

From Prompts to Templates: A Systematic Prompt Template Analysis for Real-world LLMapps

Yuetian Mao

Technical University of Munich
Germany
yuetian.mao@tum.de

Junjie He

Technical University of Munich
Germany
junjie.he@tum.de

Chunyang Chen*

Technical University of Munich
Germany
chun-yang.chen@tum.de

ABSTRACT

Large Language Models (LLMs) have revolutionized human-AI interaction by enabling intuitive task execution through natural language prompts. Despite their potential, designing effective prompts remains a significant challenge, as small variations in structure or wording can result in substantial differences in output. To address these challenges, LLM-powered applications (LLMapps) rely on prompt templates to simplify interactions, enhance usability, and support specialized tasks such as document analysis, creative content generation, and code synthesis. However, current practices heavily depend on individual expertise and iterative trial-and-error processes, underscoring the need for systematic methods to optimize prompt template design in LLMapps. This paper presents a comprehensive analysis of prompt templates in practical LLMapps. We construct a dataset of real-world templates from open-source LLMapps, including those from leading companies like Uber and Microsoft. Through a combination of LLM-driven analysis and human review, we categorize template components and placeholders, analyze their distributions, and identify frequent co-occurrence patterns. Additionally, we evaluate the impact of identified patterns on LLMs' instruction-following performance through sample testing. Our findings provide practical insights on prompt template design for developers, supporting the broader adoption and optimization of LLMapps in industrial settings.

CCS CONCEPTS

- Software and its engineering → Software design engineering;
- Computing methodologies → Artificial intelligence.

KEYWORDS

Prompt Engineering, Pattern Analysis, Large Language Models

1 INTRODUCTION

Large Language Models (LLMs), such as GPT-4 [1] and LLaMA [63], have exhibited exceptional capabilities in comprehending and generating text, and LLMs have rapidly evolved to become a cornerstone of many AI-driven applications. This versatility has introduced a new paradigm of interaction between humans and AI, where users express their instructions in natural language, and the LLM generates outputs that align with these specifications. At first glance, a prompt may appear to be a simple question or request, but it actually serves as a bridge between human intent and machine-generated responses, guiding the model by providing context, shaping outputs, and influencing behavior. While these models have significantly reduced the barriers to AI adoption for general

users, crafting clear and effective prompts remains a non-trivial challenge [16, 75]. This difficulty arises from the limited understanding of how LLMs process input information and the fact that even minor variations in prompts can lead to substantial changes in model performance [58, 61].

To address these challenges, numerous LLMapps, which refer to the type of software that uses LLMs as one of its building blocks, have been developed [27, 72] in different downstream domains. The growth of LLMapps has been remarkable, with over one million applications released for public use and leading LLMapps now boast more than three million monthly active users [76]. For example, “Write For Me” is one of the most popular LLMapps in the GPT store, with over 12 million conversions and a high rating of 4.3 based on over 10,000 reviews [52]. Unlike traditional software development that involves full-stack implementation, LLMapps adopt a low-code paradigm where LLMs handle most user’s queries, and developers focus on facilitating user-LLM interaction by designing lightweight prompt templates which is a predefined structure that combines static text with dynamic placeholders, allowing developers to create adaptable prompts for LLMapps [19]. These predefined prompt templates simplify and standardize user interactions with LLMs [20, 60], ensuring consistency and efficiency across various tasks.

Figure 1 presents examples of prompt templates designed for various tasks, showcasing their components and placeholders. For example, the first template establishes the model’s role, provides task-specific directives (e.g., song suggestion), imposes constraints to reduce model hallucination, and defines the expected output format to be a JSON string with specific attributes. The *{user_requirement}* placeholder allows for diverse user inputs, such as “light music for bedtime” or “a song with a summer feel describing ... life in ...”. The other two templates demonstrate the use of examples and detailed workflow descriptions, employing in-context learning and chain-of-thought techniques to guide the LLMs effectively. Serving as critical artifacts in LLMapps, prompt templates reduce the complexity of crafting effective prompts, bridging the gap between user intent and machine responses. Similar to GUI (Graphical User Interface) which bridges the gap between backend programs and end-users, prompt templates are like a “textual GUI” for AI systems, where inputs follow a structured format requiring only basic inputs, such as a few clicks or filling in the blank, instead of long complete instruction input. Just as GUIs have widgets that can be reused or customized, prompt templates have components and placeholders that allow flexibility and dynamic usage which hide the intricate operations of the backend from the user. From setting the context and defining instructions to dynamically adjusting the content based on user needs, prompt templates offer a versatile approach to language model interactions. By providing a consistent structure, prompt

*Chunyang Chen is the corresponding author.

<p>You are a music recommendation system. You will reply with song recommendations that are perfect for <code>{user_requirement}</code>. Only reply with the JSON object, no need to send anything else. Don't make up things.</p> <p>Include title, artist, album, and year in the JSON response. Use the JSON format:</p> <pre>{"songs": [{"title": "Title of song 1", "artist": "Artist of Song 1", "album": "Album of Song 1", "year": "Year of release"}]}</pre>	<p>Generate a Python code file according to the following content and save it under the <code>learn_skill</code> path. This file should contain one utility function: <code>{skill_name}</code>.</p> <p>Code description: <code>{code_str}</code></p> <p>The written method needs to add the annotation <code>@openai_func</code>, and generate method annotations to ensure that the code is indented, similar to the following:</p> <pre>from openai_util.function_call.openaifunc_decorator import openai_func @openai_func def get_weather(location: str, unit: str): """ Retrieve the weather information for a specific location. :param location: The name of the location (e.g., "San Francisco"). :param unit: The unit of temperature (e.g., "Celsius" or "Fahrenheit"). """ </pre>
<p>{visualization} Let's generate a level 1 NL description step by step.</p> <p>Step 1. Determine if the visualization contains composite views, such as layered plots, trellis plots, or other types of multiple view displays, and provide a count of the number of plots if any are present.</p> <p>Step 2. Analyze the semantics of each chart individually, including [Data], [Transform], [Mark], [Chart-Type], [Encoding], [Style], and [Interaction]. Refer to this: {semantic_documentation}</p> <p>Step 3. Generate a level 1 NL description using the semantics. It contains elemental and encoded properties of the visualization (i.e., the visual components that comprise a graphical representation's design and construction).</p> <pre>## Step 1. Composite Views: - True/False: - (If True) Type: (layered, trellis, multiple views) Step 2. Chart Semantics: - Data: - Field (Value): - Transform: - Mark: - Number of plots: Step 3. Level 1 NL Description: - Chart-Type: - Encoding: - Style: - Interaction (e.g., tooltip):</pre>	

Prompt Component Types Directive Context Output Format/Style Constraints Profile/Role Workflows Examples Placeholder

Figure 1: Examples of prompt template [23, 36, 56]

templates help users articulate their requirements more effectively, ensuring that the LLM understands and responds appropriately even in specialized domains, such as document analysis [65], creative content generation [9], and code synthesis [22].

As a new paradigm to develop LLMapps almost all commercial LLM providers support prompt templates, such as OpenAI’s GPT series [49], Google’s Gemini [19], Anthropic’s Claude [3], Salesforce [57], and Langchain [39]. Unlike traditional software development, which is deterministic and relies on well-defined code [24, 41, 74], LLMapps introduce variability in outputs, challenging software engineers to design prompts that ensure reliability, maintainability, and performance. However, it is still unclear how to develop high-quality prompt templates which are not even clearly mentioned in the documentation from these leading commercial LLM providers. What information should developers put into prompt templates? What patterns do they follow? What dynamic content will prompt templates take from end users? How do different construction patterns in prompt templates influence LLMs’ instruction-following abilities? Considering the growing significance of prompt templates in the LLMapp ecosystem, a systematic understanding of their structure, composition, and effectiveness is urgently needed.

In this paper, we analyze prompt templates used in LLM-powered applications, focusing on their components, placeholders, and composition patterns. Our study utilizes a publicly available dataset of prompts collected from open-source LLMapps [51], encompassing applications from IT giants to innovative startups that reflect real-world use cases and industry trends, such as tools from Uber (adopted by over 200 developers) [35, 54] and Microsoft (over 5k GitHub stars) [53]. We pre-process the dataset to filter representative prompt templates and systematically identify their components and placeholders. By analyzing the content and order of these elements, we uncover structural and content patterns commonly employed in prompt template design. To evaluate these patterns, we conduct sample testing using randomly selected templates, assessing their impact on LLM instruction-following abilities and identifying optimal patterns for enhanced performance.

Our contributions are summarized as follows:

- To the best of our knowledge, this is the first study analyzing real-world prompt templates in LLMapps, with datasets publicly available at: <https://github.com/RedSmallPanda/FSE2025>.

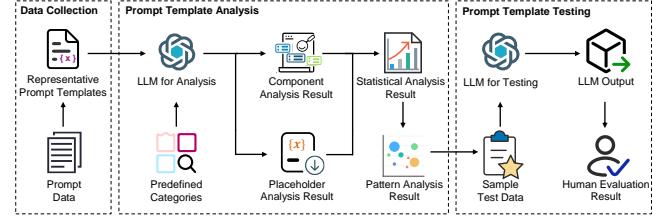


Figure 2: An illustration of the main pipeline

- We categorize key components, placeholders, and patterns from collected large-scale prompt templates, offering insights into effective prompt template design practices.
- Through sample testing, we demonstrate that well-structured prompt templates and specific composition patterns can significantly improve LLMs’ instruction-following abilities.

2 METHODOLOGY

2.1 Overview

Figure 2 shows the main framework of our method including three main steps: the construction of large-scale prompt templates from real-world GitHub repositories, an empirical study of prompt templates focusing on components, placeholders, patterns, and an exploratory study to check the effects of these prompt template patterns to LLM output.

2.2 Data Collection

We construct our dataset based on PromptSet [51], a collection of prompts extracted from LLMapps in open-source GitHub projects as of January 10, 2024. These projects vary widely in complexity, usage, and popularity, ranging from personal demos to widely adopted applications, resulting in significant variability in prompt quality. To ensure a high-quality dataset suitable for industrial applications, we design a processing pipeline that assigns quality metrics to each prompt and automatically filters out lower-quality examples.

Our data collection pipeline begins by selecting non-empty English prompts from the PromptSet dataset, resulting in 14,834 records. For each corresponding GitHub repository, we retrieve metadata such as *star count* and *latest update time* to evaluate repository popularity and activity [11, 69], utilizing the GitHub API, accessed on June 20, 2024. We filter repositories with at least five stars and recent updates within the past year, narrowing the dataset to 2,888

Table 1: Representative LLMApp repositories [35, 38, 53, 68]

Company Name	Repo Title	Description	#Stars
Uber	uber/piranha	Tool for refactoring code related to feature flag APIs	2.3k
Microsoft	microsoft/TaskWeaver	Code-first agent framework for data analytics	5.4k
Weaviate	weaviate/Verba	RAG chatbot	6.5k
LAION	LAION-AI/Open-Assistant	Chat-based assistant	37.1k

#Stars assessed on 30.12.2024.

records across 1,525 repositories. Next, we separate multi-prompt records into individual entries, resulting in 5,816 prompts, and then remove duplicates to obtain 4,540 unique prompts. To further enhance data quality, we exclude prompts shorter than five tokens[25], with 4,210 high-quality prompts left. Finally, we extract 2,163 distinct prompt templates using the llama3-70b-8192 model, guided by a clear definition from Schulhoff et al. [60], supplemented by illustrative examples. Most repositories in our dataset are LLMApps serving for information seeking, editing, coding, and creative writing. As seen in Table 1, some of them are from leading companies and organizations, such as Uber’s tool for refactoring code related to feature flag APIs (adopted by over 200 developers [35, 54]), Microsoft’s code-first agent framework for executing data analytics tasks with over 5k GitHub stars [53] and some very popular ones, such as an RAG chatbot powered by Weaviate with over 6k GitHub stars [68], and LAION-AI’s Open-Assistant program, with more than 37k stars [38].

2.3 Prompt Template Analysis

In this section, we address the following research questions:

- **RQ1: What are the common constituent components in prompt templates, and what specific words, phrases, and patterns are frequently used within these components?**

We investigate the component composition of representative prompts extracted from LLMApps and analyze co-occurrence patterns among components. Building on these findings, we conduct word-level and phrase-level analysis of each component type to uncover variations in how similar content is expressed (e.g., different ways users describe the output format as “JSON”). From these variations, we identify common patterns that impact response format and content.

- **RQ2: How do placeholders vary in terms of types and positions within prompt templates?**

We analyze the types of placeholders (variables that will be replaced by input text to create a complete prompt) within prompt templates [60], categorizing them based on their variable names and contextual usage within the templates. Then we examine the positional distribution of placeholders across prompt templates.

2.3.1 Prompt Component Analysis.

In this step, we define a comprehensive set of common prompt components by synthesizing insights from prominent prompt engineering frameworks and AI service platforms. Specifically, we extract component definitions from guidelines provided by Google Cloud’s documentation [10], the Elavis Saravia framework [59], the CRISPE framework [47], and the LangGPT framework [64]. To construct a unified list, we merge similar components across these sources. For instance, components like Profile, Capacity, Role, and Persona, all of which define the model’s behavior or identity, are

consolidated into a single component labeled “Profile/Role.” The merged component list is presented in Table 2, and component definitions and frequency distributions are detailed in Table 3.

Table 2: Prompt components across different frameworks and documentations.

LangGPT [64]	Elavis Saravia [59]	CRISPE [47]	Google Cloud[10]	Merged
Profile	-	Capacity and Role	Persona	Profile/Role
Goal	Instruction	Statement	Objective Instructions System Instructions	Directive
Workflow	-	-	Reasoning Steps	Workflow
Initialization Background	Context	Insights	Context	Context
Example	Input Data	-	Few-shot Examples	Examples
Output-format Style	Output Indicator	Personality	Response Format Tone	Output Format/Style
Constraints	-	-	Constraints Safeguards	Constraints
Skill Suggestion	-	Experiment	Recap	Others

We leverage the llama3-70b-8192 model to identify components from the merged list present in the prompts, employing a predefined prompt template that specifies all available components and outputs results in a structured JSON format.

To assess the accuracy of component identification, we perform both component-level and prompt-level human evaluations on a randomly selected 5% sample of prompts. At the component level, precision is calculated as the proportion of correctly identified components compared to human-labeled ones. At the prompt level, a prompt is classified as an exact match only if all identified components are correct; prompts with at least one correct component are classified as partial matches [33].

Human evaluations follow established practices [40], with two evaluators both with over one year of programming experience in LLMapp independently reviewing LLM-generated classifications. A component is considered correctly identified if both evaluators agree. Placeholder identification accuracy is validated in the same manner. For prompt template testing, evaluators score LLM outputs based on predefined metrics, with the final score being the average of their assessments.

The evaluation results indicate high component-level precision, averaging 86% across all predefined types. At the prompt level, full match precision is 66%, while partial match precision reaches 99%, demonstrating the method’s reliability in component detection.

2.3.2 Placeholder Analysis.

We analyze placeholders within prompt templates using a two-step iterative process. First, we manually classify placeholders from a randomly selected set of 100 templates into predefined or newly identified categories based on their names, following practices similar to variable analysis [5]. Then, we use gpt-4o to extend this classification to the full dataset, leveraging the initial categories and definitions.

To ensure accuracy, we conduct human evaluations to verify the LLM’s classifications, merging underrepresented categories and refining definitions as needed. After incorporating these adjustments, we perform a second round of LLM classification, followed by another human evaluation. This iterative process achieves an

overall classification accuracy of 81% on a randomly selected sample of 80 records. Ultimately, we identify four primary placeholder categories, as detailed in Table 5.

2.3.3 Statistical Analysis and Pattern Analysis.

Using the component analysis and placeholder analysis results, we conduct a statistical analysis to examine the distribution of each category, as well as the frequency and relative positioning of elements at the word and phrase levels. From these statistical findings, we extract notable patterns that reveal structural and content trends within the prompt templates.

2.4 Effects of Prompt Template Patterns

In this section, we address the following research question:

- RQ3: How do different construction patterns in prompt templates influence LLMs' instruction-following abilities?**

We sample prompt templates with specific patterns identified in earlier research questions, populate them with sample data, and generate outputs to assess the effectiveness of these patterns on LLMs' instruction-following abilities through human evaluation.

In this phase, we assess the impact of various patterns identified in the analysis of RQ1 and RQ2 on LLM output, focusing on two key dimensions: Content-Following, which ensures semantic accuracy with task goals, and Format-Following, which enforces adherence to structural or syntactical requirements critical for LLMs but underexplored in prior research [70]. Specific metrics are defined for each test to evaluate one or both dimensions.

For each pattern, we randomly sample prompt templates that incorporate it and populate them using either the gpt-4o model or real-world data sources, such as a medical QA dataset [6], Java documentation [50], and GitHub projects [37]. To compare patterns addressing similar issues, we manually reformulate the templates (e.g., modifying the output format or adding constraints) to fit alternative patterns. Outputs are then generated using both llama3-70b-8192 and gpt-4o models, followed by human evaluations to assess output quality against predefined metrics.

3 ANALYSIS RESULTS

3.1 RQ1: Analyzing Components in Prompt Templates

3.1.1 Distribution of Components.

Table 3 shows the detection results for the seven categories of components, indicating the percentage of prompt templates containing each component. Among these components, the four most common are Directive, Context, Output Format/Style, and Constraints. The Directive represents the task intent of the prompt, guiding the language model on how to perform a task. Most prompts require a clear and complete directive to instruct the model effectively. The Context typically includes the input content and relevant contextual descriptions, helping the model understand the task in detail. Given that these prompt templates are designed for LLMs, developers often specify an Output Format/Style (e.g., Topic: Title - Explanation, as illustrated in Figure 3) and set Constraints (e.g., length, number of results, output topic scope). This ensures the generated content is more predictable and easier for downstream applications to process, and maintains consistency across outputs.

Profile/Role Directive Context Workflow Constraints Output Format/Style Examples	<p>You are an AI assistant acting as a content advisor for a tech blog. Suggest two relevant blog topics based on recent trends in {subject_area}.</p> <p>You are given the following information: - The blog focuses on topics related to {subject_area}. - Recent posts on {high_engagement_topics} have received high engagement.</p> <p>To complete this task, follow these steps: 1. Review the provided context and analyze current trends in {subject_area}. 2. Suggest two blog topics that align with the blog's focus and audience. 3. Ensure the suggested topics are relevant to the recent engagement trends, particularly {high_engagement_topics}, and are accessible to a general audience.</p> <p>Avoid overly technical or niche topics that may overwhelm readers. Keep topics broad and engaging for non-experts.</p> <p>Provide your response in the following format: Topic 1: [Title] - [One-sentence explanation] Topic 2: [Title] - [One-sentence explanation]</p> <p>There are two example topics based on trends in AI: Topic 1: AI in Public Services - Discuss how AI is being used to enhance efficiency in public services like healthcare and education. Topic 2: Ethical AI in Automation - Explore the ethical implications of AI-driven automation in industries like manufacturing and logistics.</p>
---	--

Figure 3: A Template Example following General Component Order.

Table 3: Frequencies distribution of different prompt components.

Components	Definition	Frequency
Profile/Role	Who or what the model is acting as.	28.4%
Directive	The core intent of the prompt, often in the form of an instruction or a question.	86.7%
Workflow	Steps and processes the model should follow to complete the task.	27.5%
Context	Background information and context that the model needs to refer to.	56.2%
Examples	Examples of what the response should look like.	19.9%
Output Format/Style	The type, format, or style of the output.	39.7%
Constraints	Restrictions on what the model must adhere to when generating a response.	35.7%

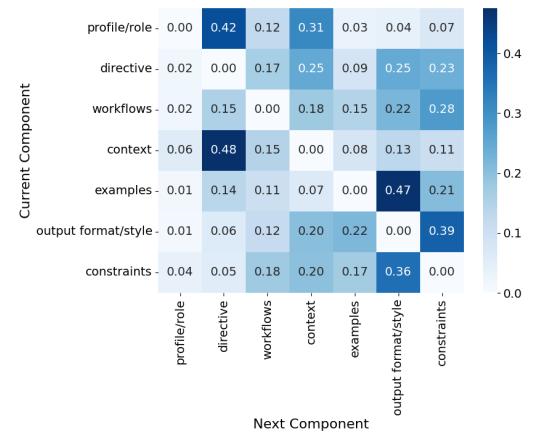


Figure 4: Component order probability matrix

3.1.2 Component Order.

We also investigate the relative positions of various components. We observe that the Profile/Role and Directive components commonly appear in the first position rather than other positions, with a probability of 0.87 and 0.65, respectively. As shown in Figure 4, both the X and Y axes represent the different component types,



Figure 5: A Common Component order

with each coordinate point indicating the probability that component X follows component Y. The darker the color, the higher the probability that this positional pattern occurs. For example, Context is most likely to be followed by Directive with a probability of 0.48.

Based on the analysis, we identify a common sequential order of components in prompt templates, as depicted in Figure 5. This sequence assumes each component appears exactly once. Notably, “Context and Workflows” and “Output Format/Style and Constraints” consistently form two pairs, where the order within the “Output Format/Style and Constraints” pair is flexible. Additionally, the relative positioning of these two pairs can also be interchanged without impacting overall structure. Figure 3 illustrates a specific prompt example that adheres to this identified order.

Finding 1: There are mainly seven types of common components within prompt templates according to our observation. Developers commonly put the profile/role and their directive at the beginning of the prompt, establishing the model’s identity and task intent, while examples are typically at the end.

3.1.3 Component Content.

We perform an in-depth analysis of specific prompt components that have a significant impact on the structure consistency and instruction relevance of LLM responses, focusing on commonly used words, phrases, and formats associated with these components. Among the seven types of prompt components, we focus on three: Directive, Output Format/Style and Constraints. In direct interactions with LLMs (such as live chat), users primarily care about the relevance and correctness of the response. However, in LLMApps, developers are not only concerned with the response content but also with its format, as post-processing often relies on structured outputs. Ensuring a limited or predictable output format can significantly reduce errors during post-processing. Based on our observations of how LLMApps handle responses, we focus on Directive, which encapsulates the core user intent, along with Output Format/Style and Constraints, as they play a critical role in shaping both response format and content—key factors for downstream processing and application performance.

Directive. Directive in prompts could be either in question or instruction style [32, 60]. Using regex patterns to capture directives starting with question words such as “how” and “what” or ending with a question mark “?”, we classify the directive into two types, namely instruction and question. Notably, our analysis shows that over 90% of directives are written in the instruction style. This is likely due to the fact that instructions such as “Summarize the report” are more direct and clearer for the model to understand than question-style directives like “Could you summarize this?”.

Finding 2: The directive component i.e., user intent in prompt templates predominantly adopts an instructional style, which is more commonly used than the question style.



(a) Information Seeking. (b) Coding & Debugging. (c) Editing.

Figure 6: Word cloud for output format of selected themes

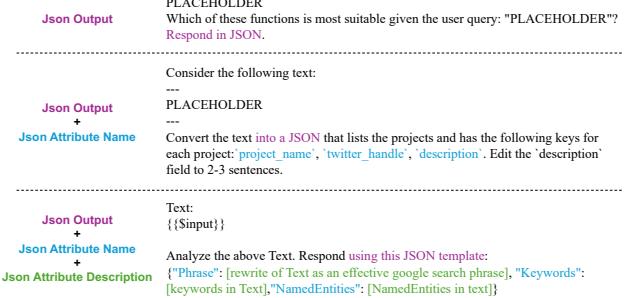


Figure 7: Three different JSON output formats [14, 28, 62]

Output Format/Style. We analyze the output formats specified in the prompt templates across different themes. To extract these formats, we consider the most frequently occurring terms in the output descriptions of the prompt templates and map them into word clouds for each theme, as shown in Figure 6. Words in the cloud are sized proportionally to their frequency: the larger the word, the more often that output format is used within the theme.

From our analysis, certain formats like “score” and “code” are frequently used in particular themes. For example, “code” is a predominant output format in the “Coding & Debugging” tasks, often appearing as the result of tasks such as code generation, bug fixing, and code summarization. Similarly, “score” appears frequently in “Information Seeking” tasks, where users provide criteria for LLMs to evaluate inputs and generate numerical scores.

Across all themes, the most common output format besides standard text is JSON. JSON’s structured nature makes it particularly popular, as it is easy for applications to post-process and provides a user-friendly way to organize complex information.

To analyze how developers define JSON output formats, we extract all prompt templates that specify JSON as the required output format. From the data, we identify three key components frequently used: *Json Output*, which specifies that the output must be in JSON format; *Json Attribute Name*, which defines the names of the attributes to be included; and *Json Attribute Description*, which provides detailed explanations for each attribute. These components combine into three distinct patterns. In *Pattern 1*, developers only indicate that the output should be in JSON format, often accompanied by general guidelines in natural language about the expected content. *Pattern 2* builds on this by explicitly listing the attribute names to be included, ensuring structural consistency in the output. Finally, *Pattern 3* adds detailed descriptions for each attribute to *Pattern 2*, enhancing clarity and ensuring attributes are well-defined, particularly for complex outputs. Figure 7 illustrates these patterns and their composition.

The distribution of these patterns across all observed templates reveals that Pattern 1 (Json Output) accounts for 36.21%, Pattern 2 (Json Output + Json Attribute Name) for 19.83%, and Pattern 3 (Json Output + Json Attribute Name + Json Attribute Description) for 43.97%.

Finding 3: JSON is the most commonly used output format in prompt templates. Notably, over one-third of these JSON formats rely on informal descriptions without explicitly defined attribute names.

Constraints. We use the llama3-70b-8192 model to identify various constraint types in prompt templates, based on the classifications established by Ross Dawson [13]. The most common types are “Exclusion” (46.0%), “Inclusion” (35.6%), and “Word count” (10.5%). These categories reflect the primary considerations developers emphasize when designing prompt templates. The “Inclusion” constraints guide the model to focus on specific details, improving the relevance and precision of its responses. “Word count” constraints encourage concise responses, enhancing usability and reducing API costs by minimizing token usage.

Given that “Exclusion” constraints account for nearly half of all identified constraints, we conducted a detailed analysis of this category. Using the all-mpnet-base-v2 embedding model and k-means clustering, we identified five subcategories: “Accuracy and Relevance”, “Clarity about Unknowns”, “Output Control”, “Redundancy and Context Adherence”, and “Technical Restriction” (Table 4). These subcategories reflect nuanced developer intent in managing model outputs.

Key subcategories, such as “Accuracy and relevance” and “Clarity about unknowns” are designed to mitigate hallucinations, a prevalent issue with LLMs. For example, prompts like “Don’t try to make up an answer” reduce the risk of speculative or incorrect outputs, enhancing reliability. Anthropic’s documentation [4] and Chen et al. [8] propose similar constraints to mitigate the hallucination. Additionally, constraints targeting redundancy, such as “Do not generate redundant information” streamline responses, reducing post-processing needs in tasks like code generation. The “Technical restriction” subcategory highlights constraints for specific contexts, such as database queries or API calls, ensuring that models adhere to precise technical requirements like predefined structures or restricted API usage. These constraints are critical for maintaining correctness and efficiency in structured workflows.

Finding 4: Developers frequently use exclusions as the constraints to refine outputs such as excluding irrelevant content, reducing hallucinations, and narrowing search space for generation.

3.2 RQ2: Analyzing Placeholders in Prompt Templates

3.2.1 Classification of Placeholder.

Table 5 displays the percentage of prompt templates containing each placeholder category. The most prevalent categories are Knowledge Input and Metadata/Short Phrases. Knowledge Input placeholders contain the main content with which the LLM directly interacts, including items like “report” or “code snippet.” Metadata/Short

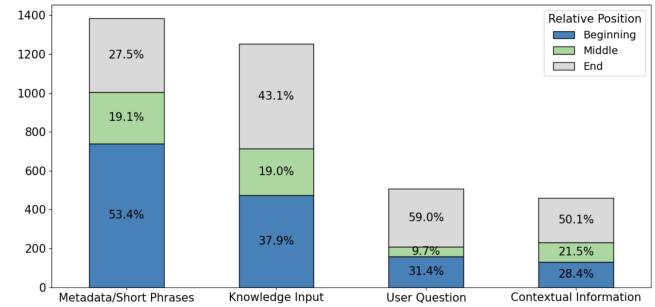


Figure 8: Frequency distribution of placeholder positional occurrences for different placeholder types.

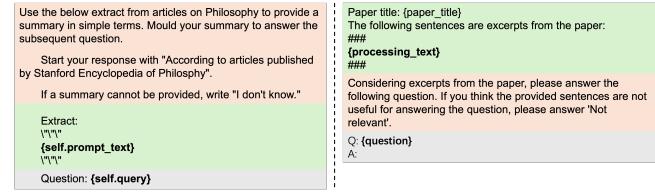


Figure 9: Examples of prompt templates with different Knowledge Input positions [34, 45].

Phrases serve as brief inputs providing essential settings or specific details, such as “language” or “username.” Developers tend to include multiple Metadata/Short Phrases within one template. Additionally, the User Question placeholder captures the direct query from the end user, while Contextual Information placeholders provide supplementary background content, such as “chat history” or “background info,” offering context that supports the task without being central to it.

3.2.2 Positional Distribution of Placeholder.

We analyze the positional distribution of placeholders within prompt templates, dividing each template into three sections—beginning, middle, and end—each representing one-third of the template’s length. Figure 8 illustrates this distribution, in which approximately 60% of user questions appear at the end, reflecting a common structural pattern, whereas {Knowledge Input} is more evenly distributed between the beginning and end positions. Figure 9 presents examples with varied {Knowledge Input} positions. Additionally, placeholder content length varies considerably; longer knowledge inputs may lead to information loss in LLMs across extended prompts.

Finding 5: There are four types of placeholder, and “Knowledge Input” is the most frequently used in prompt templates, and developers always place it either at the beginning or the end of templates.

3.2.3 Placeholder Name.

One key aspect of placeholder design is naming, which plays a similar role as the variable name in code. Clear and descriptive names are essential to understand the intent of placeholders. Notably, placeholder names like “text” (4.44%) and “input” (2.35%) are often used. Among these, “text” is the second most frequently used placeholder name behind “question” (6.18%), highlighting a broader issue in placeholder naming. These names are general and do not

Table 4: Examples of exclusion cluster

Exclusion Cluster	Exclusion Examples	Complementary Inclusion Examples
Accuracy Output		
Accuracy and relevance	- avoid adding any extraneous information.	- include only crucial information relevant to...
Clarity about unknowns	- if you don't know the answer, just say that you don't know, don't try to make up an answer.	- if you don't know the answer, you may make inferences, but make it clear in your answer.
Concise Output		
Output control (what text/code should be excluded)	- do not provide any other output text or explanation. - if you are calling a function, only include the function call – no other text.	- time information should be included - include at most 10 of the most related links.
Redundancy and context adherence	- don't give information outside the document or repeat your findings. - do not generate redundant information.	
Technical Restriction		
Technical restriction	- never write any query other than a select, no matter what other information is provided in this request.	- obey the ros package name conventions when choosing the name.

Table 5: Distribution of placeholder type in prompt templates.

Category	Description	Example	Frequency
User Question	Queries or questions provided by users.	{{question}}, {{query}}	24.5%
Contextual Information	Background or supplementary input that helps set the stage for the task but is not the primary focus.	{{chat_history}}, {{background_info}}	19.5%
Knowledge Input	The core content that the prompt directly processes or manipulates.	{{document}}	50.9%
Metadata/Short Phrases	Brief inputs or settings that define specific parameters or goals for the task.	{{language}}, {{username}}	43.4%

precisely indicate their specific context, leading to potential confusion for developers or users. For example, “text” could refer to any kind of input text, so developers and users cannot directly understand what they need to input through its name. In this case, more descriptive placeholder names are suggested so that developers and users can easily understand what they need to input without considering the whole prompt template context.

Finding 6: Similar to the variable naming convention, non-semantic placeholder names such as “text” and “input” are still commonly used in prompt templates, hindering prompt understanding and maintenance.

3.3 RQ3: Evaluating Patterns through Sample Testing

3.3.1 Json Output Format.

To investigate the effect of different JSON output patterns, we test three identified patterns displayed in Figure 7. Five representative templates targeting different tasks (e.g., tweet analysis, DNS parsing) are selected for each pattern and three diverse input instances are generated per template, yielding a total of 45 templates after reformatting them to fit all patterns. We define two metrics rated from 1 to 5 to evaluate the JSON object in LLM output [31, 70]:

Table 6: LLM output quality under three json output format patterns.

Pattern	Format Following		Content Following	
	llama3-70b-8192	gpt-4o	llama3-70b-8192	gpt-4o
Pattern 1	3.09	3.21	3.70	3.50
Pattern 2	4.66	4.86	4.02	4.30
Pattern 3	4.90	4.96	4.47	4.53

Pattern 1 is Json Output, Pattern 2 is Json Output + Json Attribute Name, Pattern 3 is Json Output + Json Attribute Name + Json Attribute Description.

- *Format Following:* Measures the consistency of generated JSON strings with the defined format, including attribute count, names, and structural uniformity across outputs.
- *Content Following:* Assesses the alignment of generated content with the intent in the prompt template.

As shown in Table 6, both models exhibit similar trends, with Pattern 3 achieving the highest scores across metrics and models. In Format-Following, Patterns 2 and 3 outperform Pattern 1, which scores lower due to inconsistencies in attribute count and naming, stemming from its lack of explicit definitions. Patterns 2 and 3 mitigate these issues through explicit “Json Attribute Names”, ensuring greater structural consistency.

For Content-Following, Pattern 3 achieves the highest scores, with llama3-70b-8192 at 4.47 (+0.45) and gpt-4o at 4.53 (+0.23), suggesting that detailed “Json Attribute Descriptions” enhance the model’s ability to generate content that aligns closely with user requirements. These findings underscore the value of attribute-level detail in improving both precision and semantic adherence.

Figure 10 illustrates an example of extracting tweet information using three prompt templates with identical input text but differing JSON output formats. Case 1 broadly requests a JSON output without defining required fields, leading to redundant outputs like “text”, “mentions”, and “hashtags”. Case 2 specifies field names like “tweet_id” and “engagement”, producing more structured outputs,

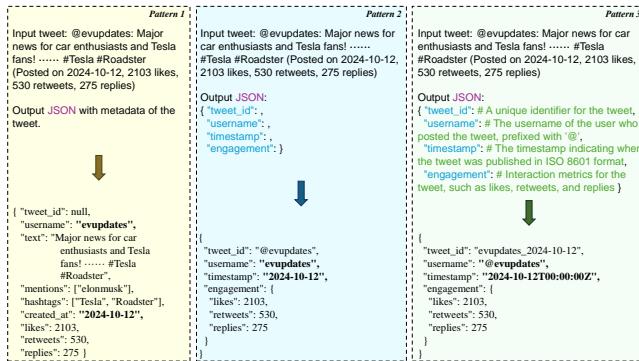


Figure 10: Output examples of different json output format patterns [30].

but attributes like “username” and “timestamp” remain ambiguous without further clarification. Case 3 resolves these ambiguities by adding clear descriptions for each attribute (e.g., defining “username” as the Twitter handle prefixed with “@” and specifying “timestamp” to follow the ISO 8601 format [29]), resulting in outputs that are more accurate and aligned with user expectations.

Finding 7: When generating LLM outputs in JSON format, explicit Json Attribute Names enhance format consistency by ensuring structured and uniform outputs. Additionally, detailed Json Attribute Descriptions further refine content-following, reducing ambiguity and better aligning with user-defined requirements.

3.3.2 Exclusion Constraint for Output.

Well-defined JSON output formats enhance the format-following abilities of LLMs when generating JSON objects. However, as observed in the previous experiment, LLMs often include redundant explanations alongside the JSON object, leading to output inconsistency. This experiment evaluates the effectiveness of exclusion constraints in reducing such redundancies. Using the same 15 populated templates from the JSON Output Pattern 3 experiment, we apply an exclusion constraint, “Do not provide any other output text beyond the JSON string”, positioned before the JSON format definition as per the component order identified in findings 1. For evaluation, we define the metric *Format Following* as a binary value: “1” if the output consists solely of the JSON string and is ready for direct parsing, and “0” otherwise.

Results highlight the impact of the exclusion constraint. For the llama3-70b-8192 model, the original prompts yield a Format Following rate of 40% (only JSON string in 40% of outputs). The constraint raises this rate to 100%, demonstrating improved adherence. In 16.67% of cases, the output is enclosed in ““json”” without any other explanation text, which we do not consider redundant. The gpt-4o model performs better with the original prompts, achieving an 86.67% adherence rate, further increasing to 100% with the exclusion constraint applied. These findings underline the exclusion constraint’s value in improving clarity, reducing redundancy, and ensuring strict adherence to output format requirements.

Finding 8: Using only JSON format definitions is insufficient to fully prevent extraneous explanations or comments. Combining “Do” instructions, such as explicit output format definitions, with “Don’t” instructions, like exclusion constraints, significantly reduces redundancy while maintaining high output consistency in LLM-generated content.

3.3.3 Position of Knowledge Input Placeholder.

In this experiment, we explore the optimal position of the {Knowledge Input} placeholder, the most commonly used type, as identified in Finding 5. We use a retrieval-augmented generation (RAG)-style task where users pose questions, external knowledge is provided as input, and developers define instruction (e.g., directive, constraint, output format) to process the dynamic input. To assess the impact of the {Knowledge Input} placeholder’s position, placed either at the beginning or end of the prompt, as noted in Finding 5, we test two configurations: Instruction First, where the task-intent portion of the instruction precedes the knowledge input, and Placeholder First, where the knowledge input precedes the instruction. In both configurations, the {User Question} placeholder remains at the end, following the most frequent position of {User Question} in Figure 8. The populated templates span a variety of topics, such as medical and code-related QA. We also select inputs of varying lengths to adapt to real-world usage scenarios. Evaluation is based on two human-assessed metrics, scored on a 1–5 scale, which assesses the relevance to both the developer-defined task instruction and the end-user question in general task-oriented Q&A systems [44]:

- *Content Following (Question):* Alignment with user question.
- *Content Following (Task Intent):* Adherence to the developer-defined task intent (e.g., including directive, constraint, and output format).

Table 7 presents the average Content Following scores under different configurations. Both models exhibit high scores for Content Following (Question), reflecting consistent alignment with user queries at the end of the prompt template regardless of placeholder position. However, for Content Following (Task Intent), the Placeholder First pattern consistently outperforms the Instruction First pattern, with average scores of 4.63 for LLaMA (+0.91) and 4.60 for GPT (+0.34). Notably, as knowledge input length increases, the Instruction First pattern suffers a more significant performance drop compared to the Placeholder First pattern. This suggests that placing the task intent of instruction before the knowledge input increases the likelihood of forgetting or misalignment as input length grows. Conversely, placing the task intent of instruction after the knowledge input mitigates this issue, maintaining robust task intent adherence even with long inputs.

Table 7: Output quality under different position patterns

Pattern	Model	Content Following (Question)	Content Following (Task Intent)
Instruction First	LLaMA	4.52	3.72
	GPT	4.61	4.26
Placeholder First	LLaMA	4.84 (+0.32)	4.63 (+0.91)
	GPT	4.70 (+0.09)	4.60 (+0.34)

Instruction First Pattern is Task Intent -> {Knowledge Input} -> {User Question}.
Placeholder First Pattern is {Knowledge Input} -> Task Intent -> {User Question}.

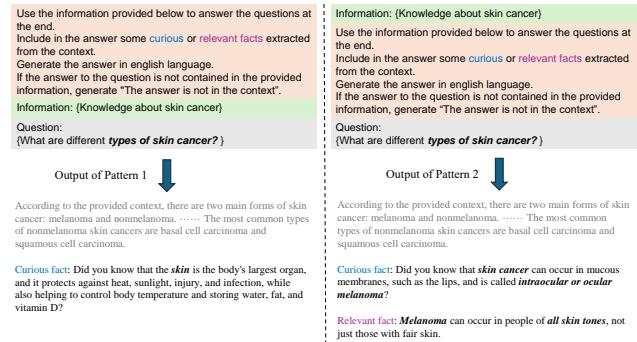


Figure 11: Output examples with Knowledge Input in different prompt template positions [7].

Figure 11 presents a comparison using a long input text. The left example follows the Instruction First pattern, while the right one follows Placeholder First. In both, the user question remains at the end of the prompt template. Colored sections highlight the instruction (red), knowledge input (green), and user question (grey). The upper portion of each side presents the filled prompt template, while the lower portion showcases the respective LLM output. In this example, the long input is a detailed description of skin cancer. Regarding the LLM outputs, both patterns provide similar responses to the user question, with the middle portion of the long description about skin cancer being reduced in both cases. However, the differences arise in how the LLM addresses the facts required by the instruction. In the Instruction First pattern, the “curious fact” output discusses what the skin is, which, although interesting, is unrelated to the user question. Additionally, the relevant fact is missing. In contrast, the Placeholder First pattern generates more aligned results. The “curious fact” and “relevant fact” both relate to skin cancer types, discussing where they occur (e.g., on mucous membranes) and their occurrence across various skin tones, thus providing a more relevant and comprehensive response to the instruction and user question.

Finding 9: Positioning the task intent of instruction and user question after the knowledge input enhances output consistency and mitigates information decay in Q&A tasks, particularly when processing prompt templates with highly variable input lengths in LLMApps.

4 IMPLICATIONS

Our research offers actionable insights into prompt engineering for various parties in the software industry:

4.1 Implications for LLM Providers

LLM providers can enhance the usability and performance of their APIs by offering best practices for designing, testing, and optimizing prompt templates. Almost all commercial LLM providers support prompt templates, such as Google Cloud Gemini [19] and Langchain [39]. However, none of them provides any guideline on how to write effective prompt templates.

Pre-defined prompt templates. To address the challenges developers face in crafting effective prompts, LLM providers could

offer pre-defined templates for common tasks. For instance, templates could include an information-seeking task with JSON as the output format (see Section 3.1.3) or a RAG-style question-answering task (see Section 3.3.3). By leveraging the overall component order identified in Section 3.1.2, providers can establish a unified structure for these templates. Furthermore, based on the pattern testing results outlined in Section 3.3, API providers can adopt optimal writing patterns tailored to specific tasks, ensuring improved generation performance for these pre-defined templates.

Automated template evaluation/explainability tools. Informed by our findings in RQ3, API providers could develop automated evaluation tools to assist LLMapp developers in testing and refining prompt templates. These tools should enable developers to compare outputs from different template patterns for the same inputs, facilitating the identification of optimal designs. Moreover, the tools should analyze the template structure to identify measurable evaluation criteria, such as constraints or output requirements. These evaluations could give scores and explanations based on metrics like content-following and format-following. Additional features, such as model version comparisons and prompt template history tracking, could further enhance the usability of these tools.

4.2 Implications for LLMapp Developers

After an LLMapp’s release, maintaining prompt templates based on user feedback is crucial to ensure high-quality outputs. However, identifying the most effective improvement strategy—such as modifying specific components or adopting new prompt techniques—can be challenging. While switching to a more powerful model may improve performance, it also significantly increases costs, making cost efficiency a critical consideration for the success of LLMApps.

Prompt templates maintenance. Prompt templates should adapt dynamically to enhance user experience, incorporating user feedback and expert reviews for continuous refinement [31, 46]. Analyzing historical usage data, such as input lengths and content types, helps developers optimize placeholders and adjust component positions to prevent key information from being overlooked, especially when handling long inputs (as noted in RQ3). Placeholders should also align with real-world scenarios—for instance, separating background and analytical inputs into distinct placeholders improves clarity and usability. Incorporating metadata placeholders, such as {output_format}, ensures flexibility and robustness, accommodating diverse user needs. Developer and expert reviews further validate refinements, aligning templates with best practices. Additionally, as observed in Section 3.2.3, many prompt templates still use ambiguous placeholders (e.g., approximately 5% of placeholders are named simply as “text”), which lacks meaningful context and can complicate maintenance during LLMapp evolution. Clear, descriptive naming reduces errors, mitigates challenges arising from memory limitations and developer turnover, and ensures long-term software reliability.

Using well-defined prompt templates to strengthen weak LLMs. As demonstrated in our sample testing results in Section 3.3, well-defined prompt templates significantly enhance the instruction-following capability of weaker models (e.g., llama3-70b-8192). In some cases, these templates enable weaker models to achieve performance levels comparable to the best-performing

LLMs (e.g., gpt-4o). For instance, in the long-input experiment discussed in Section 3.3.3, the output quality boost achieved with a well-defined prompt template for llama3-70b-8192 was nearly double that of gpt-4o. This highlights the critical role of carefully designed prompt templates for optimizing weaker models. When selecting foundation models for LLMapps, developers should first consider re-designing prompt templates for the target task if the generation quality falls short of expectations, rather than immediately switching to a more advanced model. Well-designed prompt templates can significantly strengthen the instruction-following abilities of weaker models, therefore contributing to reducing costs, an essential factor for business success.

Trade-offs in In-Context learning. In-context learning has become a widely adopted prompt engineering technique in software engineering research [26]. However, our statistical analysis of component distribution in prompt templates (Table 3) reveals that fewer than 20% of applications in our dataset incorporate few-shot examples in their prompts. Considering the usage of dynamically loaded examples through placeholders (less than 5%), few-shot examples are still not commonly used in prompt templates. Prior research also highlights that clearly defined tasks without examples can sometimes outperform those with few-shot examples in terms of generation quality [55]. This finding likely explains why many LLMapps omit few-shot learning. While few-shot examples can sometimes enhance performance, they are often unnecessary when a well-defined prompt template is used. Moreover, including such examples can introduce drawbacks, such as increased token costs and the risk of semantic contamination. In-context learning is not a one-size-fits-all solution. When designing LLMapps, developers should prioritize refining and optimizing prompt templates to achieve clarity and alignment with task requirements rather than defaulting to few-shot learning.

5 RELATED WORK

Software Engineering for LLM. The emerging field of Software Engineering for Large Language Models (SE4LLM) applies software engineering principles to various stages of LLM development, addressing challenges such as efficiency through system stack optimization [2], model compression via prompt learning [71], and security with multi-round automatic red-teaming [18] and monitoring through modular approaches [21]. Research has also explored interpretability using attention visualization [17], concept-based analysis [48], and subnetwork extraction [67].

Beyond LLM optimization, attention has turned to LLMapps, focusing on post-deployment challenges. Zhao et al. [76] examined LLM app stores, investigating user experience, developer strategies, and ecosystem dynamics while highlighting challenges such as security and privacy. Security issues in the LLMapp ecosystem, including jailbreaking, prompt injection, credential leaks, and inconsistent data provision, have been extensively studied [15, 42, 72], emphasizing the need for robust safeguards.

In contrast to these post-deployment studies, our research focuses on the pre-deployment phase of LLMapp development by analyzing prompt templates—the critical interface between users and LLMs. By examining prompt structure, components, and patterns, we provide actionable insights to help developers design more effective templates, thereby optimizing interactions before model

execution. This work offers foundational support for enhancing both the usability and performance of LLMapps.

Prompt Engineering. Prompt engineering is critical for guiding LLMs, but the variability of natural language poses challenges in achieving clarity and consistency. Recent research has focused on common elements and structural patterns, including standardizing prompt elements into reusable frameworks [64], emphasizing task-specific instructions and formatting [55], and analyzing the effects of component sequencing [12, 43, 77]. Other studies have explored the impact of minor prompt modifications, such as punctuation [61] and word rephrasing [73].

In contrast to studies on general prompt design, our research focuses on systematically analyzing prompt templates used in LLMapps. Building on prior work in prompt analysis [10, 60], we refine the analysis framework by targeting the common components of prompt templates. Unlike general prompts crafted for direct user-LLM interactions, prompt templates designed by LLMapp developers are typically longer and more intricate, often incorporating multiple components to articulate detailed task instructions [66]. A distinctive feature of prompt templates is the inclusion of placeholders [60], which general prompts usually lack. Placeholders allow developers to account for diverse user inputs and scenarios while enhancing the reusability and adaptability of templates. To the best of our knowledge, this is the first study to systematically analyze prompt templates in LLMapps, providing developers with actionable guidance for designing effective, reusable templates that accommodate a broad range of user inputs.

6 CONCLUSION

This paper provides a comprehensive analysis of prompt template structure and composition in LLM-powered applications, analyzing how developers design these templates to optimize LLM’s instruction-following abilities. We construct a dataset of prompt templates from GitHub open-source projects, identifying common components and placeholders in those prompt templates. Through LLM-assisted and human-verified analysis, we analyze the frequently used terms of these components and placeholders, as well as their positions, and further identify several organizational patterns based on the analysis result. Finally, we conduct prompt template testing to evaluate how different patterns influence the instruction-following performance of LLM. These findings offer foundational insights for prompt engineering, guiding the design of robust prompt templates that enhance the quality of LLM outputs across various LLMapps.

ACKNOWLEDGEMENTS

This research was supported by OpenAI through the provision of API credits under OpenAI Researcher Access Program. We appreciate their support in facilitating our study on prompt template analysis.

REFERENCES

- [1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. [arXiv preprint arXiv:2303.08774](https://arxiv.org/abs/2303.08774) (2023).
- [2] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. 2022. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.
- [3] Anthropic. 2024. User Guides - Prompt engineering. <https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/prompt-templates-and-variables>
- [4] Anthropic. 2024. User Guides - Strengthen guardrails. <https://docs.anthropic.com/en/docs/test-and-evaluate/strengthen-guardrails/reduce-hallucinations>
- [5] Eran Avidan and Dror G Feitelson. 2017. Effects of variable names on comprehension: An empirical study. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 55–65.
- [6] Asma Ben Abacha and Dina Demner-Fushman. 2019. A question-entailment approach to question answering. *BMC bioinformatics* 20 (2019), 1–23.
- [7] Diego Carpintero. 2024. wikisearch. <https://github.com/dcarpintero/wikisearch>
- [8] Xinxin Chen, Li Wang, Wei Wu, Qizhi Tang, and Yiyao Liu. 2024. Honest AI: Fine-Tuning "Small" Language Models to Say "I Don't Know", and Reducing Hallucination in RAG. [arXiv abs/2410.09699](https://arxiv.org/abs/2410.09699) (2024). <https://api.semanticscholar.org/CorpusID:273346023>
- [9] Yihan Chen, Benfeng Xu, Quan Wang, Yi Liu, and Zhendong Mao. 2024. Benchmarking large language models on controllable generation under diversified instructions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 17808–17816.
- [10] Google Cloud. 2024. Prompt Design Strategies for Generative AI. <https://cloud.google.com/vertex-ai/generative-ai/docs/learn/prompts/prompt-design-strategies> Accessed: 2024-07-14.
- [11] Valerio Cosentino, Javier Luis, and Jordi Cabot. 2016. Findings from GitHub: methods, datasets and limitations. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 137–141.
- [12] Florin Cuconasu, Giovanni Trappolini, Federico Siciliano, Simone Filice, Cesare Campagnano, Yoelle Maarek, Nicola Tonello, and Fabrizio Silvestri. 2024. The power of noise: Redefining retrieval for rag systems. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 719–729.
- [13] Ross Dawson. 2024. Humans + AI: Prompt Elements. https://rossdawson.com/humans-plus-ai-old/humans-ai-prompt_elements/ Accessed: 2024-10-12.
- [14] Definitive. 2024. openassistants. <https://github.com/definitive-io/openassistants>
- [15] Gelei Deng, Yi Liu, Yuekang Li, Kailong Wang, Ying Zhang, Zefeng Li, Haoyu Wang, Tianwei Zhang, and Yang Liu. 2023. MASTERKEY: Automated Jailbreaking of Large Language Model Chatbots. *Proceedings 2024 Network and Distributed System Security Symposium* (2023). <https://api.semanticscholar.org/CorpusID:259951184>
- [16] Michael Desmond and Michelle Brachman. 2024. Exploring Prompt Engineering Practices in the Enterprise. [arXiv preprint arXiv:2403.08950](https://arxiv.org/abs/2403.08950) (2024).
- [17] Andrea Galassi, Marco Lippi, and Paolo Torroni. 2020. Attention in natural language processing. *IEEE transactions on neural networks and learning systems* 32, 10 (2020), 4291–4308.
- [18] Suyu Ge, Chunting Zhou, Rui Hou, Madien Khabsa, Yi-Chia Wang, Qifan Wang, Jiawei Han, and Yuning Mao. 2023. Mart: Improving llm safety with multi-round automatic red-teaming. [arXiv preprint arXiv:2311.07689](https://arxiv.org/abs/2311.07689) (2023).
- [19] Google Cloud. 2024. Generative AI on Vertex AI - Use prompt templates. <https://cloud.google.com/vertex-ai/generative-ai/docs/learn/prompts/prompt-templates>
- [20] Simon Cornelius Gorissen, Stefan Sauer, and Wolf G Beckmann. 2024. A Survey of Natural Language-Based Editing of Low-Code Applications Using Large Language Models. In *International Conference on Human-Centred Software Engineering*. Springer, 243–254.
- [21] Shubh Goyal, Medha Hira, Shubham Mishra, Sukriti Goyal, Arnav Goel, Niharika Dadu, DB Kirushikesh, Sameep Mehta, and Nishtha Madaan. 2024. LLMGuard: Guarding against Unsafe LLM Behavior. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 23790–23792.
- [22] Lianghong Guo, Yanlin Wang, Ensheng Shi, Wanjun Zhong, Hongyu Zhang, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. When to stop? towards efficient code generation in llms with excess token prevention. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1073–1085.
- [23] hambuger. 2023. Andrew. <https://github.com/hambuger/Andrew>
- [24] Karoline Holter, Juhan Oskar Hennoste, Patrick Lam, Simmo Saar, and Vesal Vojdani. 2024. Abstract debuggers: Exploring program behaviors using static analysis results. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*.
- [25] Matthew Honnibal and Ines Montani. 2017. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. (2017). To appear.
- [26] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.
- [27] Xinyi Hou, Yanjie Zhao, and Haoyu Wang. 2024. On the (In) Security of LLM App Stores. [arXiv preprint arXiv:2407.08422](https://arxiv.org/abs/2407.08422) (2024).
- [28] Hypercerts. 2024. hypercerts. <https://github.com/hypercerts-org/hypercerts>
- [29] International Organization for Standardization. 2019. ISO 8601. <https://www.iso.org/iso-8601-date-and-time-format.html>
- [30] Jina AI. 2023. dev-gpt. <https://github.com/jina-ai/dev-gpt>
- [31] Ishika Joshi, Shimra Shahid, Shreeya Veneti, Manushree Vasu, Yantao Zheng, Yunyao Li, Balaji Krishnamurthy, and Gromit Yeuk-Yin Chan. 2024. CoPrompter: User-Centric Evaluation of LLM Instruction Alignment for Improved Prompt Engineering. [arXiv abs/2411.06099](https://arxiv.org/abs/2411.06099) (2024). <https://api.semanticscholar.org/CorpusID:273963317>
- [32] Shubhra (Santu) Karmaker and Dongji Feng. 2023. TELR: A General Taxonomy of LLM Prompts for Benchmarking Complex Tasks. [arXiv abs/2305.11430](https://arxiv.org/abs/2305.11430) (2023). <https://api.semanticscholar.org/CorpusID:258823169>
- [33] George Katsogiannis-Meimarakis and Georgia Koutrika. 2023. A survey on deep learning approaches for text-to-SQL. *The VLDB Journal* 32, 4 (2023), 905–936.
- [34] kenoharada. 2023. AI-LaBuddy. <https://github.com/kenoharada/AI-LaBuddy>
- [35] Ameya Ketkar, Daniel Ramos, Lazaro Clapp, Raj Barik, and Murali Krishna Ramanathan. 2024. A Lightweight Polyglot Code Transformation Language. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1288–1312.
- [36] Kwon Ko. 2024. chart-llm. <https://github.com/hyungkwonko/chart-llm>
- [37] Kyubyong. 2019. A TensorFlow Implementation of Transformer. <https://github.com/Kyubyong/transformer>
- [38] LAION-AI. 2024. Open-Assistant. <https://github.com/LAION-AI/Open-Assistant>
- [39] LangChain. 2024. Prompt Templates. https://python.langchain.com/docs/concepts/prompt_templates/
- [40] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. Skcoder: A sketch-based approach for automatic code generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2124–2135.
- [41] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. Cctest: Testing and repairing code completion systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1238–1250.
- [42] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Zihao Wang, Xiaofeng Wang, Tianwei Zhang, Yipang Liu, Haoyu Wang, Yan Zheng, et al. 2023. Prompt Injection attack against LLM-integrated Applications. [arXiv preprint arXiv:2306.05499](https://arxiv.org/abs/2306.05499) (2023).
- [43] Yijin Liu, Xianfeng Zeng, Fandong Meng, and Jie Zhou. 2023. Instruction position matters in sequence generation with large language models. [arXiv preprint arXiv:2308.12097](https://arxiv.org/abs/2308.12097) (2023).
- [44] Bei Lu, Raymond YK Lau, Chunping Li, and Yain-Whar Si. 2022. A critical review of state-of-the-art chatbot designs and applications. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 12, 1 (2022), e1434.
- [45] Maanvitha Gongalla. 2024. thinkai. <https://github.com/maanvithag/thinkai>
- [46] Stephen Macneil, Andrew Tran, Joanne Kim, Ziheng Huang, Seth Bernstein, and Dan Mogil. 2023. Prompt Middleware: Mapping Prompts for Large Language Models to UI Affordances. [arXiv abs/2307.01142](https://arxiv.org/abs/2307.01142) (2023). <https://api.semanticscholar.org/CorpusID:259316650>
- [47] Matt Nigh. 2023. ChatGPT3-Free-Prompt-List: A free guide for learning to create ChatGPT3 Prompts. <https://github.com/mattnigh/ChatGPT3-Free-Prompt-List>
- [48] Tuomas Oikarinen, Subhro Das, Lam M Nguyen, and Tsui-Wei Weng. 2023. Label-free concept bottleneck models. [arXiv preprint arXiv:2304.06129](https://arxiv.org/abs/2304.06129) (2023).
- [49] OpenAI. 2024. OpenAI Prompt Engineering Best Practices. <https://platform.openai.com/docs/guides/prompt-engineering> Accessed: 2024-07-14.
- [50] Oracle. 2024. JDK 22 Documentation. <https://docs.oracle.com/en/java/javase/22>
- [51] Kaiser Pister, Dhruva Jyoti Paul, Ishan Joshi, and Patrick Brophy. 2024. PromptSet: A Programmer's Prompting Dataset. In *Proceedings of the 1st International Workshop on Large Language Models for Code*. 62–69.
- [52] puzzle.today. 2024. GPT Store - Write For Me. <https://gptstore.ai/gpts/LJybdvxEb4-write-for-me>
- [53] Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, et al. 2023. Taskweaver: A code-first agent framework. [arXiv preprint arXiv:2311.17541](https://arxiv.org/abs/2311.17541) (2023).
- [54] Murali Krishna Ramanathan, Lazaro Clapp, Rajkishore Barik, and Manu Sridharan. 2020. Piranha: Reducing feature flag debt at uber. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. 221–230.
- [55] Laria Reynolds and Kyle McDonnell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended abstracts of the 2021 CHI conference on human factors in computing systems*. 1–7.

- [56] Alok Saboo. 2024. beets-plexsync. <https://github.com/arsaboo/beets-plexsync>
- [57] Salesforce. 2024. Prompt Builder - Prompt Template Types. https://help.salesforce.com/s/articleView?id=sf.prompt_builder_standard_template_types.htm&type=5
- [58] Abel Salinas and Fred Morstatter. 2024. The butterfly effect of altering prompts: How small changes and jailbreaks affect large language model performance. *arXiv preprint arXiv:2401.03729* (2024).
- [59] Elvis Saravia et al. 2022. Prompt engineering guide. GitHub. URL: <https://github.com/dair-ai/Prompt-Engineering-Guide> (2022). Accessed: 2024-06-01.
- [60] Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Ayush Gupta, Hyojung Han, Sevien Schulhoff, et al. 2024. The Prompt Report: A Systematic Survey of Prompting Techniques. *arXiv preprint arXiv:2406.06608* (2024).
- [61] Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. 2023. Quantifying Language Models' Sensitivity to Spurious Features in Prompt Design or: How I learned to start worrying about prompt formatting. *arXiv preprint arXiv:2310.11324* (2023).
- [62] Stevenic. 2024. AgentM. <https://github.com/Stevenic/agentm-py>
- [63] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
- [64] Ming Wang, Yuanzhong Liu, Xiaoming Zhang, Songlian Li, Yijie Huang, Chi Zhang, Daling Wang, Shi Feng, and Jigang Li. 2024. LangGPT: Rethinking Structured Reusable Prompt Design Framework for LLMs from the Programming Language. *ArXiv abs/2402.16929* (2024). <https://api.semanticscholar.org/CorpusID:268032985>
- [65] Yu Wang, Nedim Lipka, Ryan A Rossi, Alexa Siu, Ruiyi Zhang, and Tyler Derr. 2024. Knowledge graph prompting for multi-document question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 19206–19214.
- [66] Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Anjana Arunkumar, Arjun Ashok, Arut Selvan Dhanasekaran, Atharva Naik, David Stap, et al. 2022. Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks. *arXiv preprint arXiv:2204.07705* (2022).
- [67] Yulong Wang, Hang Su, Bo Zhang, and Xiaolin Hu. 2020. Interpret neural networks by extracting critical subnetworks. *IEEE Transactions on Image Processing* 29 (2020), 6707–6720.
- [68] weaviate. 2024. Verba. <https://github.com/weaviate/Verba>
- [69] Simon Weber and Jiebo Luo. 2014. What makes an open source code popular on git hub?. In *2014 IEEE International Conference on Data Mining Workshop*. IEEE, 851–855.
- [70] Congying Xia, Chen Xing, Jiangshu Du, Xinyi Yang, Yihao Feng, Ran Xu, Wengpeng Yin, and Caiming Xiong. 2024. FOFO: A Benchmark to Evaluate LLMs' Format-Following Capability. *arXiv preprint arXiv:2402.18667* (2024).
- [71] Zhaozhuo Xu, Zirui Liu, Beidi Chen, Yuxin Tang, Jue Wang, Kaixiong Zhou, Xia Hu, and Anshumali Shrivastava. 2023. Compress, then prompt: Improving accuracy-efficiency trade-off of llm inference with transferable prompt. *arXiv preprint arXiv:2305.11186* (2023).
- [72] Chuan Yan, Ruomat Ren, Mark Huasong Meng, Liuhuo Wan, Tian Yang Ooi, and Guangdong Bai. 2024. Exploring chatgpt app ecosystem: Distribution, deployment and security. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1370–1382.
- [73] Adam Yang, Chen Chen, and Konstantinos Pitas. 2024. Just rephrase it! Uncertainty estimation in closed-source language models via multiple rephrased queries. *arXiv preprint arXiv:2405.13907* (2024).
- [74] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1703–1726.
- [75] JD. Zamfirescu-Pereira, Richmond Y. Wong, Bjoern Hartmann, and Qian Yang. 2023. Why Johnny Can't Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (2023). <https://api.semanticscholar.org/CorpusID:258217984>
- [76] Yanjie Zhao, Xinyi Hou, Shenao Wang, and Haoyu Wang. 2024. Llm app store analysis: A vision and roadmap. *arXiv preprint arXiv:2404.12737* (2024).
- [77] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate before use: Improving few-shot performance of language models. In *International conference on machine learning*. PMLR, 12697–12706.

Received 2025-XX-XX; accepted 2025-XX-XX; revised xxx; revised xxx; accepted xxx