
- + Libraries/Frameworks

For Better Clarity:

- + Entailment
- + Less misspellings

Fig. 6: Tool for Automatic Prompt Knowledge Gap Detection

adoption and refinement, we have made the code and usage instructions available in our replication package [43].

VI. Threats to Validity

Construct Validity. To reduce subjectivity in our annotations, we conducted multiple rounds of coding and discussions to resolve conflicts and ensure consistency. The authors performing the analysis each have over three years of experience in programming and qualitative analysis. The final average Cohen's Kappa agreement between them was 78%, indicating strong inter-rater reliability.

Internal Validity. The heuristics chosen for this study are based on a thorough qualitative analysis of content shared with ChatGPT for issue resolution. While these heuristics capture critical aspects, they may not encompass every nuance of the conversations. To mitigate this, we included a wide range of heuristics and removed highly correlated features using the Variance Inflation Factor (VIF) to reduce potential biases. We also conducted sanity checks on all automated measures to ensure accuracy and prevent script errors. In our analysis, we assume that conversations in closed issues contributed to their resolution, while those in open issues did not. This assumption is reasonable, as resolved issues indicate productive exchanges. However, there might be instances where closed issues were resolved using additional help, and were not entirely based on the conversation with ChatGPT. In addition, we do not consider the difficulty of issues in our analysis. We acknowledge that further exploration of how issue complexity affects issue resolution outcomes would be valuable, and leave this as a direction for future work.

External Validity. Our findings are based on a dataset of ChatGPT conversations shared within GitHub issue threads, which may limit generalizability to other, unshared ChatGPT conversations or interactions with different LLMs, such as Gemini. We focused on ChatGPT due to its popularity and broad usage among developers. However, it is possible that a larger, more comprehensive dataset of GitHub developer-ChatGPT interactions could reveal different patterns.

VII. Related Work

LLMs for Issue Resolution. LLMs are now widely applied for bug resolution and issue tracking [2], [3], [44]–[46]. Research indicates that developers frequently engage with ChatGPT to describe bug symptoms and seek potential solutions [47]. While Q&A platforms like Stack Overflow have traditionally played a significant role in helping developers resolve issues, their traffic has declined with the rise of LLMs [48]. Studies of Q&A forum responses and LLM-generated answers reveal that LLMs struggle with inquiries related to certain frameworks and libraries [48]. For instance, ChatGPT's accuracy for security-related questions was only 56% [49]. Users often prefer ChatGPT for its well-articulated language

[50], however, ChatGPT responses are frequently of lower quality than those on Stack Overflow, often lacking relevance [51]. In addition, answers available on Stack Overflow prove to be more effective in addressing debugging tasks [52]. Advancements in LLMs have also initiated the development of automated tools for issue resolution using LLMs [53]–[55]. By compiling a dataset of GitHub issues alongside their corresponding test cases [56], researchers have evaluated various LLMs' capabilities in understanding issues and generating correct patches. For example, Tao et al. [57] introduced an LLM-powered multi-agent framework that achieved a resolution rate of 13.94%. Subsequent studies that involved interactive sessions with LLMs to identify problematic files and relevant contextual information increased the resolution rate to 22% [58].

Prompt Analysis. Recent studies have also focused on studying interactions between software developers and ChatGPT. The most frequent inquiries directed at ChatGPT include code generation, conceptual clarifications, and how-to questions [12]. The predominant topics identified by developers include advanced programming guidance, information-seeking about frameworks, and high-level design recommendations [16], [17]. Developers often engage in multi-turn conversations to enhance the quality of responses by asking follow-up questions or refining their prompts [12]. LLM-generated code are mostly used to illustrate high-level concepts or provide examples for documentation purposes. Most conversations revolve around requests for improvements and additional explanations within the generated code [59]. Additionally, Champa et al. [60] found that developers primarily seek assistance from ChatGPT for Python code related to quality management and issue resolution tasks. Developer-ChatGPT interactions have been shown to be particularly effective for software development management, optimization, and new feature implementation [60].

Mondal et al. identified 11 factors contributing to prolonged conversations with ChatGPT, with missing specifications and requests for additional functionality being the most common issues [7]. While frameworks have been developed to structure prompts in various styles and techniques for improved LLM responses [13], [14], [21], recent studies suggest that prompt engineering is often unpredictable and unreliable, emphasizing instead the importance of clearer articulation of requests [10], [13]. Our paper contributes to this research by analyzing prompt knowledge gaps across conversational styles, providing heuristics to automatically detect and address these gaps to enhance issue resolution outcomes with LLMs.

VIII. Conclusion and Future Work

LLMs have shown potential for issue resolution, but there is often a disconnect between developers' expectations and the responses they receive. This disconnect typically arises from how issues are presented to LLMs, with insufficient context, specifications, or clarity. While frameworks and prompt-engineering methods attempt to refine LLM outputs, they remain unpredictable and largely reliant on a "trial and error" approach. Therefore, we focus on identifying and mitigating knowledge gaps in prompts, addressing the need to help developers with targeted suggestions.

Our analysis reveals that developers employ a range of conversational styles in issue resolution with ChatGPT, with Directive Prompting, Chain of Thought, and Responsive Feedback being the most prevalent. The most common knowledge gaps in open issues were Missing Context, Unclear Instructions, and Multiple Contexts. For closed issues, while 86.8% of prompts contained no gaps, Missing Context remained the most frequent gap. Providing sufficient context is essential for effective resolutions, yet developers often struggle to do so effectively. Our identified heuristics also suggest that effective conversations are contextually rich, containing related code snippets, unique information, error messages, external references, and longer initial prompts. Effective prompts also include specific requirements and technical details, as well as clear, logically structured sentences with less ambiguity.

Using these heuristics, we developed a lightweight tool to detect knowledge gaps in prompts and offer templates that guide developers in crafting contextually rich, specific, and clear prompts, enabling improved LLM-driven issue resolution. Initial design demonstrates the feasibility of such a tool, though further refinement is needed to enhance accuracy. In the future, we plan to evaluate the tool through developer feedback and questionnaires, and to experiment with additional heuristics to better capture and address prompt knowledge gaps.

References

[1] "The state of developer ecosystem 2023." [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2023/

[2] 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," 2024.

[3] Y. Wu, Z. Li, J. M. Zhang, M. Papadakis, M. Harman, and Y. Liu, "Large language models in fault localisation," 2023.

[4] "How conversational programming will democratize computing." [Online]. Available: https://thenewstack.io/how-conversational-programming-will-democratize-computing/

[5] S. I. Ross, F. Martinez, S. Houde, M. Muller, and J. D. Weisz, "The programmer's assistant: Conversational interaction with a large language model for software development," in *Proceedings of the 28th International Conference on Intelligent User Interfaces*, ser. IUI '23. ACM, Mar. 2023.

[6] J. Li, E. D. Mynatt, V. Mishra, and J. Bell, ""always nice and confident, sometimes wrong": Developer's experiences engaging generative ai chatbots versus human-powered q&a platforms," *ArXiv*, vol. abs/2309.13684, 2023.

[7] S. Mondal, S. D. Bappon, and C. K. Roy, "Enhancing user interaction in chatgpt: Characterizing and consolidating multiple prompts for issue resolution," 2024.

[8] X. Zhou, P. Liang, B. Zhang, Z. Li, A. Ahmad, M. Shahin, and M. Waseem, "Exploring the problems, their causes and solutions of ai pair programming: A study with practitioners of github copilot," 2024.

[9] 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," 2024.

[10] R. Battle and T. Gollapudi, "The unreasonable effectiveness of eccentric automatic prompts," 2024. [Online]. Available: https://arxiv.org/abs/2402.10949

[11] A. Khurana, H. Subramonyam, and P. K. Chilana, "Why and when llm-based assistants can go wrong: Investigating the effectiveness of prompt-based interactions for software help-seeking," in *Proceedings of the 29th International Conference on Intelligent User Interfaces*, ser. IUI '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 288–303. [Online]. Available: https://doi.org/10.1145/3640543.3645200

[12] H. Hao, K. A. Hasan, H. Qin, M. Macedo, Y. Tian, S. H. H. Ding, and A. E. Hassan, "An empirical study on developers shared conversations with chatgpt in github pull requests and issues," 2024.

[13] Q. Ma, W. Peng, H. Shen, K. Koedinger, and T. Wu, "What you say = what you want? teaching humans to articulate requirements for llms," 2024. [Online]. Available: https://arxiv.org/abs/2409.08775

[14] J. Kim, S. Park, K. Jeong, S. Lee, S. H. Han, J. Lee, and P. Kang, "Which is better? exploring prompting strategy for llm-based metrics," 2023. [Online]. Available: https://arxiv.org/abs/2311.03754

[15] T. Xiao, C. Treude, H. Hata, and K. Matsumoto, "DevGPT: Studying Developer-ChatGPT Conversations," Feb. 2024, arXiv:2309.03914 [cs]. [Online]. Available: http://arxiv.org/abs/2309.03914

[16] S. Mohamed, A. Parvin, and E. Parra, "Chatting with ai: Deciphering developer conversations with chatgpt," 2024.

[17] M. R. I. Ertugrul Sagdic, Arda Bayram, "On the taxonomy of developers' discussion topics with chatgpt," 2024.

[18] P. M. Stahl, "pemistahl/lingua-py," 2024. [Online]. Available: https://github.com/pemistahl/lingua-py

[19] M. Oedingen, R. C. Engelhardt, R. Denz, M. Hammer, and W. Konen, "Chatgpt code detection: Techniques for uncovering the source of code," *AI*, vol. 5, no. 3, 2024. [Online]. Available: http://dx.doi.org/10.3390/ai5030053

[20] T. Azungah and R. Kasmad, "Qualitative research: deductive and inductive approaches to data analysis," 08 2020.

[21] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," 2023. [Online]. Available: https://arxiv.org/abs/2302.11382

[22] O. Fagbohun, R. M. Harrison, and A. Dereventsov, "An empirical categorization of prompting techniques for large language models: A practitioner's guide," 2024. [Online]. Available: https://arxiv.org/abs/2402.14837

[23] "Prompt Engineering Guide – Nextra," Sep. 2024. [Online]. Available: https://www.promptingguide.ai/

[24] M. McHugh, "Interrater reliability: The kappa statistic," *Biochemia medica : časopis Hrvatskoga društva medicinskih biokemičara / HDMB*, vol. 22, pp. 276–82, 10 2012.

[25] J. Corbin and A. Strauss, *Basics of Qualitative Research (3rd ed.): Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, Inc., 2008.

[26] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," 2022. [Online]. Available: https://arxiv.org/abs/2203.02155

[27] C. Chen, Z. Xing, and X. Wang, "Unsupervised software-specific morphological forms inference from informal discussions," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 450–461.

[28] "Dictionary of Software Terms." [Online]. Available: https://techterms.com/category/software

[29] "NLTK : Natural Language Toolkit." [Online]. Available: https://www.nltk.org/

[30] G. Melo, E. Law, P. Alencar, and D. Cowan, "Exploring context-aware conversational agents in software development," 2020. [Online]. Available: https://arxiv.org/abs/2006.02370

[31] P. Chatterjee, K. Damevski, N. A. Kraft, and L. Pollock, "Automatically identifying the quality of developer chats for post hoc use," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 4, jul 2021. [Online]. Available: https://doi.org/10.1145/3450503

[32] P. Chatterjee, B. Gause, H. Hedinger, and L. Pollock, "Extracting code segments and their descriptions from research articles," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 91–101.

[33] "spacy · PyPI." [Online]. Available: https://pypi.org/project/spacy/

[34] T. Barrus, "pyspellchecker: Pure python spell checker based on work by Peter Norvig."

[35] "Flesch Reading Ease and the Flesch Kincaid Grade Level." [Online]. Available: https://readable.com/readability/flesch-reading-ease-flesch-kincaid-grade-level/

[36] B. Scott, "The SMOG Readability Formula, a Simple Measure of Gobbledygook," Oct. 2023, section: Readability Formulas Help. [Online]. Available: https://readabilityformulas.com/the-smog-readability-formula/

[37] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 2019.

[38] T. K. Kim, "T test as a parametric statistic," *Korean Journal of Anesthesiology*, vol. 68, no. 6, p. 540, Nov. 2015. [Online]. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC4667138/

[39] A. Ghasemi and S. Zahediasl, "Normality Tests for Statistical Analysis: A Guide for Non-Statisticians," *International Journal of Endocrinology and Metabolism*, vol. 10, no. 2, pp. 486–489, 2012. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3693611/

[40] N. Nachar, "The mann-whitney u: A test for assessing whether two independent samples come from the same distribution," *Tutorials in Quantitative Methods for Psychology*, vol. 4, 03 2008.

[41] S. S. Habshah Midi and S. Rana, "Collinearity diagnostics of binary logistic regression model," *Journal of Interdisciplinary Mathematics*, vol. 13, no. 3, pp. 253–267, 2010. [Online]. Available: https://doi.org/10.1080/09720502.2010.10700699

[42] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 4768–4777.

[43] "Anonymized repository." [Online]. Available: https://anonymous.4open.science/r/prompt-knowledge-gap-BE45/

[44] N. Tang, M. Chen, Z. Ning, A. Bansal, Y. Huang, C. McMillan, and T. J.-J. Li, "An empirical study of developer behaviors for validating and repairing ai-generated code." Plateau Workshop.

[45] 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," vol. 1, no. FSE, 2024. [Online]. Available: https://doi.org/10.1145/3660773

[46] B. Yang, H. Tian, W. Pian, H. Yu, H. Wang, J. Klein, T. F. Bissyandé, and S. Jin, "Cref: An llm-based conversational software repair framework for programming tutors," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 882–894. [Online]. Available: https://doi.org/10.1145/3650212.3680328

[47] J. K. Das, S. Mondal, and C. K. Roy, "Investigating the utility of chatgpt in the issue tracking system: An exploratory study," 2024.

[48] L. D. Silva, J. Samhi, and F. Khomh, "Chatgpt vs llama: Impact, reliability, and challenges in stack overflow discussions," 2024.

[49] Z. Delile, S. Radel, J. Godinez, G. Engstrom, T. Brucker, K. Young, and S. Ghanavati, "Evaluating privacy questions from stack overflow: Can chatgpt compete?" in *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*, 2023, pp. 239–244.

[50] S. Kabir, D. N. Udo-Imeh, B. Kou, and T. Zhang, "Is stack overflow obsolete? an empirical study of the characteristics of chatgpt answers to stack overflow questions," 2024.

[51] B. Xu, T.-D. Nguyen, T. Le-Cong, T. Hoang, J. Liu, K. Kim, C. Gong, C. Niu, C. Wang, B. Le, and D. Lo, "Are we ready to embrace generative ai for software q&a?" in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1713–1717.

[52] J. Liu, X. Tang, L. Li, P. Chen, and Y. Liu, "Which is a better programming assistant? a comparative study between chatgpt and stack overflow," 2023.

[53] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 1482–1494. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00129

[54] T. D. Viet and K. Markov, "Using large language models for bug localization and fixing," in *2023 12th International Conference on Awareness Science and Technology (iCAST)*, 2023, pp. 192–197.

[55] J. Zhao, D. Yang, L. Zhang, X. Lian, Z. Yang, and F. Liu, "Enhancing automated program repair with solution design," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association

for Computing Machinery, 2024, p. 1706–1718. [Online]. Available: https://doi.org/10.1145/3691620.3695537

[56] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" 2024. [Online]. Available: https://arxiv.org/abs/2310.06770

[57] W. Tao, Y. Zhou, Y. Wang, W. Zhang, H. Zhang, and Y. Cheng, "Magis: Llm-based multi-agent framework for github issue resolution," 2024. [Online]. Available: https://arxiv.org/abs/2403.17927

[58] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1592–1604. [Online]. Available: https://doi.org/10.1145/3650212.3680384

[59] 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," Mar. 2024, arXiv:2402.11702 [cs]. [Online]. Available: http://arxiv.org/abs/2402.11702

[60] A. I. Champa, M. F. Rabbi, C. Nachuma, and M. F. Zibran, "Chatgpt in action: Analyzing its use in software development," 2024.