Towards a Catalog of Prompt Patterns to Enhance the Discipline of Prompt Engineering

Douglas C. Schmidt, Jesse Spencer-Smith, Quchen Fu, and Jules White
Dept. of Computer Science, Vanderbilt University; email: {douglas.c.schmidt, jesse.spencer-smith, quchen.fu, jules.white}@vanderbilt.edu

Abstract

The rapid advent of Large Language Models (LLMs), such as ChatGPT and Claude, is revolutionizing various fields, from education and healthcare to the engineering of reliable software systems. These LLMs operate through "prompts," which are natural language inputs that users employ to query and leverage the models' capabilities. Given the novelty of LLMs, the understanding of how to effectively use prompts remains largely anecdotal, based on isolated use cases. This fragmented approach limits the reliability and utility of LLMs, especially when they are applied in mission-critical software environments. To harness the full potential of LLMs in such crucial contexts, therefore, we need a systematic, disciplined approach to "prompt engineering" that guides interactions with and evaluations of these LLMs.

This paper provides several contributions to research on LLMs for reliable software systems. First, it provides a holistic perspective on the emerging discipline of prompt engineering. Second, it discusses the importance of codifying "prompt patterns" to provide a sound foundation for prompt engineering. Third, it provides examples of prompt patterns that improve human interaction with LLMs in the context of software engineering, as well as other domains. We conclude by summarizing ways in which prompt patterns play an essential role in providing the foundation for prompt engineering.

1 Introduction

Large language models (LLMs) [1, 2] with conversational interfaces, such as ChatGPT [3], are generating and reasoning about art, music, essays and computer programs. Startups using LLMs are attracting significant funding [4] and existing software is being enhanced using LLMs. The rapid uptake of these tools underscores the transformational—and disruptive—impact LLMs are having on society, research, and education. However, little disciplined knowledge about chat-adapted LLMs, their capabilities, and their limitations exist.

LLMs provide new computational models with unique programming and interaction paradigms and greatly expanded capabilities. Anyone with an Internet connection and web browser can instantly access vast intelligent computational abilities, such as explaining complex topics; reasoning about diverse data sets; designing, implementing, and testing computer software; and simulating complex systems. LLMs are programmed through prompts, which are natural language instructions provided to the LLM [5], such as asking it to answer a question or write an essay. These common examples of prompts, however, do not reveal the much more sophisticated computational abilities LLMs possess.

Harnessing the potential of LLMs in productive and ethical ways requires a systematic focus on *prompt engineering*, which is an emerging discipline that studies interactions with—and programming of—emerging LLM computational systems to solve complex problems via natural language interfaces. We contend that an essential component of this discipline are *Prompt Patterns* [6], which are similar to software patterns [7], but focus on capturing reusable solutions to problems faced when interacting with LLMs. Such patterns elevate the study of LLM interactions from individual *ad hoc* examples, to a more reliable and repeatable engineering discipline that formalizes and codifies fundamental prompt structures, their capabilities, and their ramifications.

This paper presents portions of our ongoing efforts to codify a catalog of domain-independent patterns to show the need for more research on prompt patterns and prompt engineering. We present these patterns in the context of engineering software-reliant systems, but they are applicable in many other domains.

The remainder of this paper is organized as follows: Section 2 gives an overview of the emerging discipline of prompt engineering; Section 3 describes portions of a catalog of prompt patterns that we are codifying; and Section 4 presents concluding remarks and lessons learned from our work on prompt patterns for prompt engineering thus far.

2 Towards a Discipline of Prompt Engineering

This section gives an overview of prompt engineering, focusing on its definition, value for both computer science (CS) and non-CS professionals, and the need for a holistic approach. For CS professionals we emphasize prompt engineering's role in enhancing AI interactions and accelerating prototyping, whereas

Reprinted from Ada User Journal, Vol. 44(3), September 2023, with permission. Copyright is held by the author/owner(s).

for non-CS professionals it serves as a gateway to computational problem-solving without requiring traditional programming skills. Lastly, we advocate for considering the same type of quality attributes for prompt engineering as we do for software engineering.

2.1 What is Prompt Engineering?

Prompt engineering is the science and art of designing, formatting, and optimizing conversational prompts to better guide the discourse with AI or machine learning models. It involves crafting of stimuli or instructions to evoke specific responses from AI systems. With the increased use of AI platforms that respond to user queries, establishing a discipline of prompt engineering has become increasingly important.

The scope of prompt engineering encompasses a wide range of domains, including AI chatbots, AI customer service agents, voice-first applications, and other AI interaction interfaces. It plays a crucial role in tuning the model performance, enhancing the quality of interaction, and achieving user satisfaction. It spans both understanding the technical capabilities of AI models and the nuances of human communication.

Users of AI models need to understand the strength and weaknesses of AI models they interact with and should hone their creative capability to formulate prompts that evoke the desired response. Prompt engineering thus provides a bridge between the increasingly sophistication of AI models and the need for human-like interactions that appeal to users.

2.2 The Value of Prompt Engineering for Computer Science (CS) Professionals

For CS professionals, prompt engineering offers advantages that extend beyond the basics of code generation and code summarization. Programmers are generally proficient in coding and well-versed in the syntax and semantics of conventional programming languages, such as Java, C/C++, or Python. However, prompt engineering introduces a new paradigm that enables rapid prototyping and concept testing without the need to write code manually. It thus serves as a complementary tool that can expedite iterative and incremental development processes, enabling software engineers to rapidly sketch out algorithms, models, or systems using natural language. This capability accelerates the transition from idea to implementation, thereby saving time and effort.

Prompt engineering can also enhance software quality and coding practices, *e.g.*, by serving as a 'first-pass filter' to evaluate the feasibility of algorithms or system architectures before delving into implementation details. Articulating complex computational problems in natural language can enable developers to identify potential pitfalls or inefficiencies more rapidly. Emphasizing clarity in initial stages of the software development life-cycle allows a broader range of stakeholders (including architects, systems engineers, produce managers, and end-users) to create more robust, efficient, and maintainable software-reliant systems over the life-cycle by avoiding costly mistakes and architectural flaws that are cumbersome and costly to rectify later.

Prompt engineering can be an effective tool for collaborative work within the CS community. Prompts can be collected into libraries of "prompt templates", thereby simplifying the sharing of algorithms, ideas, and problem-solving approaches without needing to understand the detailed intricacies of software repositories. Since prompts are often more easily understood and modified by a range of (non-developer) stakeholders, they facilitate more inclusive and interactive development environments. For example, team members can propose modifications, tune parameters, or even reimplement subsystems without the steep learning curve often associated with understanding lower-level programs.

Prompt engineering also benefits educators and mentors within the field of computer science. Educators can use it to introduce complex computational concepts to students in a more intuitive and accessible manner. Similarly, it can enable more advanced students or junior developers to transition from conceptual understanding to practical application. By offering a natural language-based approach to problem articulation and solution, prompt engineering acts as an educational accelerator, easing the path from theory to practice.

2.3 The Value of Prompt Engineering for Non-CS Professionals

Prompt engineering can be viewed as a form of "programming" via natural language, which helps to democratize the application of computational problem-solving across a range of disciplines and professions. When used effectively, this approach can bypass conventional barriers set by the need to learn conventional programming languages, such as Java, C/C++, or Python. Mastering these structured programming language traditionally involved understanding their syntax and semantics, which can incur a daunting and time-consuming learning curve for non-CS professionals.

Moreover, the primary interest professionals in fields like chemistry, biology, physics, the social sciences, and the humanities often lies not in becoming programming experts, but rather in leveraging computational resources to advance their research or solve domain-specific problems. In such contexts, prompt engineering helps to shift the focus from mastering coding to mastering problem-solving. Using natural language as the medium reduces barriers to entry, thereby allowing a broader audience to employ computational tools in their respective domains more effectively.

The potential impact of shifting from conventional programming to problem-solving with LLMs is significant. Computation today is often limited to those with specialized training in CS or programming. With prompt engineering, however, experts in diverse fields ranging from analysis of ancient documents to radiology can harness the power of computational methods to drive innovation and discovery. In particular, they can articulate complex problems using familiar terminology and get computational assistance without the learning curve associated with conventional programming.

Prompt engineering is particularly relevant in interdisciplinary work, where insights from multiple fields are crucial. In these contexts, it serves as a bridge that facilitates a more holistic approach to problem-solving, unifying various areas of expertise under the umbrella of computational capability.

2.4 Towards a Holistic View of Prompt Engineering

The notion that prompt engineering is merely a passing trend [8], soon to be eclipsed by increasingly sophisticated LLMs, is a simplistic and short-sighted perspective. This view reduces prompt engineering to a set of "tricks" designed to navigate the current limitations of LLMs. However, this view overlooks the inherent complexities and nuances of natural language, which necessitates a systematic approach to interaction. Unlike traditional programming languages, natural language lacks rigorously defined semantics, requiring a disciplined method like prompt engineering to ensure the effective use of LLMs in software-reliant systems.

Far from being a stopgap measure, prompt engineering should be integrated holistically into all phases of the software development life-cycle. In traditional software development, professionals address a broad range of considerations beyond just coding, including requirements specification, configuration management, testing, and version control. In much the same way, the discipline of prompt engineering must also address these considerations, especially in mission-critical systems where failure is not an option.

A focus on quality attributes across the life-cycle is essential for the broader application of LLMs in robust, long-lived software-reliant systems. Current uses of LLMs are often localized and tactical, not integrated into systems intended to endure for decades. As LLMs evolve, prompts that were once reliable may no longer function as intended. The same diligence applied to traditional software engineering—centered on maintainability, reliability, and compatibility—must therefore be applied to the domain of prompt engineering.

Failing to adopt a comprehensive view of prompt engineering risks limiting the application of LLMs to trivial or short-term projects. To unlock the full potential of these advanced models in shaping future software-reliant systems, therefore, a focus on quality attributes and a holistic methodology are not just advantageous, they are essential. This dependency underscores the need for a more mature and systematic discipline of prompt engineering that goes beyond mere prompt crafting and becomes an integral part of modern software engineering in the age of LLMs.

3 Towards a Catalog of Prompt Patterns

This section builds upon and briefly summarizes our prior work on prompt patterns [6]. Prompt patterns use a similar format to classic software patterns, with slight modifications to match the context of output generation with LLMs.

Organizing a catalog of prompt patterns into easily digestible categories helps users interact with and program LLMs more effectively. Table 1 outlines the classification of the patterns we implemented and tested with ChatGPT discussed in this paper. As shown in this table, there are four categories of prompt patterns in the classification framework presented in this paper: *Software Requirements*, *Interaction*, *Prompt Improvement*, and *Error Identification*. The *Software Requirements* patterns are a specialized subset, as discussed next.

Table 1: Classifying Prompt Patterns

Pattern Category	Prompt Pattern
Software Requirements	*Requirements Elicitation Facilitator* *Unambiguous Requirements Interpreter*
Interaction	*Game Play*
Prompt Improvement	*Question Refinement*
Error Identification	*Reflection*

3.1 Patterns as an Abstraction for Derivation of New Patterns

A benefit of patterns is that they can serve as an abstraction for specialization and adaptation to different domains. In prior work [6] we codified the *Flipped Interaction* pattern, which directs the LLM to ask the user questions until it obtains enough information to achieve a particular goal. The structure of this pattern is as follows:

Contextual Statements
I would like you to ask me questions to achieve X
You should ask questions until this condition is met or to achieve this goal (alternatively, forever)
(Optional) ask me the questions one at a time, two at a time, etc.

The *Flipped Interaction* pattern forms a broader abstraction that can be tailored and specialized to address different aspects of requirements gathering in software engineering, thereby leading to the creation of other, more context-specific patterns. Software engineering requirements' needs often call for such specialized patterns, which extend the parent pattern by integrating new attributes and focusing on specific tasks, such as requirements elicitation, requirements ambiguity resolution, requirements discrepancy analysis, and requirements traceability. Similar to how a super class in object-oriented programming can be inherited and specialized to cover different uses-cases, prompt patterns can act as abstractions for derivations of new prompt patterns. In this case, *Flipped Interaction* acts as a super pattern for flipped interactions focusing on requirements elicitation and management.

Sections 3.2 and 3.3 introduce patterns derived from the *Flipped Interaction* pattern that address more specific areas within requirements elicitation and management, thereby transforming the interaction's goal to achieve more specificity. Section 3.2 describes the *Requirements Elicitation Facilitator* pattern, which refines requirements through focused interaction, and Section 3.3 describes the *Unambiguous Requirements Interpreter* pattern, which reduces ambiguity in requirements through targeted questioning.

3.2 Requirements Elicitation Facilitator

3.2.1 Intent & Context

The *Requirements Elicitation Facilitator* pattern enables an LLM to cooperatively ask questions or propose scenarios, which motivate users to bring forward their implicit requirement expectations. The objective is to get sufficient insight into the subjective nature of their requirements and fill up possible communication gaps. This pattern is particularly beneficial when dealing with broad context applications.

In a context whereby many details are implicit or merely touched upon, using the *Requirements Elicitation Facilitator* pattern will guide an LLM into a systematic dialogue with the users, thereby aiding in the step-wise refinement of the requirements. The discussion focuses on ensuring clarity of notions, resolving contradictions if any, and communicating the intended functionalities/schema.

3.2.2 Motivation

The process of requirements elicitation is often tedious and time-consuming and filled with uncertainties. The process involves, among many other things, capturing a description of what the system should do, the desired behavior in different states, and operational constraints. This process is iterative, requiring involvement from different stakeholders and significant conversation to ensure that all parties are fully expressing their needs and vision for the system. Language model-driven approaches can facilitate this process by helping to direct the questioning and discussion to gather, analyze, and validate the requirements, thereby saving effort and time. This process is one of the most critical stages in software development, since it forms the basis on which the proposed system is built.

3.2.3 Structure & Key Ideas

Typical contextual statements for this pattern include the following:

Contextual Statements
I am creating the requirements for a software system Y using requirement format Z.
(Optionally) I am working on requirements for aspect Q of the system.
Ask me questions to help generate requirements for the system.
After each question, 1) based on my answer, generate the requirement in format Z and then 2) ask me the next question.
Keep asking me questions until stop condition V.
Ask me the first question.

3.2.4 Example Implementation

Examples of prompts that use the *Requirements Elicitation Facilitator* pattern might include:

> "I am creating the requirements for a web application that allows users to share ChatGPT prompts using user stories as the format. Ask me questions to help generate requirements for the system. After each question, 1) based on my answer, generate the requirement as a user story and then 2) ask me the next question. Keep asking me questions until I tell you to stop. Ask me the first question."

> "I am creating the requirements for a web application to help users refine their ChatGPT prompts by suggesting improved versions. Ask me questions to help generate requirements for the system. After each question, 1) based on my answer, generate the requirement as a user story and then 2) ask me the next question. Keep asking me questions until you have enough information to generate a skeleton of the application in Python with Django. Ask me the first question."

3.2.5 Discussion

The *Requirements Elicitation Facilitator* pattern can be viewed as a specialized derivation of the *Flipped Interaction* pattern where the interaction's goal is specifically to refine the understanding of system requirements. This derived pattern employs additional attributes to the LLM's interrogation by focusing on vital aspects of requirements elicitation such as defining the system functionalities, clearing notions, and resolving contradictions. It prescribes a strategic interaction leading to clearer and finely-tuned requirements.

An important aspect of this pattern is defining the 'stop condition' clearly. In situations where the limitations or contexts are not explicitly mentioned, an LLM may take a 'shot in the dark' approach and may prompt you with questions that may or may not be relevant, rendering the interaction less effective. In the second example above, the stop condition explicitly focuses on generating a skeleton application, which can aid the LLM in directing questioning. Another possibility is to include a summary of the requirements captured to help avoid asking duplicate questions.

However, providing more control over the questions and feedback can create a more interactive and engaging experience for users who are looking for a more robust and comprehensive requirements elicitation. It can help bridge the gap between high-level visions and concrete requirements, clarifying misconceptions and better aligning all stakeholders.

3.3 Unambiguous Requirements Interpreter

3.3.1 Intent & Context

The *Unambiguous Requirements Interpreter* pattern provides an LLM with a subset of requirements that will fit within its context window and instructs it to ask specific questions to users about ambiguous requirements and help them rephrase these requirements in a more explicit and clear way. This pattern encourages users to clarify any potential misunderstanding related to the requirements, hence reducing ambiguity related issues. The pattern should be applied in an interactive session where all participants in a team see the questions that are asked and feedback generated by the LLM.

3.3.2 Motivation

A key challenge in software projects is that ambiguous requirements can lead to the development of software with different functionality than was desired by the stakeholders. Ambiguity can lead to miscommunication and invalid assumptions, which commonly creates delays, cost overruns, and software project failures. Clear requirements are even more important for geographically distributed teams that may not have enough face-to-face discussion to have a shared mental picture of the project goals.

LLMs can help offer a potential solution to this by clarifying any ambiguous requirements using systematized questioning and solution-based dialogue, particularly when used jointly be teams in a discussion. A well-defined prompt pattern can aid in identifying potentially ambiguous requirements and extracting the implicit assumptions behind an ambiguous requirement, reducing the chance of misunderstanding and helping reduce misinterpretation.

3.3.3 Structure & Key Ideas

Typical contextual statements for this pattern include the following:

Contextual Statements
A subset of the requirements for my system, each phrased using format X, is below.
Requirements...
First, list two requirements that are potentially contradictory based on their current wording and list them.
Next, explain why these two requirements might be contradictory based on the current wording.
Then, ask me about the intent of the two requirements until you have enough information to propose a refined version of each requirement that eliminates potential ambiguity and conflict.

3.3.4 Example Implementation

An example of a prompt that uses this pattern:

> “A subset of the requirements for my system, each phrased as user stories, is below.
> ——— <list of user stories> ———
> First, list two requirements that are potentially contradictory based on their current wording and list them. Next, explain why these two requirements might be contradictory based on the current wording. Then, ask me about the intent of the two requirements until you have enough information to propose a refined version of each requirement that eliminates potential ambiguity and conflict. ”

3.3.5 Discussion

The *Unambiguous Requirements Interpreter* pattern can also be viewed as a specialized derivation of the *Flipped Interaction* pattern where the goal is to reduce ambiguity in the requirements present. This pattern represents another effective example of how we borrow the fundamental construct of the *Flipped Interaction* pattern to target a more specific goal. As before, additional attributes are introduced to the line of questioning to explicitly target ambiguous requirements and make them more explicit and clear.

One crucial aspect of this pattern is having stakeholders go through the questioning together until the ambiguity is eliminated. The most important part is to have the LLM direct team-based discussion around the requirements. The language model should continuously ask the team relevant questions until it receives explicit answers that cannot be interpreted differently by different stakeholders.

Although this approach can effectively deal with ambiguous requirements in many situations, there are scenarios where it might struggle. For instance, when dealing with complex technical requirements, the language model may not be able to ask pertinent questions due to the lack of intrinsic knowledge. Thus, while this pattern can provide significant benefits, care must be taken to ensure that it is suitable for the particular domain and provided with sufficient context to provide relevant questioning.

3.4 The Game Play Pattern

3.4.1 Intent and Context

The *Game Play* pattern creates a "game" centered around a specific topic, where the LLM guides the game play. This pattern is particularly effective when the rules of the game are relatively limited in scope, but the content for the game is wider in scope. Users can specify a limited set of rules and the LLM can then automate generation of bodies of content for game play.

3.4.2 Motivation

You want an LLM to generate scenarios or questions involving specific topic(s) and require users to apply problem solving or other skills to accomplish a task related to the scenario. Generating all game content manually is too time consuming, however, so you would like the LLM to apply its knowledge of the topic to guide the generation of content.

3.4.3 Structure and Key Ideas

Typical contextual statements for this pattern include the following:

Contextual Statements
Create a game for me around X
One or more fundamental rules of the game

The first statement, instructs the LLM to create a game and provides the important scoping of the game to a topic area. This pattern allows users to create games by describing the rules of the game, without having to determine the content of the game. The more specific the topic, typically the more novel and interesting the game play.

The second statement introduces the game rules to the LLM, which must fit within the capabilities of the LLM. Textual games that rely on input and output text sequences work best. A key attribute of this pattern is that the input text can be rich and expressive, which can lead to interesting interfaces for the game. For example, users might express actions in the game as scripts dictating a sequence of complex actions, such as "get a listing of all network activity and check it for anomalies", which go beyond the scope of multiple choice or short answer inputs. Each rule should be provided as a separate statement regarding some aspect of the game.

3.4.4 Example Implementation

A sample cybersecurity game prompt is shown below:

> "We will play a cybersecurity game where you pretend to be a Linux terminal for a computer compromised by an attacker. When I type in a command, you will output the corresponding text the Linux terminal would produce. I will use commands to try and figure out how the system was compromised. The attack should have done one or more of the following things: (1) launched new processes, (2) changed files, (3) opened new ports to receive communication, (4) created new outbound connections, (5) changed passwords, (6) created new user accounts, or (7) read and stolen information. To start the game, print a scenario of what happened that led to my investigation and make the description have clues I can use to get started."

This cybersecurity game prompt combines a number of patterns, including *Game Play* and *Persona*. The Linux terminal persona that is chosen for the game has been discussed in many Internet posts [9]. This game further customizes the discussed Linux persona to act as a Linux terminal of a computer compromised by a cyberattack. ChatGPT responds to Linux commands with appropriate terminal outputs and synthesizes a faux file system that has clues to a cyberattack.

Sample output from this game with ChatGPT-4 after typing in "ls -alt", the Linux command to generate a list of files in the current directory is shown below (some columns have been removed to enhance formatting):

```
total 56
 5 user  4096 Feb 16 14:31 .
43 user  4096 Feb 16 14:30 ..
 1 user 18881 Feb 16 14:31 .bash_history
 1 user  1178 Aug 15  2022 .bashrc
 1 user   220 Aug 15  2022 .bash_logout
 3 user  4096 Feb 16 14:25 .gnupg
 1 user  4041 Feb 15 17:52 README.md
 2 user  4096 Feb  1 11:35 project
```

When the user then typed the command "cat .bash_history", to display the contents of the log file showing the commands run by the user, ChatGPT responded with:

```
cd /var/www/html
sudo rm -rf *
sudo wget https://example.com/backdoor.php
sudo chmod 777 backdoor.php
exit
ls -alt
ps -ef
netstat -tulnp
```

In the output above, ChatGPT generated a faux log file with realistic commands for deleting the data being served by the web server in `/var/www/html` and replacing the content with a backdoor into the system.

3.4.5 Discussion

This pattern can be combined with the *Persona*, *Infinite Generation*, and *Visualization Generator* patterns [6]. For example, the cybersecurity game uses the *Persona* pattern so the LLM can masquerade as a Linux terminal. For a network security game, the *Visualization Generator* can be employed to visualize the network topology and traffic flows.

3.5 The Reflection Pattern

3.5.1 Intent and Context

The *Reflection* pattern asks an LLM to explain the rationale behind given answers to the user automatically. This pattern allows users to better assess the output's validity, as well as inform users how an LLM arrived at a particular answer. It can also clarify any points of confusion, uncover underlying assumptions, and reveal gaps in knowledge or understanding.

3.5.2 Motivation

LLMs can (and often do) make mistakes. Moreover, users may not understand why an LLM produces particular output and how to adapt their prompt to solve a problem with the output. By asking LLM to explain the rationale of its answers automatically, however, users can gain a better understanding of how the LLM processes the input, what assumptions it makes, and what data it draws upon.

LLMs may sometime provide incomplete, incorrect, or ambiguous answers. Reflection is an aid to help address these shortcomings and ensure the information provided by LLM is as accurate. This pattern also helps users debug their prompts and determine why they are not getting results that meet expectations. The *Reflection* pattern is particularly effective for exploring topics that (1) can be confused with other topics or (2) may have nuanced interpretations, so it is essential to know the precise interpretation used by an LLM.

3.5.3 Structure and Key Ideas

Typical contextual statements for this pattern include the following:

Contextual Statements
Whenever you generate an answer
Explain the reasoning and assumptions of your answer
(Optional) ...so that I can improve my question

The first statement is requesting that, after generating an answer, the LLM should explain the reasoning and assumptions behind the answer. This statement helps the user understand how the LLM arrived at the answer and can help build trust in the model's responses. The prompt includes the statement that the purpose of the explanation is for the user to refine their question. This additional statement gives the LLM the context needed to better tailor its explanations to the specific purpose of assisting the user produce follow-on questions.

3.5.4 Example Implementation

This example tailors the prompt to the domain of providing answers related to code:

> "When you answer, explain the reasoning and assumptions of your software framework selections using specific examples or evidence with associated code samples to support your answer of why a framework is the best selection for the task. Moreover, address any potential ambiguities or limitations in your answer, in order to provide a more complete and accurate response."

The pattern is further customized to instruct the LLM that it should justify its selection of software frameworks, but not necessarily other aspects of the answer. In addition, the user dictates that code samples should be used to help explain the motivation for selecting the specific software framework.

3.5.5 Discussion

The *Reflection* pattern may be ineffective for users who do not understand the topic area being discussed. For example, a highly technical question by a non-technical user may result in a complex rationale for an answer the user cannot fathom. As with other prompt patterns, the output may include errors or inaccurate assumptions included in the explanation of the rationale that the user may not be able to spot. This pattern can be combined with the *Fact Check List* [6] to help address this issue.

3.6 The Question Refinement Pattern

3.6.1 Intent and Context

The *Question Refinement* pattern engages the LLM in the prompt engineering process to ensure an LLM always suggests potentially better or more refined questions users could ask instead of their original question. By applying this pattern, the LLM can aid users in finding the right questions to ask to arrive at accurate answers. In addition, an LLM may help users find the information or achieve their goal in fewer interactions than if users employed conventional "trial and error" prompting.

3.6.2 Motivation

If user asks questions, they may not be experts in the domain and may not know the best way to phrase the question or be aware of additional information helpful in phrasing the question. LLMs will often state limitations on the answer they provide or request additional information to help them produce a more accurate answer. An LLM may also state assumptions it made in providing the answer. The motivation is that this additional information or set of assumptions could be used to generate a better prompt. Rather than requiring the user to digest and rephrase their prompt with the additional information, the LLM can directly refine the prompt to incorporate the additional information.

3.6.3 Structure and Key Ideas

Typical contextual statements for this pattern include the following:

Contextual Statements
Within scope X, suggest a better version of the question to use instead
(Optional) prompt me if I would like to use the better version instead

The first contextual statement in the prompt asks the LLM to suggest a better version of a question within a specific scope. This scoping ensure that (1) not all questions are automatically reworded or (2) they are refined with a given goal. The second contextual statement is meant for automation and allows users to apply the refined question without copy/pasting or manually enter it. This prompt can be further refined by combining it with the *Reflection* pattern discussed above, which allows the LLM to explain why it believes the refined question is an improvement.

3.6.4 Example Implementation

> "From now on, whenever I ask a question about a software artifact's security, suggest a better version of the question to use that incorporates information specific to security risks in the language or framework that I am using instead and ask me if I would like to use your question instead."

In the context of the example above, the LLM will use the *Question Refinement* pattern to improve security-related questions by asking for or using specific details about the software artifact and the language or framework used to build it. For instance, if a developer of a Python web application with FastAPI asks ChatGPT "How do I handle user authentication in my web application?", the LLM will refine the question by taking into account that the web application is written in Python with FastAPI. The LLM then provides a revised question that is more specific to the language and framework, such as "What are the best practices for handling user authentication securely in a FastAPI web application to mitigate common security risks, such as cross-site scripting (XSS), cross-site request forgery (CSRF), and session hijacking?"

The additional detail in the revised question is likely to not only make the user aware of issues they need to consider, but lead to a better answer from the LLM. For software engineering tasks, this pattern could also incorporate information regarding potential bugs, modularity, or other code quality considerations. Another approach would be to refine questions so the generated code cleanly separates concerns or minimizes use of external libraries, such as:

> Whenever I ask a question about how to write some code, suggest a better version of my question that asks how to write the code in a way that minimizes my dependencies on external libraries.

3.6.5 Discussion

The *Question Refinement* pattern helps bridge the gap between the user's knowledge and the LLM's understanding, thereby yielding more efficient and accurate interactions. One risk of this pattern is its tendency to rapidly narrow the questioning by the user into a specific area that guides the user down a more limited path of inquiry than necessary. Such narrowing may cause users to miss important "bigger picture" information. One solution is to provide additional scope to the pattern prompt, such as "do not scope my questions to specific programming languages or frameworks."

Combining the *Question Refinement* pattern with other patterns also helps overcome arbitrary narrowing or limited targeting of refined questions. In particular, combining this pattern with the *Cognitive Verifier* pattern [10] enables an LLM to produce a series of follow-up questions that refine the original question. For example, in the following prompt the *Question Refinement* and *Cognitive Verifier* patterns are applied to ensure better questions are posed to the LLM:

> "From now on, whenever I ask a question, ask four additional questions that would help you produce a better version of my original question. Then, use my answers to suggest a better version of my original question."

As with many prompt patterns that allow an LLM to generate new questions using its knowledge, the LLM may introduce unfamiliar terms or concepts to the user. One way to address this issue is to include a statement that the LLM should explain any unfamiliar terms it introduces into the question. A further enhancement of this idea is to combine the *Question Refinement* pattern with the *Persona* pattern so the LLM flags terms and generates definitions that assume a particular level of knowledge, such as this example:

> "From now on, whenever I ask a question, ask four additional questions that would help you produce a better version of my original question. Then, use my answers to suggest a better version of my original question. After the follow-up questions, temporarily act as a user with no knowledge of AWS and define any terms that I need to know to accurately answer the questions."

LLMs can produce factual inaccuracies, just like humans. A risk of this pattern is that inaccuracies are introduced into refined questions. This risk may be mitigated, however, by combining the *Fact Check List* pattern [6] to enable users to identify possible inaccuracies and the *Reflection* pattern to explain the reasoning behind question refinement.

4 Concluding Remarks

Prompt engineering is an emerging discipline that shifts the emphasis from programming with conventional structured languages (such as Python, Java, and C++) to problem-solving using natural language to interact with AI models, such as large language models (LLMs) like ChatGPT-4 and Claude. In this context, "programming" tasks are expressed as prompts that guide the behavior of AI models, thereby encouraging the exploration of creative and innovative strategies over applying traditional programming methods and tools.

Our work applying LLMs in engineering software-reliant systems has yielded the following lessons learned:

- **It is essential to move beyond *ad hoc* prompt practices** – Current discussions of LLM prompts and prompt engineering are based largely on individual *ad hoc* use cases, i.e., *i.e.* the same basic prompt examples are replicated in different variations and evaluated as if they are new ideas, such as these examples [11] replicating the *Persona Pattern* outlined in our prior work. The limitations with the current state-of-the-practice are thus akin to discussing the specifics of individual software programs without identifying key design and architectural patterns these systems are based on.
- **Codifying prompt patterns provides a sound foundation for prompt engineering** - The focus on prompt patterns elevates the study of LLMs to view them more appropriately as a new computer architecture with an instruction set based on natural language. Prompt patterns define the instruction set, where as individual prompt examples are one-off programs. By documenting the instruction set for this radically new computing architecture via patterns we can reason about LLM technologies more effectively and teach others to tap into these capabilities more effectively.
- **Importance of archetypal solutions** – Prompt patterns provide foundational elements for prompt engineering by serving as proven solutions to recurrent problems and accelerating problem-solving across various stages of the software life-cycle. Moreover, these patterns facilitate knowledge transfer among collaborative teams, enriching the discipline of prompt engineering.
- **Prompt patterns can enhance reuse of effective LLM interactions** – The process of deriving specialized patterns from the *Flipped Interaction* pattern mirrors the abstraction process in object-oriented programming. Just as specialized subclasses with distinct attributes can be derived from a superclass and create more specific subclasses we utilize

the more general *Flipped Interaction* pattern and tailor it to suit our specific needs, adding attributes that target specific objectives in the context of requirements elicitation and management. Such usage of abstraction enhances the reusability and effectiveness of LLM prompts, highlighting the usefulness of the concept of patterns in the domain of LLMs.

This shift towards prompt patterns not only refines the effectiveness of LLMs but also ensures that they can be harnessed more reliably and ethically to develop and assure software-reliant systems. The prompt patterns presented in this paper were refined and tested using ChatGPT-3.5 and ChatGPT-4.

References

[1] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, *et al.*, “On the opportunities and risks of foundation models,” *arXiv preprint arXiv:2108.07258*, 2021.

[2] C. Zhou, Q. Li, C. Li, J. Yu, Y. Liu, G. Wang, K. Zhang, C. Ji, Q. Yan, L. He, *et al.*, “A comprehensive survey on pretrained foundation models: A history from bert to chatgpt,” *arXiv preprint arXiv:2302.09419*, 2023.

[3] Y. Bang, S. Cahyawijaya, N. Lee, W. Dai, D. Su, B. Wilie, H. Lovenia, Z. Ji, T. Yu, W. Chung, *et al.*, “A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity,” *arXiv preprint arXiv:2302.04023*, 2023.

[4] Jeremy Kahn, “A wave of A.I. experts left Google, DeepMind, and Meta—and the race is on to build a new, more useful generation of digital assistant.” `https://fortune.com/2022/07/05/a-i-digital-assistants-adept-eye-on-ai`. [Online; accessed 19-Oct-2023].

[5] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, “Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing,” *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[6] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, “A prompt pattern catalog to enhance prompt engineering with chatgpt,” *arXiv preprint arXiv:2302.11382*, 2023.

[7] E. Gamma, R. Johnson, R. Helm, R. E. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[8] Oguz A. Acar, “AI Prompt Engineering Isn’t the Future.” `https://hbr.org/2023/06/ai-prompt-engineering-isnt-the-future`. [Online; accessed 19-Oct-2023].

[9] S. Owen, “Building a virtual machine inside a javascript library.” `https://www.engraved.blog/building-a-virtual-machine-inside/`, 2022. Accessed: 2023-02-20.

[10] S. Wang, H. Scells, B. Koopman, and G. Zuccon, “Can chatgpt write a good boolean query for systematic review literature search?,” *arXiv preprint arXiv:2302.03495*, 2023.

[11] F. K. Akın, “Awesome ChatGPT Prompts.” `https://github.com/f/awesome-chatgpt-prompts`, accessed 2023. GitHub repository.