# ChatGPT Chats Decoded: Uncovering Prompt Patterns for Superior Solutions in Software Development Lifecycle

Liangxuan Wu
liangxuanwu@hust.edu.cn
Huazhong University of Science and Technology, China

Yanjie Zhao
carolzhao233@gmail.com
Huazhong University of Science and Technology, China

Xinyi Hou
xinyihou@hust.edu.cn
Huazhong University of Science and Technology, China

Tianming Liu
Tianming.Liu@monash.edu
Monash University
Melbourne, Australia

Haoyu Wang*
haoyuwang@hust.edu.cn
Huazhong University of Science and Technology, China

## ABSTRACT

The advent of Large Language Models (LLMs) like ChatGPT has markedly transformed software development, aiding tasks from code generation to issue resolution with their human-like text generation. Nevertheless, the effectiveness of these models greatly depends on the nature of the prompts given by developers. Therefore, this study delves into the DevGPT dataset, a rich collection of developer-ChatGPT dialogues, to unearth the patterns in prompts that lead to effective problem resolutions. The underlying motivation for this research is to enhance the collaboration between human developers and AI tools, thereby improving productivity and problem-solving efficacy in software development. Utilizing a combination of textual analysis and data-driven approaches, this paper seeks to identify the attributes of prompts that are associated with successful interactions, providing crucial insights for the strategic employment of ChatGPT in software engineering environments.

## CCS CONCEPTS

• **Information systems** → **Data mining**.

## KEYWORDS

Data mining, Large language model, LLM, ChatGPT

**ACM Reference Format:**
Liangxuan Wu, Yanjie Zhao, Xinyi Hou, Tianming Liu, and Haoyu Wang. 2024. ChatGPT Chats Decoded: Uncovering Prompt Patterns for Superior Solutions in Software Development Lifecycle. In *21st International Conference on Mining Software Repositories (MSR '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3643991.3645069

*Haoyu Wang is the corresponding author.

## 1 INTRODUCTION

The introduction of Large Language Models (LLMs) in the ever-changing environment of software engineering has signaled a fundamental shift in how development processes are addressed. These models have enriched automated developer support by encompassing a spectrum of functionalities, from code generation to enhanced problem-solving capabilities. The increasing reliance on these LLM-driven tools signifies a major transformation in the software development paradigm [4].

Despite their potential, the effectiveness of LLMs like ChatGPT in software development is not intrinsic but contingent upon the nature of interaction, specifically the prompts given by developers. The quality and structure of these prompts play a crucial role in determining the utility and accuracy of the solutions provided by ChatGPT. This aspect of human-AI interaction remains underexplored, **with a lack of data-driven guidelines or patterns for optimal use in software development contexts**. Therefore, this study aims to bridge this gap by mining the DevGPT dataset [13], a comprehensive collection of developer interactions with ChatGPT. The primary goal is to unravel the underlying prompt patterns that lead to successful problem resolutions, addressing three key research questions (RQs):

- **RQ1:** What's the usual conversation structure between developers and ChatGPT, and how many turns, on average, does it take to reach a conclusion?
- **RQ2:** What are the most efficient prompt patterns for eliciting high-quality responses from ChatGPT in software development contexts?
- **RQ3:** How can the identified prompt patterns be classified and optimized in relation to different activities of the software development lifecycle?

Understanding these patterns has far-reaching consequences for software engineering practice. By elucidating the characteristics of effective prompts, this research offers practical insights for developers to enhance their interaction with ChatGPT, thereby optimizing problem-solving and productivity in software development.

## 2 BACKGROUND

### 2.1 Prompt Patterns

In the context of developer interactions with LLMs, prompt patterns [2] are fundamental structures that guide the flow and effectiveness of these dialogues. These patterns serve as frameworks for formulating queries in a manner that enhances the relevance and usefulness of the responses generated by the model. As illustrated in Table 1 and following the classification system proposed by White et al. [11], we classify high-quality prompt patterns into six distinct categories, each exemplified by several instances. We will now delve into each category in detail.

**Table 1: The categorization of prompt patterns.**

| Category | Prompt Patterns |
|---|---|
| Input Semantics | Meta Language Creation |
| Output Customization | Output Automater, Persona, Visualization Generator, Recipe, Template |
| Error Identification | Fact Check List, Reflection |
| Prompt Improvement | Question Refinement, Alternative Approaches, Cognitive Verifier, Refusal Breaker |
| Interaction | Flipped Interaction, Game Play, Infinite Generation |
| Context Control | Context Manager |

**Input Semantics.** The input semantics category primarily involves the *meta language creation* pattern, where users construct prompts like "Whenever I say A, do B" to guide LLMs. While this helps in contextual understanding, it risks semantic ambiguity and potential misinterpretation of instructions by LLMs.

**Output Customization.** This category shapes the output of LLMs by specifying types, formats, and structures. The *output automater* pattern instructs the model to perform suggested steps, reducing repetitive user prompts. The *persona* pattern guides the model to adopt a specific role or character for problem-solving. The *visualization generator* pattern focuses on creating visual outputs for tools like Graphviz Dot [3]. In the *recipe* pattern, users set constraints, and LLMs provide a step-by-step guide or action recommendations. Lastly, the *template* pattern uses a fixed template for LLMs' responses, directing it to fill specific positions for a structured effect.

**Error Identification.** This category focuses on LLMs addressing errors in their outputs, encompassing two main patterns. The *fact check list* pattern is used to mitigate LLMs' tendency to produce off-target or fabricated outputs by guiding it to validate data for authenticity. Alternatively, for theoretical deduction issues without factual checks, prompts may request detailed explanations from LLMs on their conclusions, enabling users to assess potential errors in their reasoning. This approach is classified as *reflection* pattern.

**Prompt Improvement.** This category of patterns aims to enhance prompt quality, thus improving the quality of LLMs' outputs. The *question refinement* pattern encourages LLMs to generate better, more comprehensive prompts, potentially boosting LLMs' task performance. The *alternative approaches* pattern allows LLMs to solve problems using varied methods, not just those directly stated in the prompt, which could lead to more efficient or irrelevant answers. The *cognitive verifier* pattern involves breaking down a question into sub-questions and combining responses for more precise answers [14]. Moreover, the *refusal breaker* pattern involves reformulating a question to elicit a response from LLMs when initially met with refusal.

**Interaction.** This category enhances interactivity with LLMs. The *flipped interaction* pattern allows LLMs to ask questions during problem-solving. The *game play* pattern creates a gaming atmosphere for LLM responses. For repetitive tasks, the *infinite generation* pattern enables LLMs to continue working until stopped, reducing the need for repeated prompts.

**Context Control.** This category addresses the issue of LLMs including irrelevant details or overemphasizing unwanted aspects in responses. The *context manager* pattern guides LLMs to focus on specific areas, using directives like "only consider", "ignore", etc., to refine their response scope.

## 2.2 Software Development Lifecycle

According to Hou et al. [4], the software development lifecycle includes six activities: requirements engineering, software design, software development, software quality assurance, software maintenance, and software management.

**Requirements engineering** involves defining and managing system requirements through gathering, analysis, specification, and validation. It forms the foundation for the software system's functionality and performance. **Software design** covers GUI retrieval, specification synthesis, and rapid prototyping. This stage designs the system structure and components, focusing on user experience, detailed system specifications, and validating design concepts. **Software development** encompasses code generation, agile estimation, code completion, API documentation, and optimization. It aims to enhance efficiency, code quality, and developer experience with various technologies and tools. **Software quality assurance** ensures software quality and stability, involving test generation, bug localization, vulnerability detection, and other methods to verify software functionality and performance. **Software maintenance** includes bug fixing, code reviews, debugging, and responding to user feedback. It focuses on enhancing software stability, optimizing performance, and maintaining long-term viability. **Software management** involves effort estimation to determine the necessary human, time, and resource efforts for a project, aiding in effective planning and resource allocation.

## 3 METHODOLOGY

Our methodology can be divided into three modules: the Prompt Clustering Module (PCM), the Response Analysis Module (RAM), and the Lifecycle Categorization Module (LCM), as illustrated in Figure 1. Utilizing the capabilities of three modules, we aim to extract key conversations from the DevGPT dataset and analyze them in the software development lifecycle context. Our focus is on understanding how prompt patterns affect ChatGPT's response effectiveness, providing insights into the relationship between prompt structure and model performance in software engineering.

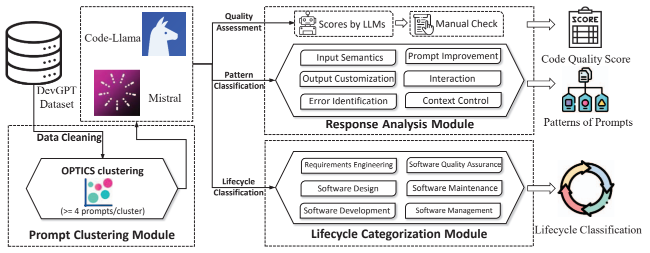

**Figure 1: The overview of our methodology.**

## 3.1 PCM: Prompt Clustering Module

This module focuses on extracting prompts from conversations and clustering them based on similar topics, resulting in the formation of distinct topic-focused clusters. In this module, the input is the raw data, and the output is the clustering of prompts from all conversations processed using the `OPTICS` clustering method [1]. This facilitates our subsequent data analysis.

**Step 1: Data Cleaning.** Per DevGPT [13], each snapshot updates the previous, leading to repeated conversations across snapshots. We streamlined and deduplicated these across all snapshots, keeping only the latest version of each segment to aid in upcoming clustering operations. From the 17,908 prompts in the `20230831` snapshot, we meticulously cleaned and retained 7,832 prompts, ensuring that in cases of partial duplication, only the most comprehensive, or longest, conversations were preserved.

**Step 2: `OPTICS` clustering.** After data cleaning, we aim to categorize and discuss the most common topics among these non-duplicate data points. To achieve this, we have employed the `OPTICS` clustering algorithm [1] to process this subset of data. The `OPTICS` (Ordering Points To Identify the Clustering Structure) algorithm, effective for large datasets, identifies density-based clusters without predefining cluster counts or density thresholds, unlike K-means or DBSCAN. This approach, focusing on density connectivity of points, is particularly suited to our study where cluster numbers aren't predetermined, making `OPTICS` an optimal choice for our clustering task.

By iteratively adjusting the parameters of the `OPTICS` clustering algorithm, we can record clusters for all prompts. Through sorting, we identify the top 50 clusters for focused analysis and determine their cluster centers. And every cluster has 4 samples at least.

## 3.2 RAM: Response Analysis Module

In §3.1, our study focuses on the top 50 largest clusters for pattern recognition and quality assessment. We analyze conversations from these clusters, assigning a quality rating to the responses generated by prompts in each cluster.

**Step 1: Pattern Categorization.** To investigate the influence of diverse questioning patterns on the effectiveness of prompts in yielding anticipated outcomes, we classified the prompts within each selected cluster according to the categories of patterns outlined in §2.1 or identified them as having no specific pattern. Acknowledging the potential for omissions and cognitive biases inherent in human classification, we initiated a collaborative effort with LLMs to enhance the accuracy and objectivity of our analysis. ChatGPT is employed to assist in validating the prompt patterns identified within the clustering outcomes. Each prompt undergoes a meticulous manual examination to ascertain its adherence to the identified patterns. This process culminates in the confirmation of the categorization of all prompts, as illustrated in Table 1.

**Step 2: Quality Assessment.** After pattern identification, the next step involves scoring the quality of the answers. Recall that the snapshot in DevGPT [13] is composed of a dataset generated through conversations with ChatGPT. To ensure fair evaluation of code quality in ChatGPT responses, we have established specific criteria. Additionally, we've implemented a "peer review" process using other LLMs, apart from ChatGPT, to maintain an objective assessment standard. Also, considering that 40.34% of the conversations involve code generation, we leverage Code-Llama-34B [8], which excels in providing services for extended conversations and code discussions. With the assessment from Code-Llama, we can effectively score the code generated by ChatGPT and its corresponding answers. Nevertheless, assessing with a single model can lead to misjudgments and occasional situations. We observed that Mistral-7B [6], stands as the best-performing LLM for its size to date. Therefore, we used Mistral-7B for a second round of evaluation to ensure the accuracy of judgments for Code-Llama-34B.

Our emphasis was particularly on conversations involving code generation. For each answer, we generated a percentage-based score using Code-Llama and Mistral. Additionally, each code snippet received a separate percentage-based score. We compiled the score distribution, as shown in Table 2, to facilitate further analysis. The basis for our scoring has three main points:

- We established various code quality assessment criteria according to Stamelos et al. [10] and constructed corresponding prompts for Code-Llama and Mistral. For responses containing code, we use eight parameters: Cyclomatic Complexity, Maximum Levels, Number of Paths, Unconditional Jumps, Comment Frequency, Program Length, Average Size, and Number of Inputs/Outputs. These were ultimately used to assess the quality of the code itself.
- For responses not containing code, we utilize Code-Llama and Mistral to analyze intent. The original prompt is then reviewed to determine if the response fulfills its requirements. A manual static analysis phase is included for precise assessment of reasonableness.
- Regarding readability and audience-oriented rationality, parameters such as Cyclomatic Complexity are considered. This stage involves a subjective manual evaluation of each answer's readability.

## 3.3 LCM: Lifecycle Categorization Module

In the second module, we categorized prompts from various conversations into different pattern types. This module will utilize the software development lifecycle [4] to classify different conversations, aiming to discuss the practical implications of each prompt pattern in real-world scenarios. This approach allows us to delve into the contextual relevance and application of these patterns, providing insights into their effectiveness throughout the software development process.

We start by using LLMs (i.e., Code-Llama and Mistral) together with human efforts to classify each cluster. Following that, we perform an analysis to derive meaningful insights from the data related to the six activities of the software development lifecycle as mentioned in §2.2. Concurrently, we conduct a cross-sectional comparison to examine whether the insights drawn from patterns are universally applicable across various lifecycles.

Finally, catering to the current trend of ChatGPT, we aim to summarize methods for constructing prompts through data and example comparisons. Our goal is to facilitate users of ChatGPT (or other LLMs) in effortlessly obtaining the results they need.

## 4 RESULTS AND DISCUSSION

### 4.1 Answer to RQ1

In developer-ChatGPT interactions, developers often ask questions and refine prompts to improve ChatGPT's responses. To ascertain the average number of turns per conversation, we calculate it by dividing the aggregate number of prompts (17,908) by the total count

**Table 2: Results of code quality assessment.**

| Score | Input Semantics | | Output Customization | | Error Identification | | Prompt Improvement | | Interaction | | Context Control | | No pattern | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Code-Llama | Mistral | Code-Llama | Mistral | Code-Llama | Mistral | Code-Llama | Mistral | Code-Llama | Mistral | Code-Llama | Mistral | Code-Llama | Mistral |
| 100 | 0 | 1 | 2 | 31 | 0 | 5 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 4 |
| [95,100) | 26 | 12 | 97 | 68 | 31 | 14 | 28 | 9 | 29 | 13 | 24 | 8 | 2 | 0 |
| [90,95) | 27 | 40 | 95 | 102 | 41 | 49 | 34 | 44 | 35 | 46 | 27 | 43 | 11 | 12 |
| [85,90) | 1 | 1 | 53 | 20 | 24 | 18 | 13 | 12 | 7 | 7 | 9 | 3 | 18 | 3 |
| [80,85) | 1 | 1 | 14 | 32 | 19 | 28 | 12 | 23 | 15 | 23 | 7 | 12 | 27 | 34 |
| [75,80) | 0 | 0 | 3 | 6 | 9 | 7 | 8 | 2 | 8 | 3 | 0 | 1 | 7 | 9 |
| [70,75) | 0 | 0 | 2 | 6 | 12 | 14 | 1 | 3 | 1 | 4 | 0 | 0 | 5 | 5 |
| [65,70) | 0 | 1 | 1 | 2 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 5 |
| [60,65) | 1 | 0 | 2 | 2 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 3 | 2 |
| [55,60) | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| [50,55) | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| **# Prompts** | **56** | | **271** | | **138** | | **96** | | **98** | | **67** | | **75** | |
| **Average** | **91.52** | **90.54** | **89.46** | **89.39** | **85.76** | **84.75** | **88.07** | **86.35** | **87.19** | **85.92** | **90.07** | **88.36** | **80.40** | **79.67** |

of conversations (2,706), resulting in an average of approximately 6.62 prompts per conversation. Nevertheless, this average is substantially affected by outliers. For instance, there is an exceptional conversation comprising 573 turns, and there are 57 conversations, each with 64 turns. It can be observed that conversations exceeding 30 turns are quite rare. Thus, our analysis focused on conversations consisting of 30 turns or less, yielding an average of 4.05 turns and a median of 2 turns per conversation. Detailed distributions can be referred to Figure 2 (in Appendix).

## 4.2 Answer to RQ2

We notice that for each cluster with similar topics, different categories' performance is different. To obtain a fair comparison result, we utilized Code-Llama-34B [8] and Mistral-7B [6]. These models are not only distinct from ChatGPT but also stand out as top-tier language models in the current landscape, particularly in terms of code comprehension. In conjunction with manual static analysis, we categorized various prompts and conducted a statistical assessment of scores for different patterns, as illustrated in Table 2.

Based on the data in Table 2, it can be observed that among all conversations, the usage proportion of the Output Customization pattern is as high as 33.8%, making it the most frequently used prompt mode by ChatGPT users. Within Output Customization, the Recipe Pattern and Template Pattern are the most prevalent. Prompt Improvement and Interaction also perform well. Input Semantics and Context Control show good accuracy, but they are not used frequently. The formulation of these two prompt modes could be reconsidered in the future. On the other hand, for Error Identification, a significant amount of data involves repeatedly fine-tuning ChatGPT's results, resulting in less than optimal performance. This data indicates that ChatGPT's self-correction ability still needs improvement, and it would benefit from user-provided templates or interactive methods to enhance accuracy.

## 4.3 Answer to RQ3

Regarding the types of issues that developers most frequently present to ChatGPT, we systematically approach this by examining each distinct activity within the software development lifecycle. Statistics indicate that Software Development is the predominant area of inquiry. Consequently, we will first delve into Software Development, followed by a sequential discussion of other activities in the software development lifecycle.

In the **Software Development** phase, prevalent issues include code generation, optimization, and data analysis. From a developer's standpoint, it is imperative to explore efficient and precise patterns to develop a personalized Template Pattern to bolster ChatGPT's performance. Furthermore, inquiries in this phase often involve extensive code, which may sometimes exceed ChatGPT's semantic analysis capabilities. Utilizing Input Semantics or Context Control when interacting with ChatGPT can significantly improve its ability to manage scenarios with extensive code.

In the **Requirements Engineering** phase, questions typically involve collecting and analyzing various issues or making inquiries about system problems. Therefore, clarity in questioning is essential. During this stage, the majority of issues do not involve a significant amount of code. The high accuracy of Context Control is not applicable in such scenarios. However, in this context, Input Semantics maintains an average score of 92.14, demonstrating its effectiveness even in situations where code is not predominant.

In the **Software Design** series of conversations, common inquiries revolve around interface modifications and architectural design. For ChatGPT, questions related to Software Design often involve generating code on the spot and queries about the usage of various tools. Statistics show that all patterns but Error Identification demonstrate great performance.

**Software Quality Assurance** primarily involves providing testing services to users, while **Software Maintenance** involves software upgrades, troubleshooting, and similar activities. In these two phases, Error Identification is the most adopted pattern. Our observation is that multiple rounds of queries with ChatGPT to drive itself to continuously find and solve problems could yield much better performance than a single round of queries.

For **Software Management**, inquiries are predominantly made through direct questioning (excluding code), with output customization being the most common prompt pattern. Notably, the Template mode stands out. Because templates provide ample information to ChatGPT, offering it a wealth of reference material to draw from.

## 5 RELATED WORK

Significant research exists on LLMs in software development, notably their integration throughout the software development lifecycle [4, 7, 9] and prompt engineering optimization [11]. Studies focus on how tailored prompts enhance LLM performance in understanding and addressing complex software problems and code generation [5, 12, 14]. Our discussion synthesizes these two areas to reach an encompassing conclusion.

## 6 CONCLUSION

Our research delves into the interaction between developers and ChatGPT across the software development lifecycle. We analyzed conversation patterns, classifying and summarizing prompt types based on their quality and usage frequency. These insights were then integrated into the software development lifecycle to offer guidance on prompt formulation at different development activities.

## ACKNOWLEDGMENT

This work was supported in part by the Key R&D Program of Hubei Province (2023BAB017, 2023BAB079), HUST CSE-HongXin Joint Institute for Cyber Security, and HUST CSE-FiberHome Joint Institute for Cyber Security.

## REFERENCES

[1] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. *ACM Sigmod record*, 28(2):49–60, 1999.

[2] Sabit Ekin. Prompt engineering for chatgpt: A quick guide to techniques, tips, and best practices. *Authorea Preprints*, 2023.

[3] John Ellson, Emden R Gansner, Eleftherios Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz and dynagraph—static and dynamic graph drawing tools. *Graph drawing software*, pages 127–148, 2004.

[4] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*, 2023.

[5] Zhengbao Jiang, Frank F Xu, Jun Araki, and Graham Neubig. How can we know what language models know? *Transactions of the Association for Computational Linguistics*, 8:423–438, 2020.

[6] MistralAI. Mistral, 2023. mistral.ai.

[7] Ameya Shastri Pothukuchi, Lakshmi Vasuda Kota, and Vinay Mallikarjunaradhya. Impact of generative ai on the software development lifecycle (sdlc). *International Journal of Creative Research Thoughts*, 11(8), 2023.

[8] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[9] Xinyu She, Yue Liu, Yanjie Zhao, Yiling He, Li Li, Chakkrit Tantithamthavorn, Zhan Qin, and Haoyu Wang. Pitfalls in language models for code intelligence: A taxonomy and survey, 2023.

[10] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L Bleris. Code quality analysis in open source software development. *Information systems journal*, 12(1):43–60, 2002.

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

[12] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *arXiv preprint arXiv:2303.07839*, 2023.

[13] Tao Xiao, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. Devgpt: Studying developer-chatgpt conversations. In *Proceedings of the International Conference on Mining Software Repositories (MSR 2024)*, 2024.

[14] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.

## APPENDIX

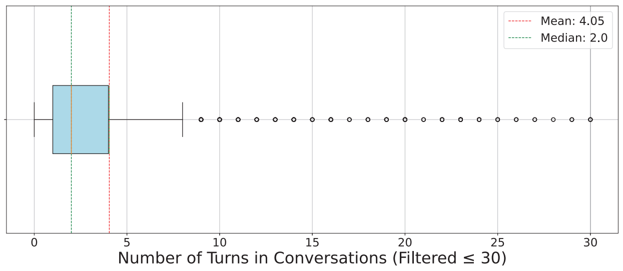

**Figure 2: The distribution of the conversation turns in the DevGPT dataset.**