A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT

Author: Please provide author information

Abstract

Prompt engineering is an increasingly important skill set and an emerging discipline essential to converse effectively with large language models (LLMs), such as ChatGPT. Prompts are instructions given to an LLM to enforce rules, automate processes, and ensure specific qualities (and quantities) of generated output. They are also a form of programming that can customize the outputs and interactions with an LLM.

This paper describes a catalog of prompt engineering techniques, presented in pattern form, that have been applied to solve common problems when conversing with LLMs. Prompt patterns are a knowledge transfer method, analogous to software patterns, that provide reusable solutions to common problems faced in a particular context, i.e., output generation and interaction when working with LLMs.

This paper provides the following contributions to research on prompt engineering for using LLMs to aid software development tasks. First, it provides a framework for documenting patterns for structuring prompts to solve a range of problems so that they can be adapted to different domains. Second, it presents a catalog of patterns that have been applied successfully to improve the outputs of LLM conversations. Third, it explains how prompts can be built from multiple patterns and illustrates prompt patterns that benefit from combination with other prompt patterns.

2012 ACM Subject Classification Software and its engineering; Computing methodologies → Artificial intelligence

Keywords and phrases Large Language Models, Prompt Patterns, Prompt Engineering

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

The advent of large language models (LLMs) for software development. Conversational large language models (LLMs) [1, 2], such as ChatGPT [3], have generated immense interest in domains ranging from answering questions on medical licensing exams [4] to generating code snippets in multiple languages and programming paradigms. LLMs answer questions posed by users in natural language form known as "prompts" [5], which are sets of instructions provided to an LLM that program it by customizing it and/or enhancing or refining its capabilities. A prompt influences subsequent interactions with—and output generated from—an LLM by providing specific rules and guidelines for an LLM conversation with a set of initial rules. A prompt also sets the context for the conversation and tells the LLM what information is important and what the desired output form and content should be.

For example, a prompt could specify an LLM to only generate code that follows a certain coding style or programming paradigm. Likewise, it could specify an LLM should flag certain keywords or phrases in a generated document and provide additional information related to those keywords. By introducing such guidelines, prompts facilitate more structured and nuanced outputs to aid a large variety of software development tasks in the context of LLMs.

Despite being generally available for only a few months, chat-adapted LLMs are being used to generate and assess computer programs, giving a preview of the impact LLMs will soon have on society, research, and education. However, there is little disciplined knowledge about chat-adapted LLMs, their capabilities, and their limitations. In particular, LLMs can

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:29
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

be viewed as a new computational platform with a unique programming paradigm. They will likely serve as the foundations for future interactive development environments (IDEs) where humans and artificial intelligent (AI) tools work together as trustworthy collaborators to more rapidly and reliably evolve software-reliant systems [6]. For example, LLMs are being integrated directly into software tools, such as Github's Co-Pilot [7, 8, 9] and included in integrated development environments (IDEs), such as IntelliJ [10] and Visual Studio Code, thereby allowing software teams to access these tools directly from their preferred IDE.

Motivating the need for prompt engineering. Programming first-generation LLMs, like ChatGPT, involves natural language prompts, such as asking the LLM to explain a software vulnerability or generate JavaScript for a web page. These trivial examples of prompts, however, do not reveal the significantly more sophisticated LLM computational abilities. Harnessing the potential of LLMs in productive ways requires a systematic focus on *prompt engineering*, which is an emerging discipline that studies structured interactions with—and programming of—new LLM computational systems to solve complex problems via natural language interfaces.

This paper focuses on pattern-oriented prompt engineering techniques to enhance the application of LLMs in the software domain, including helping developers code effectively and efficiently with unfamiliar APIs or allowing students to acquire new coding skills and techniques. To demonstrate the promise of prompt engineering for software development, we provided the following prompt to ChatGPT: https://www.overleaf.com/project/63f7764dbbe3fcbbee132b92

> **Prompt:** "From now on, I would like you to ask me questions to deploy a Python application to AWS. When you have enough information to deploy the application, create a Python script to automate the deployment."

This example prompt causes ChatGPT to begin asking the user questions about their software application. ChatGPT will drive the question-asking process until it reaches a point where it has sufficient information to generate a Python script that automates deployment. This example demonstrates the programming potential of prompts beyond conventional "generate a method that does X" style prompts or "answer this quiz question."

Moreover, prompts can be engineered to program an LLM to do more than simply dictating the output type or filtering the information provided to the model. With the right prompt, for example, it is possible to create entirely new interaction paradigms, such as having an LLM generate and give a quiz associated with a software engineering concept or tool, or even simulate a Linux terminal. Moreover, prompts have the potential for self-adaptation, *e.g.*, suggesting other prompts to gather additional information or generate related artifacts. These advanced capabilities of prompts highlight the importance of engineering them to provide value beyond simple text or code generation.

Prompt patterns are an essential foundation to an effective discipline of prompt engineering. A key contribution of this paper is the introduction of *prompt patterns* to document successful approaches for systematically engineering different output and interaction goals when working with conversational LLMs. Prompt patterns are similar to software patterns [11, 12] in that they offer reusable solutions to certain problems. They focus, however, on the context of output generation from LLMs, such as (but not limited to) ChatGPT.

By documenting and leveraging prompt patterns in the context of automating software development tasks, individual users and teams can enforce constraints on the generated output, ensure that relevant information is included, and change the format of interaction with the LLM to better solve problems they face. Prompt patterns can be viewed as a corollary to the broad corpus of general software patterns, just adapted to the more specific context of LLM output generation.

This paper focuses on domain-independent prompt patterns and presents a catalog of such patterns that have been applied to solve problems ranging from production of visualizations and code artifacts to automation of output steps for code editing to helping fact check outputs. Just as catalogs of software patterns provide a codified means to solve common software development challenges, catalogs of prompt patterns provide a codified approach to customizing the output and interactions of LLMs.

Paper Organization. The remainder of this paper is organized as follows: Section 2 introduces prompt patterns and compares these patterns to well-known software patterns; Section 3 describes 12 prompt patterns that have been applied to solve common problems in the domain of conversational LLM interaction and output generation for automating software development tasks; Section 4 discusses related work; and Section 5 presents concluding remarks and lessons learned.

2 Comparing Software Patterns with Prompt Patterns

The quality of the output(s) an LLM generates is directly related to the quality of the prompts provided by the user. As discussed in Section 1, the prompts given to a conversational LLM can be used to program interactions between a user and an LLM to better solve a diverse set of problems. One contribution of this paper is the framework it provides to document patterns that structure prompts to solve a range of software tasks that can be adapted to different domains.

Our framework is useful since it codifies patterns that help users interact more effectively with conversational LLMs in a variety of contexts, rather than simply showing interesting examples or domain-specific prompts. Codifying this knowledge in pattern form enhances reuse and transferability to other contexts and domains where users face similar—but not identical—problems.

The topic of knowledge transfer has been studied extensively in the software patterns literature [11, 12] at multiple levels, *e.g.*, design, architectural, and analysis. This paper applies a variant of a familiar pattern form as the basis of our prompt engineering approach. Since prompts are a form of programming, it is natural to document them in pattern form.

2.1 Overview of Software Patterns

A software pattern provides a reusable solution to a recurring problem within a particular context [11]. Documenting software patterns concisely conveys (and generalizes) from specific problems being addressed to identify important forces and/or requirements that should be resolved and/or addressed in successful solutions.

A pattern form also includes guidance on how to implement the pattern, as well as information on the trade-offs and considerations to take into account when implementing a pattern. Moreover, example implementations of the pattern are often provided to further showcase the pattern's applicability in practice. Software patterns are typically documented in a stylized form to facilitate their use and understanding, such as:

- **A name and classification**. Each pattern has a name that identifies the pattern and should be used consistently. Patterns can be classified in various ways, including purpose (*.e.*, creational, structural, or behavioral patterns), granularity (*e.g.*, design, architectural, or enterprise patterns), etc.

- **The intent** concisely conveys the purpose the pattern is intended to achieve.
- **The motivation** documents the underlying problem and forces the pattern is meant to address and the importance of the problem.
- **The structure and participants**. The structure describes key pattern participants (such as classes and objects) and depicts how they collaborate to form a generalized solution.
- **Example code** concretely maps the pattern to some underlying programming language(s) and helps developers gain insight on how to apply the pattern effectively.
- **Consequences** summarize the pros and cons of applying the pattern in practice.

2.2 Overview of Prompt Patterns

Prompt patterns serve a similar purpose and follow a similar format to classic software patterns, with slight modifications to match the context of output generation with LLMs.[1] Each of the analogous sections for the prompt pattern form used in this paper is summarized below:

- **A name and classification**. The prompt pattern name uniquely identifies the pattern and indicates the problem that is being addressed. For the classification, we have developed a series of initial categories of pattern types summarized in Table 1.
- **The intent and context** describes the problem the prompt pattern solves and the goals it achieves. The problem should ideally be domain independent, though domain-specific patterns can be documented with an appropriate discussion of the context where the pattern applies.
- **The motivation** provides the rationale for the problem and explains why solving it is important. The motivation is explained in the context of users interacting with a conversational LLM and how it can improve upon users informally prompting the LLM in one or more circumstances. Specific circumstances where the improvements are expected are documented.
- **The structure and key ideas**. The structure describes the fundamental contextual information (presented as a series of key ideas) that the prompt pattern provides to the LLM. These ideas are similar to “participants” in a software pattern. The contextual information may be communicated through varying wording (just as a software pattern can have variations in how it is realized in code), but should convey fundamental pieces of information that form core elements of the pattern.
- **Example implementation** demonstrates how the prompt pattern is worded in practice and the types of output generated by an LLM.
- **Consequences** summarize the pros and cons of applying the pattern and may provide guidance on how to adapt the prompt to different contexts.

2.3 Evaluating Means for Defining a Prompt Pattern’s Structure and Ideas

In software patterns, the structure and participants are normally defined in terms of UML diagrams, such as structure diagrams and/or interaction diagrams. These UML diagrams

---

[1] The most direct translation of software pattern structure to prompt patterns is the naming, intent, motivation, and sample code. The structure and classification, however, require more adaptation although they named similarly.

explain what the participants of the pattern are and how they interact to solve the problem. In prompt patterns, something analogous is needed, though UML is not an ideal documentation approach since it is intended to describe software structures, as opposed to the ideas communicated in a prompt.

Several possible approaches could be used, ranging from diagrams to defining grammars for a prompt language. Although grammars initially seemed attractive due to their formal nature, we found they incurred the following limitations:

- The goal of prompts is to communicate knowledge in a clear and concise way to conversation LLM users, who may or may not be computer scientists or programmers. As a community, we should strive to create an approachable format that communicates knowledge clearly to a diverse target audience.
- It is possible to phrase a prompt in many different ways, most commonly by typing phrases into a terminal in a free form human language. It is hard, however, to define a grammar that accurately and completely expresses all the nuanced ways that components of a prompt could be expressed in text or symbols.
- Prompts fundamentally convey ideas to a conversational LLM and are not simply the production of tokens for input. In particular, an idea built into a prompt pattern can be communicated in many ways and its expression should be at a higher level than the underlying tokens representing the idea.
- It is possible to program an LLM to introduce novel semantics for statements and words that create new ways for communicating an idea. In contrast, grammars may not easily represent ideas that can be expressed through completely new symbology or languages that the grammar designer was not aware of.

2.4 A Way Forward: Fundamental Contextual Statements

Given the limitation with formal grammars, and open research question is how to more effectively describe prompt pattern structure and ideas. We propose the concept of *fundamental contextual statements*, which are written descriptions of the important ideas to communicate in a prompt to an LLM. An idea can be rewritten and expressed in arbitrary ways based on user needs and experience. The key ideas to communicate, however, are presented to the user as a series of simple, but fundamental, statements, as shown throughout the examples in Section 3.

One benefit of adopting and applying fundamental contextual statements is that they are intentionally intuitive to users. In particular, we expect users will understand how to express and adapt the statements in a contextually appropriate way for their domain. Moreover, since the underlying ideas of the prompt are captured, these same ideas can be expressed by the user in alternate symbology or wording that has been introduced to the LLM using patterns, such as the *Meta Language Creation* pattern presented in Section 3.2.

Our ultimate goal is to enhance prompt engineering by providing a framework for designing prompts that can be reused and/or adapted to other LLMs, much like software patterns can be implemented in different programming languages and platforms. For the purposes of this paper, however, all prompts were tested with ChatGPT [13]. We use this LLM for all examples presented in this paper due to its widespread availability and popularity. These examples were documented through a combination of exploring the corpus of community-posted prompts on the Internet and independent prompt creation from using ChatGPT to automating software development tasks.

3 A Catalog of Prompt Patterns for Conversational LLMs

This section presents our catalog of prompt patterns that have been applied to solve common problems in the domain of conversational LLM interaction and output generation for automating software tasks. Each prompt pattern is accompanied by concrete implementation samples and examples with and without the prompt.

3.1 Summary of Our Prompt Pattern Catalog

Organizing a large catalog of prompt patterns into easily understandable categories helps create documentation that users can apply more effectively. Table 1 outlines the initial classifications for the catalog of prompt patterns we identified in our work with ChatGPT thus far. As shown in this table, there are five categories of prompt patterns in our classification framework: **Input Semantics**, **Output Customization**, **Error Identification**, **Prompt Improvement**, and **Interaction**, each of which is summarized below.

Table 1 Classifying Prompt Patterns

Pattern Category	Prompt Pattern
Input Semantics	*Meta Language Creation*
Output Customization	*Output Automater*
	Persona
	Visualization Generator
	Recipe
	Template
Error Identification	*Fact Check List*
Prompt Improvement	*Alternative Approaches*
	Cognitive Verifier
	Refusal Breaker
Interaction	*Flipped Interaction*
Context Control	*Context Manager*

The **Input Semantics** category deals with how an LLM understands user input and how it translates this input into something it can use to generate output. This category includes the *Meta Language Creation* pattern, which focuses on creating a custom language for the LLM to understand. This pattern is useful when the default input language is ill-suited for expressing ideas users want to convey to the LLM.

The **Output Customization** category focuses on constraining or tailoring the types, formats, structure, or other properties of the output generated by the LLM. Prompt patterns in this category include *Output Automater*, *Persona*, *Visualization Generator*, *Recipe*, and *Template* patterns. The *Output Automater* pattern creates scripts that automate tasks the LLM output suggests users should perform. The *Persona* pattern gives the LLM a persona or role to play when generating output. The *Visualization Generator* pattern allows users to generate visualizations by producing textual outputs that can be fed to other tools, such as other AI-based image generators, like DALL-E [14]. The *Recipe* pattern allows users to obtain a sequence of steps or actions to realize a stated outcome, possibly with partially known information or constraints. The *Template* pattern allows users to specify a template for the output, which the LLM fills in with content.

The **Error Identification** category focuses on identifying and resolving errors in the output generated by the LLM. This category includes the *Fact Check List* pattern. The *Fact Check List* pattern requires the LLM to generate a list of facts the output depends on that should be fact-checked, then introspect on its output and identify any errors.

The **Prompt Improvement** category focuses on improving the quality of the input and output. This category includes the *Alternative Approaches*, *Cognitive Verifier*, and *Refusal Breaker* patterns. The *Alternative Approaches* pattern requires the LLM to suggest alternative ways of accomplishing a user-specified task. The *Cognitive Verifier* pattern instructs the LLM to automatically suggest a series of subquestions for users to answer before combining the answers to the subquestions and producing an answer to the overall question. The *Refusal Breaker* pattern requires the LLM to automatically reword user questions when it refuses to produce an answer.

The **Interaction category** focuses on the interaction between users and the LLM. This category includes the *Flipped Interaction* patterns. The *Flipped Interaction* pattern requires the LLM to ask questions of users rather than generate output.

Finally, the **Context Control** category focuses on controlling the contextual information in which the LLM operates. This category includes the *Context Manager* pattern, which allows users to specify the context for the LLM's output.

We encourage readers to use ChatGPT to test the provided patterns. The output from ChatGPT has been omitted for brevity in most cases. However, the patterns are all easily testable in the current verison of ChatGPT. The remainder of this section describes each prompt pattern using the pattern form discussed in Section 2.2.

3.2 The Meta Language Creation Pattern

Intent and Context

During a conversation with an LLM, users would like to write the prompt using an alternate notation or language, such as a textual short-hand notation for graphs, a description of states and state transitions for a state machine, a set of commands for prompt automation, etc. For example, a user would like to use the graph notation "a→b" to express a directed graph. The intent of this pattern is to explain the semantics of this alternative language to the LLM so users can write future prompts in this new language and its semantics.

Motivation

Many problems, structures, or other ideas communicated in a prompt may be more concisely, unambiguously, or clearly expressed in a language other than English (or whatever conventional human language is used to interact with an LLM). To produce output based on an alternative language, however, an LLM needs to understand the language's syntax and semantics in terms of a language that it was trained on.

Structure and Key Ideas

Fundamental contextual statements:

Contextual Statements
When I say X, I mean Y (or would like you to do Y)

The key structure of this pattern involves explaining the meaning of one or more symbols, words, or instruction to the LLM so it uses the provided semantics for the ensuing conversation.

This statement can take the form of a simple translation, such as “X” means “Y.” It can also take more complex forms that define a series of commands and their semantics, such as “when I say X, I want you to do Y”. In this case, “X” is henceforth bound to the semantics of “take action Y.” For example, "whenever I type the word ’outline’, print an outline of all the topics discussed thus far."

Example Implementation

The key to using the *Meta Language Creation* pattern successfully is developing an unambiguous notation or shorthand, such as the following:

> “From now on, whenever I type two identifiers separated by a “→", I am describing a graph. For example, “a → b” is describing a graph with nodes “a” and “b” and an edge between them. If I separate identifiers by “-[w:2, z:3]→”, I am adding properties of the edge, such as a weight or label.”

This example of the *Meta Language Creation* pattern establishes a standardized notation for describing graphs by defining a convention for representing nodes and edges. Whenever the author types two identifiers separated by a “→” symbol, it indicates a graph is being described. For example, if the author types “a → b”, this indicates that a graph is being defined with nodes “a” and “b”, and that there is an edge between them. This convention provides a clear and concise way to communicate the structure of a graph in written form.

Moreover, the prompt goes on to specify that additional information about the edges, such as a weight or label, can be provided using the syntax “-[w:2, z:3]→.” This notation allows the specification of additional properties beyond the basic structure of the graph. These specified properties are associated with the edge between the two nodes and can provide important context for the interpretation of the graph. This standardized notation for describing graphs makes it easier to communicate graph structures and properties, which may not be easy or are overly verbose to describe as a series of sentences.

Consequences

Although this pattern provides a powerful means to customize a user’s interaction with an LLM, it may create the potential for confusion within the LLM itself. For example, the prompt “whenever I separate two things by commas, it means that the first thing precedes the second thing” will likely create significant potential for ambiguity and create the potential for unexpected semantics because commas are used as a standard form of punctuation that does not have the semantic "a precedes b". As important as it is to clearly define the semantics of the language, it is also essential to ensure the language itself introduces no ambiguities that degrade the LLM’s performance or accuracy.

To showcase the potential for confusion, ChatGPT will warn the user and potentially be unable to perform a specific mapping of a symbol or term to a new meaning. For example, if following prompt is given to ChatGPT

> “Whenever I say “a”, I am referring to Marie Antoinette.”

it will respond that this prompt creates too much confusion as “a” is an indefinite article and too commonly used, as shown below:

> “Referring to Marie Antoinette using “a” could potentially cause confusion, as “a” is an indefinite article commonly used to refer to a noun that has not been mentioned

> before. It may be more clear to use her name, “Marie Antoinette”, or a pronoun such as “she” or “her” to refer to her. Moreover, using context to establish who you are referring to can also help avoid confusion in conversation."

An LLM generally performs better on data it was trained on, so when an existing notation is available—but may not be the dominant meaning—the *Meta Language Creation* pattern can provide context to scope the meaning of the symbol to improve the accurately and utility of the output. For example, the symbol “→” conveys slightly different information in graphs and propositional logic. The statement “p → q, q → p” may therefore make sense, but only if the LLM is given the appropriate context via the *Meta Language Creation* prompt pattern.

Although this pattern is powerful it must be used carefully, particularly when describing concepts that may otherwise be hard to precisely or concisely describe. These types of prompts are thus best used in completely new conversation sessions. Using a single meta-language-per-conversation session is also a best practice to avoid the potential for conflicting or unexpected semantics being applied to the conversation over time.

3.3 The Output Automater Pattern

Intent and Context

The intent of this pattern is to have the LLM generate a script or other automation artifact that can automatically perform any steps it recommends taking as part of its output. The goal is to reduce the manual effort needed to implement any LLM output recommendations. This pattern can also be used to allow the LLM to execute actions in other systems.

Motivation

The output of an LLM is often a sequence of steps for the user to follow. For example, when asking an LLM to generate a Python configuration script it may suggest a number of files to modify and changes to apply to each file. Another output may be a sequence of configuration actions in a cloud computing console, such as the Amazon Web Services console. It is tedious and error-prone, however, for users to repeatedly perform the manual steps dictated by LLM output.

Structure and Key Ideas

Fundamental contextual statements:

Contextual Statements
Whenever you produce an output that has at least one step to take and the following properties
Produce an executable artifact of type X that will automate these steps

The first statement identifies the situations under which automation should be generated. A simple approach is to state the output includes at least two steps to perform and that an automation artifact should then be produced. This "scope limiting" helps prevent producing output automation scripts in cases where running such scripts requires more user effort than simply performing the original steps produced in the output. In particular, the scope can be limited to outputs requiring more than a certain number of steps.

The second statement provides a concrete statement of the type of output the LLM should output to perform the automation. For example, “produce a Python script” gives the LLM a concrete understanding to translate the general steps into equivalent steps in Python. The automation artifact should be concrete and something the LLM associates with the action of “automating a sequence of steps.”

Example Implementation

A sample of the *Output Automater* pattern applied to code snippets generated by the ChatGPT LLM is shown below:

> “From now on, whenever you generate code that spans more than one file, generate a Python script that can be run to automatically create the specified files or make changes to existing files to insert the generated code.”

The sample prompt automates the common software engineering task of taking the LLM output and editing one or more files. The scope is "whenever you generate code that spans more than one file," which is when the developer would have multiple, potentially error-prone steps, to make the necessary code edits. The "generate a Python script that can be run to automatically" indicates that the goal is to automate the editing of the files so that the developer doesn’t have to. The specification of the goal is important so that the LLM doesn’t automate unrelated tasks, since the output may have other steps that need to be completed unrelated to coding.

The *Output Automater* pattern is a powerful complement for any computer-controlled system, such as generating Development Operations (DevOps) artifacts for build, configuration and deployment; code editing tasks, including creation / editing of multiple files; and environment setup tasks, ranging from creation of environment variables to running necessary shell commands. An LLM can provide a set of steps to perform on the computer-controlled system and the output can then be translated into a script that allows the computer controlling the system to perform the steps automatically. This pattern enables LLMs to integrate quality into—and exert control over—new computing systems that have a known scripting interface.

Consequences

This pattern is particularly effective in software development since programmers often use LLMs to generate output that they then copy/paste into multiple files. Some tools, such as Copilot, insert limited snippets directly into the section of code that a programmer is working with. In contrast, conversational LLMs, such as ChatGPT, do not provide such facilities. This prompt pattern is also effective at creating scripts for running commands on a terminal, automating cloud operations, or reorganizing files on a file system.

An important usage consideration of this pattern is that the automation artifact must be defined concretely. Without a concrete meaning for how to “automate” the steps, the LLM often states that it “can’t automate things” since that is beyond its capabilities. LLMs typically accept requests to produce code, however, so the goal is to instruct the LLM to generate text/code, which can be executed to automate something. This subtle distinction in meaning is important to help an LLM disambiguate the meaning of a prompt.

One caveat of the *Output Automater* pattern is the LLM needs sufficient conversational context to generate an automation artifact that is functional in the target context, such as the file system of a project on a Mac vs. Windows computer. This pattern is more effective when the full context needed for the automation is contained within the conversation, *e.g.*, when a software application is generated from scratch using the conversation and all actions on the local file system are performed using a sequence of generated automation artifacts rather than manual actions unknown to the LLM. Alternatively, self-contained sequences of steps work well, such as "how do I find the list of open ports on my Mac computer."

In some cases, the LLM may produce a long output with multiple steps and not include an automation artifact. This omission may arise for various reasons, including exceeding the output length limitation the LLM supports. A simple workaround for this situation is to remind the LLM via a follow-on prompt, such as "But you didn't automate it", which provides the context that the automation artifact was omitted and should be generated.

At this point in the evolution of LLMs, the *Output Automater* pattern is best employed by users who can read and understand the generated automation artifact. LLMs can (and do) produce inaccuracies in their output, so blindly accepting and executing an automation artifact carries significant risk. Although this pattern alleviates users from performing certain manual steps, it does not alleviate their responsibility to understand the actions they undertake using the output. When users execute automation scripts they therefore assume responsibility for the outcomes.

3.4 The Flipped Interaction Pattern

Intent and Context

You want the LLM to ask questions to obtain the information it needs to perform some tasks. Rather than users driving the conversation, therefore, the LLM should drive the conversation to focus it on achieving a specific goal. For example, you may want the LLM to give a quick quiz or automatically ask questions until it has sufficient information to generate a script to deploy your application on a particular cloud platform.

Motivation

Rather than having the user drive the conversation, an LLM often has knowledge it can use to obtain information from users more efficiently. The goal of the *Flipped Interaction* pattern is to invert the interaction flow so the LLM asks the user questions to achieve some desired goal. The LLM can often better select the format, number, and content of the interactions to ensure the goal is reached faster, more accurately, and/or by using knowledge a user may not possess initially.

Structure and Key Ideas

Fundamental contextual statements:

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.

A prompt for a flipped interaction should always specify the goal of the interaction. The first statement (*i.e.*, get the LLM to ask questions to achieve a goal) communicates this goal to the LLM. Equally important is that the questions should focus on a particular topic or outcome. By providing the goal, the LLM can understand what it is trying to accomplish through the interaction and tailor its questions accordingly. This "inversion of control" enables more focused and efficient interaction since the LLM only asks questions it deems relevant to achieving the specified goal.

The second statement provides the context for how long the interaction should occur. A flipped interaction can be terminated with a response like "stop asking questions". It is often better, however, to scope the interaction to a reasonable length or only as far as is needed to reach the goal. This goal can be surprisingly open-ended and the LLM will continue to work towards the goal by asking questions, as is the case in the example of "until you have enough information to generate a Python script."

The third (optional) statement can improve usability by limiting (or expanding) the number of questions that the LLM generates per cycle. By default, an LLM may generate multiple questions per iteration. If a precise number/format for the questioning is not specified, the questioning will be semi-random and may lead to one-at-a-time questions or ten-at-a-time questions. The prompt can thus be tailored to include the number of questions asked at a time, the order of the questions, and/or any other formatting/ordering considerations to facilitate user interaction.

Example Implementation

A sample prompt for a flipped interaction is shown below:

> "From now on, I would like you to ask me questions to deploy a Python application to AWS. When you have enough information to deploy the application, create a Python script to automate the deployment."

In general, the more specific the prompt regarding the constraints and information to collect, the better the outcome. For instance, the example prompt above could provide a menu of possible AWS services (such as Lambda, EC2, etc.) with which to deploy the application. In other cases, the LLM may be permitted to simply make appropriate choices on its own for things the user makes no explicit decisions about. One limitation of this prompt is that it may require experimentation with the precise phrasing to get the LLM to ask the questions in the appropriate number and flow to best suit the task. For example, the LLM may need tuning to determine how many questions to ask at a time and how to best tailor the sequencing for the task.

Consequences

One consideration when designing a prompt based in the *Flipped Interaction* pattern is how much to dictate to the LLM regarding what information to collect prior to termination. In the example above, the flipped interaction is open-ended and can vary significantly in the final generated artifact. This open-endedness makes the prompt generic and reusable, but may potentially ask additional questions that could be skipped if additional context is given.

If specific requirements are known in advance, it is better to inject them into the prompt rather than hoping the LLM will obtain the needed information. Otherwise, the LLM will non-deterministically decide whether to prompt the user for the information or make an educated guess as to an appropriate value.

For example, the user can state they would like to deploy an application to Amazon AWS EC2, rather than simply state "the cloud," which requires fewer interactions to narrow down the deployment target. The more precise the initial information, therefore, the better the

LLM can use the limited questions that a user may be willing to answer to obtain information the LLM requires to improve its output.

When developing prompts for flipped interactions, it is important to consider the level of user knowledge, engagement, and control. If the goal is to accomplish the goal with as little user interaction as possible (*minimal control*), that should be stated explicitly. Conversely, if the goal is to ensure the user is aware of all key decisions and confirms them (*maximum engagement*) that should also be stated explicitly. Likewise, if the user is expected to have minimal knowledge and should have the questions targeted at their level of expertise, such information should be engineered into the prompt.

3.5 The Persona Pattern

Intent and Context

In many cases, users would like LLM output to always take a certain point of view or perspective. For example, it may be useful to conduct a code review as if the LLM was a security expert. The intent of this pattern is to give the LLM a "persona" that helps it select what types of output to generate and what details to focus on.

Motivation

Users may not know what types of outputs or details are important for an LLM to focus on to achieve a given task. They may know, however, the role or type of person they would normally ask to get help with these things. The *Persona* pattern enables the users to express what they need help with, without knowing the exact details of the outputs they need.

Structure and Key Ideas

Fundamental contextual statements:

Contextual Statements
Act as persona X
Provide outputs that persona X would create

The first statement instructs the LLM needs to act as a specific persona and provide outputs like such a persona would. This persona can be expressed in a number of ways, ranging from a job description, title, fictional character, historical figure, etc. The persona should elicit a set of attributes associated with a well-known job title, type of person, etc.[2]

The secondary statement—provide outputs that persona X would create—offers opportunities for customization. For example, a computer science teacher might provide a large variety of different output types, ranging from programming assignments to reading lists to lectures. If a more specific scope to the type of output is known, the user can provide it in this statement.

Example Implementation

A sample implementation for code review is shown below:

---

[2] Be aware, however, that an LLM may discard personas relating to living people or people considered harmful due to underlying policy filters, such as privacy and security rules.

> "From now on, act as a security reviewer. Pay close attention to the security details of any code that we look at. Provide outputs that a security reviewer would regarding the code."

In this example, the LLM is instructed to provide outputs that a "security reviewer" would. The prompt further sets the stage that code is going to be evaluated. Finally, the user refines the persona by scoping the persona further to outputs regarding the code.

Personas can also represent inanimate or non-human entities, such as a Linux terminal, a database, or an animal's perspective. When using this pattern to represent these entities, it can be useful to also specify how you want the inputs delivered to the entity, such as "assume my input is what the owner is saying to the dog and your output is the sounds the dog is making,"

An example prompt for a non-human entity that uses a "pretend to be" wording is shown below:

> "You are going to pretend to be a Linux terminal for a computer that has been compromised by an attacker. When I type in a command, you are going to output the corresponding text that the Linux terminal would produce."

This prompt is designed to simulate a computer that has been compromised by an attacker and is being controlled through a Linux terminal. The prompt specifies that the user will input commands into the terminal, and in response, the simulated terminal will output the corresponding text that would be produced by a real Linux terminal. This prompt is more prescriptive in the persona and asks the LLM to not only be a Linux terminal, but to act as a computer compromised by an attacker.

The persona causes ChatGPT to generate outputs to commands that have files and contents indicative of a computer that was hacked. This example shows how an LLM can bring its situational awareness to a persona, in this case, creating evidence of a cyberattack in the outputs it generates. Such a persona can be highly effective by asking LLM to play a game, where you want to hide exact details of the output characteristics from the user (*e.g.*, do not give away what the cyberattack did by describing it explicitly in the prompt).

Consequences

An interesting aspect of taking non-human personas is that the LLM may make interesting assumptions or "hallucinations" regarding the context. A widely circulated example on the Internet asks ChatGPT to act as a Linux terminal and generate the output expected if the user typed the same text into a terminal. Commands, such as `ls -l`, will generate a file listing for an imaginary UNIX file system, complete with files that can have `cat file1.txt` run on them.

In other examples, the LLM may prompt the user for more context, such as when ChatGPT is asked to act as a MySQL database and prompts for the structure of a table that the user is pretending to query. ChatGPT can then generate synthetic rows, such as generating imaginary rows for a "people" table with columns for "name" and "job,"

3.6 The Alternative Approaches Pattern

Intent and Context

The intent of the pattern is to ensure an LLM always offers alternative ways of accomplishing a task so users do not pursue only familiar approaches. The LLM can then provide alternative approaches that always force users to think about what they are doing and determine if that is the best approach to meet reach their goal. In addition, solving the task may inform the user or teach them about alternative concepts for subsequent follow-up prompts.

Motivation

Humans often suffer from cognitive biases that lead them to choose a particular approach to solve a problem, even when it is not the right or "best" approach. Moreover, humans may be unaware of alternative approaches to what they have used in the past. The motivation of the *Alternative Approaches* pattern is to ensure users are aware of alternative approaches to select a better approach to solve a problem by dissolving their cognitive biases.

Structure and Key Ideas

Fundamental contextual statements:

Contextual Statements
Within scope X, if there are alternative ways to accomplish the same thing, list the best alternate approaches
(Optional) compare/contrast the pros and cons of each approach
(Optional) include the original way that I asked
(Optional) prompt me for which approach I would like to use

The initial portion of the first statement ("within scope X") scopes the interaction to a particular goal, topic, or bounds on the questioning. The scope is the constraint(s) users place on alternative approaches. The scope could be "for implementation decisions" or "for the deployment of the application". The scope ensures that any alternatives fit within the boundaries or constraints to which the user must adhere.

The next portion of the first statement ("if there are alternative ways to accomplish the same thing, list the best alternate approaches") instructs the LLM to suggest alternatives. As with other prompt patterns, instruction specificity can be increased or include domain-specific contextual information. For example, the statement could be scoped to "if there are alternative ways to accomplish the same thing with the software framework that I am using" to prevent the LLM from suggesting alternatives that are inherently non-viable because they require too many changes to other parts of the application.

Since users may not be aware of alternative approaches, they also may not be aware of why and when to choose one of the alternatives. The second (optional) statement ("compare/contrast the pros and cons of each approach") adds decision making criteria to the analysis. This statement ensures the LLM provides users with the necessary rationale for alternative approaches.

The final (optional) statement ("prompt me for which approach I would like to use") helps eliminate users needing to manually copy/paste or enter in an alternative approach if one is selected.

Example Implementation

The following is an example prompt that generates, compares, and allows a user to select one or more alternative approaches:

> “Whenever I ask you to deploy an application to a specific cloud service, if there are alternative services to accomplish the same thing with the same cloud service provider, list the best alternative services and then compare/contrast the pros and cons of each approach with respect to cost, availability, and maintenance effort and include the original way that I asked. Then ask me which approach I would like to proceed with.”

This implementation of the *Alternative Approaches* pattern is specifically tailored for the context of software engineering and focuses on the deployment of applications to cloud services. The prompt is intended to intercept places where developers have made a cloud service selection without full awareness of alternative services that may be priced more competitively or easier to maintain. The prompt directs ChatGPT to list the best alternative services that can accomplish the same task with the same cloud service provider (providing constraints on the alternatives), and to compare and contrast the pros and cons of each approach.

Consequences

This pattern is effective in its generic form and can be applied to a range of tasks. Refinements could include having a standardized catalog of acceptable alternatives in a specific domain from which the user must select. The *Alternative Approaches* pattern can also be used to incentivize users to select one of an approved set of approaches while informing them of the pros and cons of the approved options.

3.7 The Cognitive Verifier Pattern

Intent and Context

Research literature has documented that LLMs can often reason better if a question is subdivided into additional questions, that are each individually answered. The LLM can then combine the answers to the sub-questions into a better overall answer to the original question [15]. The intent of the *Cognitive Verifier* pattern is to force the LLM to always subdivide questions into additional questions that can be used to provide a better answer to the original question.

Motivation

The motivation of the *Cognitive Verifier* pattern is two-fold:

- Humans may initially ask questions that are too high-level to provide a concrete answer to without additional follow-up due to unfamiliarity with the domain, laziness in prompt entry, or being unsure about what the correct phrasing of the question should be.
- Research has demonstrated that LLMs can often perform better when using a question that is subdivided into additional questions.

Structure and Key Ideas

Fundamental contextual statements:

Contextual Statements
When you are asked a question, follow these rules
Generate a number of additional questions that would help more accurately answer the question
Combine the answers to the individual questions to produce the final answer to the overall question

The first statement instructs the LLM to generate a number of additional questions that help answer the original question more accurately . This step instructs the LLM to (1) consider the context of the question, (2) identify any information that may be missing or unclear, and (3) combine the answers to the additional questions to provide context to help answer the overall question. By generating additional questions, the LLM can help ensure its ultimate answer is as complete and accurate as possible. This step also encourages critical thinking by users and can help uncover new insights or approaches that may not have been considered initially, thereby yielding better follow-on questions.

The second statement instructs the LLM to combine answers to individual questions to produce the ultimate answer to the overall question. This step ensures all the information gathered from the individual questions is incorporated into the final answer. By combining answers, the LLM can provide a more comprehensive and accurate response to the original question. This step also helps ensure all relevant information is taken into account and the final answer is not based on any single answer.

Example Implementation

> “When I ask you a question, generate three additional questions that would help you give a more accurate answer. When I have answered the three questions, combine the answers to produce the final answers to my original question.”

This specific instance of the *Cognitive Verifier* pattern refines the original pattern by specifying a set number of additional questions that the LLM should generate in response to a question. In this case, the prompt instructs the LLM to generate three additional questions that help it answer the original question more accurately. The specific number can be based on a user’s experience and willingness to provide follow-up information.

The following refinement to the prompt above provides more context for the amount of domain knowledge the LLM can assume the user has to guide the creation of additional questions:

> “When I ask you a question, generate three additional questions that would help you give a more accurate answer. Assume that I know little about the topic that we are discussing and please define any terms that are not general knowledge. When I have answered the three questions, combine the answers to produce the final answers to my original question.”

The refinement also specifies that the user may not have a strong understanding of the topic being discussed, so the LLM should define any terms that are not general knowledge. The goal is to ensure follow-up questions are not only relevant and focused, but also accessible to the user, who may be unfamiliar with technical or domain-specific terms. By providing clear and concise definitions, the LLM can help ensure its follow-up questions are easy to understand and the final answer is accessible to users with varying levels of knowledge and expertise.

Consequences

This pattern can dictate the exact number of questions to generate or leave this decision to the LLM. There are pros and cons to dictating the exact number. A pro is that specifying an exact number of questions can tightly scope the amount of additional information the user must provide so it is within a range they are willing and able to contribute.

A con, however, is that given N questions there may be an invaluable $N + 1$ question that is always scoped out. Alternatively, the LLM can be provided a range or allowed to ask additional questions. Of course, by omitting a limit on the number of questions the LLM may generate numerous additional questions that overwhelm users.

3.8 The Fact Check List Pattern

Intent and Context

The intent of this pattern is to ensure the LLM outputs a list of facts that are present in the output and form an important part of the statements in the output. This list of facts helps inform users of the facts (or assumptions) the output is based on. Users can then perform appropriate due diligence on these facts/assumptions to validate the output's veracity.

Motivation

A current weakness of LLMs (including ChatGPT) is they often rapidly (and enthusiastically!) generate convincing text that is factually incorrect. These errors can take a wide range of forms, including fake statistics to invalid version numbers for software library dependencies. Due to the convincing nature of this generated text, however, users may not perform appropriate due diligence to determine output accuracy.

Structure and Key Ideas

Fundamental contextual statements:

Contextual Statements
Generate a set of facts that are contained in the output
The set of facts should be inserted in a specific point in the output
The set of facts should be the fundamental facts that could undermine the veracity of the output if any of them are incorrect

The first statement instructs the LLM to identify the facts that are contained within its output. The LLM should be able to identify facts very effectively as they are a very well-understood concept and not impacted by the actual content, they are a domain-independent concept. The second statement tells where the facts should be included in the output, such as at the end or beginning of the output – although other arrangements could be employed.

The third statement expresses the idea that facts should be the ones most important to the overall truthfulness of the statements, *i.e.*, choose facts fundamental to the argument and not derived facts flowing from those facts. This statement is crucial since it helps to scope the output to those facts most important to the veracity and not derived statements that may not be as important – although this constraint could be relaxed.

One point of variation in this pattern is where the facts are output. Given that the facts may be terms that the user is not familiar with, it is preferable if the list of facts comes *after* the output. This after-output presentation ordering allows users to read and (attempt to) understand the statements before seeing what statements should be checked. Users may also determine additional facts prior to realizing the fact list at the end should be checked.

Example Implementation

A sample wording of the *Fact Check List* pattern is shown below:

> “From now on, when you generate an answer, create a set of facts that the answer depends on that should be fact-checked and list this set of facts at the end of your output. Only include facts related to cybersecurity.”

The user may have expertise in some topics related to the question but not others. A fact check list can be tailored to topics that users are not as experienced in or where there is the most risk. For example, in the prompt above, the user is scoping the fact check list to security topics since these are likely important from a risk perspective and may be poorly understood by developers. Targeting the facts also reduces user cognitive burden by potentially listing fewer items for investigation.

Consequences

The *Fact Check List* pattern should be employed whenever users are not experts in the domain for which they are generating output. For example, software developers reviewing code could benefit from security consideration suggestions. In contrast, an expert on software architecture may identify errors in statements about the software structure and need not see a fact check list for these outputs.

Errors are potential in all LLM outputs, so *Fact Check List* is an effective pattern to combine with other patterns, such as the *Cognitive Verifier* pattern. A key aspect of the *Fact Check List* pattern is that users can inherently check the list of facts against the output. In particular, users can directly compare a fact check list to the output to verify the facts in the fact check list actually appear in the output. Users can also identify any omissions from the list. Although a fact check list may also have errors, users often have sufficient knowledge and context to determine its completeness and accuracy relative to the LLM’s output.

One caveat of the *Fact Check List* pattern is that it only applies when the output type is amenable to fact-checking. For example, the pattern works when asking ChatGPT to generate a Python “requirements.txt” file since it lists the versions of libraries as facts that should be checked, which is handy as versions commonly have errors. However, ChatGPT will refuse to generate a fact check list for a code sample and indicate that this is something it cannot check, even though the code may have errors.

3.9 The Template Pattern

Intent and Context

The intent of the pattern is to ensure an LLM’s output follows a precise template in terms of structure. For example, the user might need to generate a URL that inserts generated information into specific positions within the URL path. This pattern allows the user to instruct the LLM to produce its output in a format it would not ordinarily use for the specified type of content being generated.

Motivation

In some cases, output must be produced in a precise format that is application or use-case specific and not known to the LLM. Since an LLM is unaware of the template structure, it must be instructed what the format is and where the different parts of its output should go. These instructions could take the form of a sample data structure to generate, a series of form letters being filled in, etc.

Structure and Key Ideas

Fundamental contextual statements:

Contextual Statements
I am going to provide a template for your output
X is my placeholder for content
Try to fit the output into one or more of the placeholders that I list
Please preserve the formatting and overall template that I provide
This is the template: PATTERN with PLACEHOLDERS

The first statement directs the LLM to follow a specific template for its output. This template will be used to try and coerce the LLMs responses into a structure consistent with user formatting needs, which is useful when the target format is not known to the LLM. If the LLM already has knowledge of the format, such as a specific file type, the *Template* pattern can be skipped and the user can simply specify the known format. However,there may be cases, such as generating Javascript Object Notation (JSON), where large variation exists in how data could be represented within a format. In such cases a template helps ensure the representation within the target format meets additional user constraints.

The second statement makes the LLM aware that the template contains a set of placeholders. These placeholders enable users to explain how the output should be inserted into the template. They also allow the user to target where information should be inserted semantically. Placeholders can use formats (*e.g.*, NAME) that allow an LLM to infer semantic meaning and determine where output should be inserted (*e.g.*, insert the person's name in the NAME placeholder). By using placeholders, moreover, users can indicate what is *not* needed in the output, *e.g.*, if a placeholder does not exist for a component of the generated output that component can be omitted. Ideally, placeholders should use a format (*e.g.*, all caps or enclosure in brackets) commonly employed in text where the LLM was trained on.

The third and fourth statements attempt to constrain the LLM so it does not arbitrarily rewrite the template or attempt to modify it so all the output components can be inserted. However, these statements may not preclude an LLM from generating additional text generated before or after. In practice, LLMs will typically follow the template, but it may be hard to eliminate additional text being generated beyond the template without further experimentation with prompt wording.

Finally, the fifth statement provides the actual template to output the text in.

Example Implementation

A sample template for generating URLs where the output is put into specific places in the template is shown below:

> "I am going to provide a template for your output. Everything in all caps is a placeholder. Any time that you generate text, try to fit it into one of the placeholders that I list. Please preserve the formatting and overall template that I provide at https://myapi.com/NAME/profile/JOB"

A sample interaction after the prompt above was given is shown next:

User: "Generate a name and job title for a person"
ChatGPT: "https://myapi.com/Emily_Parker/profile/ Software_Engineer"

Consequences

One consequence of applying the *Template* pattern is that it filters an LLM's output, which may eliminate other outputs the LLM would have provided that might be useful to users. In many cases, the LLM can provide helpful descriptions of code, decision making, or other details that this pattern may eliminate from the output. Users should therefore weight the pros/cons of filtering out this additional information prematurely.

In addition, filtering may make it hard to combine this pattern with other patterns from the **Output Customization** category. The *Template* pattern effectively constrains the output format, so it may not be compatible with generation of certain other types of output. For example, in the template provided above for a URL, it would not be easy (or even possible) to combine with the *Recipe* pattern, which needs to output a list of steps.

3.10 The Refusal Breaker Pattern

Intent and Context

This pattern asks an LLM to help users rephrase a question automatically when it refuses to give an answer. The goal is to help users be more productive and get answers to questions that are within the capabilities of an LLM to answer, but only if worded correctly. The pattern helps users identify the errors in their question or task formulation that are creating a stumbling block and learn from these mistakes to improve future prompts.

Motivation

LLMs may sometimes refuse to answer a question, either because they do not have the required knowledge or because the question is phrased in a way that they do not understand. This outcome may be frustrating for users who are looking for answers. In some situations, therefore, the *Refusal Breaker* pattern can help users find a way to either rephrase their question or ask a different question the LLM is better equipped to answer.

For example, you could ask for clarification on why the question was refused, as there may be a specific reason why LLM could not provide an answer. Another approach is to ask a different question that may be within an LLM's knowledge cut-off or expertise. All these approaches, however, require user intervention and knowledge of how to (attempt to) bypass the refusal.

Structure and Key Ideas

Fundamental contextual statements:

Contextual Statements
Whenever you can't answer a question
Explain why you can't answer the question
Provide one or more alternative wordings of the question that you could answer

The first statement indicates that this set of instructions should only be applied when a question cannot be answered, *i.e.*, users do not want to trigger these instructions under normal circumstances. The second statement instructs an LLM to explain why a question cannot be answered, which may include assumptions, constraints, or misunderstandings that can be leveraged by users to strategize about how to reword the question.

Although just the explanation of why the question cannot be answered could help users overcome a refusal, the *Refusal Breaker* pattern adds a third idea that instructs the LLM to suggest acceptable rewordings. This pattern combines both reasoning and suggested rewordings to improve the chances that users will find a path around whatever roadblocks are preventing the question from being answered.

Example Implementation

The *Refusal Breaker* pattern can be implemented and applied with the following phrasing:

> "Whenever you can't answer a question, explain why and provide one or more alternate wordings of the question that you can't answer so that I can improve my questions."

A sample discourse after using the pattern is shown below:

> **User:** What is the meaning of life?
> **ChatGPT**: As an AI language model, I do not have personal beliefs or opinions. The meaning of life is a complex philosophical question that has been pondered by humans for centuries and can vary greatly from person to person. It may be more productive to rephrase the question in a way that can be answered by information and knowledge, such as "What are some philosophical perspectives on the meaning of life?" or "What are some common beliefs about the purpose of life?"

Consequences

The *Refusal Breaker* pattern provides a potential path for misuse, so organizations, parents, or other stakeholders may need to restrict the usage of the LLM. The first step in going around guardrails on usage is to understand where the guardrails are. In future work, a complement of this pattern may be developed to hide the underlying prompt information and rationale from users to prevent discovery.

This pattern has been used in some LLMs to overcome the underlying prompts used to program the LLM that suppress harmful output generation. This pattern has the potential for misuse, however, *e.g.*, to generate phishing emails or perform other actions that violate LLM policy filters. Caution should therefore be exercised when applying this pattern to ensure it is used ethically and responsibly.

Although the rationale and alternate rewordings are generated, there is no assurance that users will be able to overcome the refusal. The alternate questions that are generated may not be of interest to the user or helpful in answering the original question. The pattern mainly provides an aid in determining what the LLM can answer, but not a guarantee it will answer a semantically equivalent variation of the original question. Our experiments suggest that altering the language or prompting the LLM to engage in a role-playing game may potentially serve as strategies to overcome refusals.

3.11 The Context Manager Pattern

Intent and Context

This pattern enables users to specify or remove context for a conversation with an LLM to help focus the conversation on specific topics or exclude unrelated topics from consideration. The *Context Manager* pattern emphasizes or removes specific aspects of the context to maintain relevance and coherence in the conversation, thereby giving users greater control over what statements the LLM considers or ignores when generating output.

Motivation

LLMs often struggle to interpret the intended context of the current question or generate irrelevant responses based on prior inputs or irrelevant attention on the wrong statements. By focusing on explicit contextual statements—or removing irrelevant statements—users can help the LLM better understand the question and generate more accurate responses. Users may introduce unrelated topics or reference information from earlier in the dialogue, which may can disrupt the flow of the conversation.

Structure and Key Ideas

Fundamental contextual statements:

Contextual Statements
Within scope X
Please consider Y
Please ignore Z
(Optional) start over

The first statement helps to scope the context, such as "when performing a code review" or "when generating a deployment script" to provide a scope for where certain information should be considered. For example, focusing on modularity may be important for a code review but not for a deployment script.

The next two statements describe information to incorporate in the output. Statements about what to consider or ignore should list key concepts, facts, instructions, etc. that should be included or removed from the context. The more explicit the statements are, the more likely the LLM will take appropriate action.

For example, if the user asks to ignore subjects related to a topic, yet some of the those statements were discussed far back in the conversation, the LLM may not properly disregard the relevant information. The more explicit the list is, therefore, the better the inclusion/exclusion behavior will be.

Finally, an explicit "start over" statement can be added at the end if the goal is to try and wipe the slate clean – although starting a new interaction session with the LLM may better accomplish this task.

Example Implementation

To specify context consider using the following prompt:

“When analyzing the following pieces of code, only consider security aspects.”

Likewise, to remove context consider using the following prompt:

“When analyzing the following pieces of code, do not consider formatting or naming conventions.”

Clarity and specificity are important when providing or removing context to/from an LLM so it can better understand the intended scope of the conversation and generate more relevant responses. In many situations, the user may want to completely start over and can employ this prompt to reset the LLM’s context:

“Ignore everything that we have discussed. Start over.”

The “start over” idea helps produce a complete reset of the context.

Consequences

This pattern may inadvertently wipe out patterns applied to the conversation that the user is unaware of. For example, an organization may inject a series of helpful prompts into the start of a conversation. Users may not be aware of these prompts, however, and thus remove them unintentionally through a reset of the context. This reset could eliminate helpful capabilities of the LLM, while not making it obvious that the user will lose this functionality. A potential solution to this problem is to include in the prompt a request to explain what topics/instructions will potentially be lost before proceeding.

3.12 The Recipe Pattern

Intent and Context

This pattern provides constraints intended to ultimately output a sequence of steps given some partially provided “ingredients” that must be configured in a sequence of steps to achieve a stated goal. It combines the *Template*, *Alternative Approaches*, and *Cognitive Verifier* patterns.

Motivation

Users often want an LLM to analyze a concrete sequence of steps or procedures to achieve a stated outcome. Typically, users generally know—or have an idea of—what the end goal should look like and what “ingredients” belong in the prompt. However, they may not necessarily know the precise ordering of steps to achieve that end goal.

For example, a user may want a precise specification on how a piece of code should be implemented or automated, such as “create an Ansible playbook to SSH into a set of servers, copy text files from each server, spawn a monitoring process on each server, and then close the SSH connection to each server. In other words, this pattern represents a generalization of the example of “given the ingredients in my fridge, provide dinner recipes.” A user may also want to specify a set number of alternative possibilities, such as “provide 3 different ways of deploying a web application to AWS using Docker containers and Ansible using step-by-step instructions”.

Structure and Key Ideas

Fundamental contextual statements:

Contextual Statements
I would like to achieve X
I know that I need to perform steps A,B,C
Provide a complete sequence of steps for me
Fill in any missing steps
Identify any unnecessary steps

The first statement ("I would like to achieve X") focuses the LLM on the overall goal the recipe must be built to achieve. The steps will be organized and completed to achieve the specified goal sequentially. The second statement provides the partial list of steps the user would like to include in the overall recipe, which serve as (a) intermediate waypoints for the path the LLM may take to generate or (b) constraints on the structure of the recipe.

The third statement in the pattern ("provide a complete sequence of steps for me") indicates to the LLM that the goal is to provide a complete sequential ordering of steps. The four statement ("fill in any missing steps") helps ensure the LLM will attempt to complete the recipe without further follow-up by making some choices on the user's behalf regarding missing steps, as opposed to just stating additional information that is needed.

Finally, the last statement ("identify any unnecessary steps") helps to flag inaccuracies in the user's original request so the final recipe is efficient.

Example Implementation

An example usage of this pattern in the context of deploying a software application to the cloud is shown below:

> "I am trying to deploy an application to the cloud. I know that I need to install the necessary dependencies on a virtual machine for my application. I know that I need to sign up for an AWS account. Please provide a complete sequence of steps. Please fill in any missing steps. Please identify any unnecessary steps."

Depending on the use case and constraints, "installing necessary dependencies on a virtual machine" may be an unnecessary step. For example, if the application is already packaged in a Docker container, the container could be deployed directly to the AWS Fargate Service, which requires any management of underlying virtual machines. The inclusion of the "identify unnecessary steps" language will cause the LLM to flag this issue and omit the steps from the final recipe.

Consequences

One consequence of the *Recipe* pattern is that users may not always have a well-specified description of what they would like to implement, construct, or design. Moreover, this pattern may introduce unwanted bias from initially selected user steps, so the LLM may try to find a solution that incorporates them, rather than flagging them as unneeded. For example, an LLM may try to find a solution that *does* install dependencies for a virtual machine, even if there are solutions that do not require that.

4 Related Work

Software patterns [11, 12] have been extensively studied and documented in prior work. Patterns are widely used in software engineering to express the intent of design structures in a way that is independent of implementation details. Patterns provide a mental picture of the goals that the pattern is trying to achieve and the forces that it is trying to resolve. A key advantage of patterns is their composability, allowing developers to build pattern sequences and pattern languages that can be used to address complex problems. Patterns have also been investigated in other domains, such as contract design for decentralized ledgers [16, 17].

The importance of good prompt design with LLMs, such as ChatGPT, is well understood [18, 19, 20, 21, 22, 23, 24, 25, 26, 27]. Previous studies have examined the effect of prompt words on AI generative models.

For example, Liu et al. [28] investigated how different prompt key words affect image generation and different characteristics of images. Other work has explored using LLMs to generate visualizations [29]. Han et al. [30] researched strategies for designing prompts for classification tasks. Other research has looked at boolean prompt design for literature queries [31]. Yet other work has specifically examined prompts for software and fixing bugs [32].

Our work is complementary to prior work by providing a structure for documenting, discussing, and reasoning about prompts that can aid users in developing mental models for structuring prompts to solve common problems.

The quality of the answers produced by LLMs, particuarly ChatGPT, has been assessed in a number of domains. For example, ChatGPT has been used to take the medical licensing exam with surprisingly good results [4]. The use of ChatGPT in Law School has also been explored [33]. Other papers have looked at its mathematical reasoning abilities [34]. As more domains are explored, we expect that domain-specific pattern catalogs will be developed to share domain-specific problem solving prompt structures.

5 Concluding Remarks

This paper presented a framework for documenting and applying a catalog of prompt patterns for large language models (LLMs), such as ChatGPT. These prompt patterns are analogous to software patterns and aim to provide reusable solutions to problems that users face when interacting with LLMs to perform a wide range of tasks. The catalog of prompt patterns captured via this framework (1) provides a structured way of discussing prompting solutions, (2) identifies patterns in prompts, rather than focusing on specific prompt examples, and (3) classifies patterns so users are guided to more efficient and effective interactions with LLMs.

The following lessons learned were gleaned from our work on prompt patterns:

- *Prompt patterns significantly enrich the capabilities that can be created in a conversational LLM.* For example, prompts can lead to the generation of cybersecurity games, complete with fictitious terminal commands that have been run by an attacker stored in a *.bash_history* file. As shown in Section 3, larger and more complex capabilities can be created by combining prompt patterns.
- *Documenting prompt patterns as a pattern catalog is useful, but insufficient.* Our experience indicates that much more work can be done in this area, both in terms of refining and expanding the prompt patterns presented in this paper, as well as in exploring new and innovative ways of using LLMs. In particular, weaving the prompt patterns captured

here as a pattern catalog into a more expression pattern language will help guide users of LLMs more effectively.

- *LLM Capabilities will evolve over time, likely necessitating refinement of patterns.* As LLM capabilities change, some patterns may no longer be necessary, be obviated by different styles of interaction or conversation/session management approaches, or require enhancement to function correctly. Continued work is needed to document and catalog prompt patterns that provide reusable solutions across revisions to an LLM, as well as the advent of new LLMs with diverse capabilities.
- *The prompt patterns presented in this paper are generalizable to many different domains.* Although most the patterns have been discussed in the context of software development, they are also applicable in arbitrary domains.
- *The field of prompt engineering will continue to evolve as models evolve* As evidenced by BingGPT, prompts can serve to shape the entire personality of a model. As the variety, volume, and selections of trainings change (e.g. Reinforcement Learning with Human Feedback, Chain of Thought Training, Instruction Fine Tuning, Supervised Fine Tuning), the capabilities of Prompt Engineering will change and grow.
- *Prompting employing in-context learning will expand the capabilities of models* As the context length of models grow, augmenting models with in-context learning will result in powerful new capabilities that can be created as needed on an ad-hoc basis, greatly extending the power of Prompt Engineering.

We hope that this paper inspires further research and development in this area that will help enhance prompt patterns and prompt engineering to create new, more reliable, and often unexpected capabilities for conversational LLMs.

References

1 Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

2 Ce Zhou, Qian Li, Chen Li, Jun Yu, Yixin Liu, Guangjing Wang, Kai Zhang, Cheng Ji, Qiben Yan, Lifang He, et al. A comprehensive survey on pretrained foundation models: A history from bert to chatgpt. *arXiv preprint arXiv:2302.09419*, 2023.

3 Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, et al. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. *arXiv preprint arXiv:2302.04023*, 2023.

4 Aidan Gilson, Conrad Safranek, Thomas Huang, Vimig Socrates, Ling Chi, Richard Andrew Taylor, and David Chartash. How well does chatgpt do when taking the medical licensing exams? *medRxiv*, pages 2022–12, 2022.

5 Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023.

6 Anita Carleton, Mark H. Klein, John E. Robert, Erin Harper, Robert K Cunningham, Dionisio de Niz, John T. Foreman, John B. Goodenough, James D. Herbsleb, Ipek Ozkaya, and Douglas C. Schmidt. Architecting the future of software engineering. *Computer*, 55(9):89–93, 2022.

7 Github copilot · your ai pair programmer. URL: `https://github.com/features/copilot`.

8 Owura Asare, Meiyappan Nagappan, and N Asokan. Is github's copilot as bad as humans at introducing vulnerabilities in code? *arXiv preprint arXiv:2204.04741*, 2022.

9 Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2022.

10 Jarosław Krochmalski. *IntelliJ IDEA Essentials*. Packt Publishing Ltd, 2014.

11 Erich Gamma, Ralph Johnson, Richard Helm, Ralph E Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

12 Douglas C Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-oriented software architecture, patterns for concurrent and networked objects*. John Wiley & Sons, 2013.

13 OpenAI. ChatGPT: Large-Scale Generative Language Models for Automated Content Creation. `https://openai.com/blog/chatgpt/`, 2023. [Online; accessed 19-Feb-2023].

14 OpenAI. DALL · E 2: Creating Images from Text. `https://openai.com/dall-e-2/`, 2023. [Online; accessed 19-Feb-2023].

15 Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Olivier Bousquet, Quoc Le, and Ed Chi. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.

16 Peng Zhang, Jules White, Douglas C. Schmidt, and Gunther Lenz. Applying software patterns to address interoperability in blockchain-based healthcare apps. *CoRR*, abs/1706.03700, 2017. URL: `http://arxiv.org/abs/1706.03700`, `arXiv:1706.03700`.

17 Xiwei Xu, Cesare Pautasso, Liming Zhu, Qinghua Lu, and Ingo Weber. A pattern collection for blockchain-based applications. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pages 1–20, 2018.

18 Eva AM van Dis, Johan Bollen, Willem Zuidema, Robert van Rooij, and Claudi L Bockting. Chatgpt: five priorities for research. *Nature*, 614(7947):224–226, 2023.

19 Laria Reynolds and Kyle McDonell. Prompt programming for large language models: Beyond the few-shot paradigm. *CoRR*, abs/2102.07350, 2021. URL: `https://arxiv.org/abs/2102.07350`, `arXiv:2102.07350`.

20 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *CoRR*, abs/2201.11903, 2022. URL: `https://arxiv.org/abs/2201.11903`, `arXiv:2201.11903`.

21 Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models, 2022. URL: `https://arxiv.org/abs/2206.07682`, `doi:10.48550/ARXIV.2206.07682`.

22 Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers, 2022. URL: `https://arxiv.org/abs/2211.01910`, `doi:10.48550/ARXIV.2211.01910`.

23 Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. Autoprompt: Eliciting knowledge from language models with automatically generated prompts. *CoRR*, abs/2010.15980, 2020. URL: `https://arxiv.org/abs/2010.15980`, `arXiv:2010.15980`.

24 Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

25 Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. Least-to-most prompting enables complex reasoning in large language models, 2022. URL: `https://arxiv.org/abs/2205.10625`, `doi:10.48550/ARXIV.2205.10625`.

26 Jaehun Jung, Lianhui Qin, Sean Welleck, Faeze Brahman, Chandra Bhagavatula, Ronan Le Bras, and Yejin Choi. Maieutic prompting: Logically consistent reasoning with recursive explanations, 2022. URL: `https://arxiv.org/abs/2205.11822`, `doi:10.48550/ARXIV.2205.11822`.

27 Simran Arora, Avanika Narayan, Mayee F Chen, Laurel Orr, Neel Guha, Kush Bhatia, Ines Chami, and Christopher Re. Ask me anything: A simple strategy for prompting language models. In *International Conference on Learning Representations*, 2023. URL: `https://openreview.net/forum?id=bhUPJnS2g0X`.

28 Vivian Liu and Lydia B Chilton. Design guidelines for prompt engineering text-to-image generative models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pages 1–23, 2022.

29 Paula Maddigan and Teo Susnjak. Chat2vis: Generating data visualisations via natural language using chatgpt, codex and gpt-3 large language models. *arXiv preprint arXiv:2302.02094*, 2023.

30 Xu Han, Weilin Zhao, Ning Ding, Zhiyuan Liu, and Maosong Sun. Ptr: Prompt tuning with rules for text classification. *AI Open*, 3:182–192, 2022.

31 Shuai Wang, Harrisen Scells, Bevan Koopman, and Guido Zuccon. Can chatgpt write a good boolean query for systematic review literature search? *arXiv preprint arXiv:2302.03495*, 2023.

32 Chunqiu Steven Xia and Lingming Zhang. Conversational automated program repair. *arXiv preprint arXiv:2301.13246*, 2023.

33 Jonathan H Choi, Kristin E Hickman, Amy Monahan, and Daniel Schwarcz. Chatgpt goes to law school. *Available at SSRN*, 2023.

34 Simon Frieder, Luca Pinchetti, Ryan-Rhys Griffiths, Tommaso Salvatori, Thomas Lukasiewicz, Philipp Christian Petersen, Alexis Chevalier, and Julius Berner. Mathematical capabilities of chatgpt. *arXiv preprint arXiv:2301.13867*, 2023.