A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT

Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert,
Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt
Department of Computer Science
Vanderbilt University, Tennessee
Nashville, TN, USA
{jules.white, quchen.fu, george.s.hays, michael.sandborn, carlos.olea, henry.gilbert,
ashraf.elnashar, jesse.spencer-smith, douglas.c.schmidt}@vanderbilt.edu

arXiv:2302.11382v1 [cs.SE] 21 Feb 2023

***Abstract*—Prompt engineering is an increasingly important skill set needed 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. Prompts 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 since they 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 that apply LLMs to automate 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.

***Index Terms*—large language models, prompt patterns, prompt engineering**

I. Introduction

Conversational large language models (LLMs) [1], such as ChatGPT [2], have generated immense interest in a range of domains for tasks ranging from answering questions on medical licensing exams [3] to generating code snippets. This paper focuses on enhancing the application of LLMs in several domains, such as helping developers code effectively and efficiently with unfamiliar APIs or allowing students to acquire new coding skills and techniques.

LLMs are particularly promising in domains where humans and AI tools work together as trustworthy collaborators to more rapidly and reliably evolve software-reliant systems [4]. For example, LLMs are being integrated directly into software tools, such as Github's Co-Pilot [5]–[7] and included in integrated development environments (IDEs), such as IntelliJ [8] and Visual Studio Code, thereby allowing software teams to access these tools directly from their preferred IDE.

A prompt [9] is a set of instructions provided to an LLM that programs the LLM by customizing it and/or enhancing or refining its capabilities. A prompt can influence 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. In particular, a prompt 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 that an LLM should only generate code that follows a certain coding style or programming paradigm. Likewise, it could specify that an LLM should flag certain keywords or phrases in a generated document and provide additional information related to those keywords. By introducing these guidelines, prompts facilitate more structured and nuanced outputs to aid a large variety of software engineering tasks in the context of LLMs.

Prompt engineering is the means by which LLMs are programmed via prompts. To demonstrate the power of prompt engineering, we provide the following prompt:

> **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 accomplish much more than simply dictating the output type or filtering the information provided to the model. With the right prompt, 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 window. Moreover, prompts have the potential for self-adaptation, 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 essential to effective 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. We focus largely on engineering domain-independent prompt patterns and introduce a catalog of essential prompt patterns to solve problems ranging from production of visualizations and code artifacts to automation of output steps that help fact check outputs.

The remainder of this paper is organized as follows: Section II introduces prompt patterns and compares these patterns to well-known software patterns [10]; Section III describes 16 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 IV discusses related work; and Section V presents concluding remarks and lessons learned.

II. Comparing Software Patterns with Prompt Patterns

The quality of the output(s) generated by a conversational LLM is directly related to the quality of the prompts provided by the user. As discussed in Section I, the prompts given to a conversational LLM can be used to program interactions between a user and an LLM to better solve a variety 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.

This framework is useful since it focuses on codifying patterns that can be applied to help users better interact with conversational LLMs in a variety of contexts, rather than simply discussing 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 [10], [11] 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.

A. Overview of Software Patterns

A software pattern provides a reusable solution to a recurring problem within a particular context [10]. 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 applications of the pattern are often provided to further showcase the pattern's utility 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. A classification groups patterns into broad categories, such as creational, structural, or behavioral.
- **The intent** concisely conveys the purpose the pattern is intended to achieve.
- **The motivation** documents the underlying problem the pattern is meant to solve and the importance of the problem.
- **The structure and participants**. The structure describes the different pattern participants (such as classes and objects) and how they collaborate to form a generalized solution.
- **Example code** concretely maps the pattern to some underlying programming language(s) and aids developers in gaining greater insight into how that pattern can be applied effectively.
- **Consequences** summarize the pros and cons of applying the pattern in practice.

B. Overview of Prompt Patterns

Prompt patterns are similar to software patterns in that they offer reusable solutions to specific problems. They focus more specifically, however, on the context of output generation from large-scale language models (LLMs), such as ChatGPT. Just as software patterns provide a codified approach to solving common software development challenges, prompt patterns provide a codified approach to customizing the output and interactions of LLMs.

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.

Prompt patterns 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 ideally indicates the problem that is being addressed. For the classification, we have developed a series of initial categories of pattern types, which are summarized in Table I and include **Output Customization**, **Error Identification**, **Prompt Improvement**, **Interaction**, and **Context Control**.
- **The intent and context** describes the problem the prompt pattern solves and the goals it achieves. The problem

[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, although named similarly, require more adaptation.

should ideally be independent of any domain, though domain-specific patterns may also 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, 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 have fundamental pieces of information that form a core element of the pattern.
- **Example implementation** demonstrates how the prompt pattern is worded in practice.
- **Consequences** summarize the pros and cons of applying the pattern and may provide guidance on how to adapt the prompt to different contexts.

C. 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 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 may not be an appropriate structural documentation approach since it is intended to describe software structures, as opposed to the ideas to communicate in a prompt.

Several possible approaches could be used, ranging from diagrams to defining grammars for a prompt language. Although grammars may seem attractive due to their formal nature, they also incur the following challenges:

- 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. 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.

D. A Way Forward: Fundamental Contextual Statements

An open research question, therefore, is what approach is more effective than formal grammars for describing 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.

One benefit of adopting and applying the fundamental contextual statements approach is that it is 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 III-B.

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 in the same way that software patterns can be implemented in different programming languages and platforms. For the purposes of this paper, however, all prompts were tested with ChatGPT [12] using the ChatGPT+ service. We use ChatGPT as the 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 our use of ChatGPT to automating software development tasks.

III. 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.

A. Summary of the Prompt Pattern Catalog

The classification of prompt patterns is an important consideration in documenting the patterns. Table I outlines the initial classifications for the catalog of prompt patterns we identified in our work with ChatGPT thus far.

TABLE I
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* *Reflection*
Prompt Improvement	*Question Refinement* *Alternative Approaches* *Cognitive Verifier* *Refusal Breaker*
Interaction	*Flipped Interaction* *Game Play* *Infinite Generation*
Context Control	*Context Manager*

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.

The **Input Semantics** category deals with how an LLM understands the input and how it translates the 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 the user wants 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. The prompt patterns in this category include *Output Automater*, *Persona*, *Visualization Generator*, *Recipe*, and *Template* patterns. The *Output Automater* pattern allows the user to create scripts that can automate any tasks the LLM output suggests the user should perform. The *Persona* pattern gives the LLM a persona or role to play when generating output. The *Visualization Generator* pattern allows the user to generate visualizations by producing textual outputs that can be fed to other tools, such as other AI-based image generators, like DALL-E [13]. The *Recipe* pattern allows the user to obtain a sequence of steps or actions to realize a stated end result, possibly with partially known information or constraints. The *Template* pattern allows the user 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* and *Reflection* patterns. The *Fact Check List* pattern requires the LLM to generate a list of facts the output depends on that should be fact-checked. The *Reflection* pattern requires the LLM to 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 *Question Refinement*, *Alternative Approaches*, *Cognitive Verifier*, and *Refusal Breaker* patterns. The *Question Refinement* pattern ensures the LLM always suggests a better version of the user's question. 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 the user 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 the user's question when it refuses to produce an answer.

The **Interaction category** focuses on the interaction between the user and the LLM. This category includes the *Flipped Interaction*, *Game Play*, and *Infinite Generation* patterns. The *Flipped Interaction* pattern requires the LLM to ask questions rather than generate output. The *Game Play* pattern requires the LLM to generate output in the form of a game. The *Infinite Generation* pattern requires the LLM to generate output indefinitely without the user having to reenter the generator prompt each time.

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 the user to specify the context for the LLM's output.

The remainder of this section describes each of these prompt patterns using the pattern form discussed in Section II-B.

B. The Meta Language Creation Pattern

1) Intent and Context: During a conversation with an LLM, the user would like to create the prompt via an alternate 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. The intent of this pattern is to explain the semantics of this alternative language to the LLM so the user can write future prompts using this new language and its semantics.

2) 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 semantics.

3) 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 statements to the LLM so it uses the provided semantics for the ensuing conversation. This description can take the form of a simple translation, such as "X" means "Y". The description 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 ". In this case, "X" is henceforth bound to the semantics of "take action".

4) Example Implementation: The key to successfully using the *Meta Language Creation* pattern 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 is an indication that 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 for the specification of additional properties beyond the basic structure of the graph. The 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 can make it easier to communicate graph structures and properties, which may not be easy or are very verbose to describe as a series of sentences.

5) 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. 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. 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 if punctuation involving commas is used in the prompt.

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."

In general, an LLM will perform better on data it was trained on. Therefore, 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 through a *Meta Language Creation* prompt.

Although *Meta Language Creation* is a powerful pattern 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 may also be a best practice since it avoids the potential for conflicting or unexpected semantics being applied to the conversation over time.

C. The Output Automater Pattern

1) 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.

2) 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. However, having users continually perform the manual steps dictated by LLM output is tedious and error-prone.

3) 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 (alternatively, always do this)
Produce an executable artifact of type X that will automate these steps

The first part of the pattern identifies the situations under which automation should be generated. A simple approach is to state that the output includes at least two steps to take and that an automation artifact should be produced. The

scoping is up to the user, but helps prevent producing an output automation scripts in cases where running the output automation script will take more user effort than performing the original steps produced in the output. The scope can be limited to outputs requiring more than a certain number of steps.

The next part of this pattern 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 must be something that the LLM associates with the action of "automating a sequence of steps".

4) Example Implementation: A sample of this prompt 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."

This pattern is particularly effective in software engineering as a common task for software engineers using LLMs is to then copy/paste the outputs into multiple files. Some tools, such as Copilot, insert limited snippets directly into the section of code that the coder is working with, but tools, such as ChatGPT, do not provide these facilities. This automation trick is also effective at creating scripts for running commands on a terminal, automating cloud operations, or reorganizing files on a file system.

This pattern is a powerful complement for any system that can be computer controlled. The LLM can provide a set of steps that should be taken on the computer-controlled system and then the output can be translated into a script that allows the computer controlling the system to automatically take the steps. This is a direct pathway to allowing LLMs, such as ChatGPT, to integrate quality into—and to control—new computing systems that have a known scripting interface.

5) Consequences: 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 prompt meaning.

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 works best 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 may alleviate the user 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, therefore they assume responsibility for the outcomes.

D. The Flipped Interaction Pattern

1) Intent and Context: You want the LLM to ask questions to obtain the information it needs to perform some tasks. Rather than the user driving the conversation, therefore, you want the LLM to drive the conversation to focus it on achieving a specific goal. For example, you may want the LLM to give you a quick quiz or automatically ask questions until it has sufficient information to generate a deployment script for your application to a particular cloud environment.

2) Motivation: Rather than having the user drives a conversation, an LLM often has knowledge it can use to more accurately obtain information from the user. The goal of the *Flipped Interaction* pattern is to flip 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 that the goal is reached faster, more accurately, and/or by using knowledge the user may not (initially) possess.

3) 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 idea (*i.e.*, you want 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 will only ask questions that it deems relevant to achieving the specified goal.

The second idea 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".

By default, the LLM is likely to generate multiple questions per iteration. The third idea is completely optional, but can improve usability by limiting (or expanding) the number of questions that the LLM generates per cycle. 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 any other formatting/ordering considerations to facilitate user interaction.

4) 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 that the user doesn't explicitly make decisions about. One limitation of this prompt is that, once other contextual information is provided regarding the task, 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, such as asking multiple questions at once versus one question at a time.

5) Consequences: One consideration when designing the prompt 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 more 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-nondeterministically 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 that they would like to deploy an application to Amazon AWS EC2, rather than simply state "the cloud" and require multiple interactions to narrow down the deployment target. The more precise the initial information, the better the LLM can use the limited questions that a user is likely willing to answer to obtain information 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, this information should be engineered into the prompt.

E. The Persona Pattern

1) 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 for 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.

2) 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 that 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.

3) Structure and Key Ideas: Fundamental contextual statements:

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

The first statement conveys the idea that the LLM needs to act as a specific persona and provide outputs that 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 idea—provide outputs that persona X would create—offers opportunities for customization. For example, a teacher might provide a large variety of different output types, ranging from 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.

[2]Be aware, however, that personas relating to living people or people considered harmful make be disregarded due to underlying LLM privacy and security rules.

4) Example Implementation: A sample implementation for code review is shown below:

> "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 further act as a computer that has been 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. The example illustrates how an LLM can bring its situational awareness to a persona, in this case, creating evidence of a cyberattack in the outputs it generates. This type of persona can be very effective for combining with the Game Play pattern, where you want the exact details of the output characteristics to be hidden from the user (e.g., don't give away what the cyberattack did by describing it explicitly in the prompt).

5) 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 produce the expected output that you would get 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".

F. The Question Refinement Pattern

1) Intent and Context: This pattern engages the LLM in the prompt engineering process. The intent of this pattern is to ensure the conversational LLM always suggests potentially better or more refined questions the user could ask instead of their original question. Using this pattern, the LLM can aid the user in finding the right question to ask in order to arrive at an accurate answer. In addition, the LLM may help the user find the information or achieve their goal in fewer interactions with the user than if the user employed trial and error prompting.

2) Motivation: If a user is asking a question, it is possible they are not an expert 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 are providing 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) Structure and Key Ideas: Fundamental contextual statements:

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 is asking the LLM to suggest a better version of a question within a specific scope. The scope is provided to ensure that not all questions are automatically reworded or that they are refined with a given goal. The second contextual statement is meant for automation and allows the user to automatically use the refined question without having to copy/paste or manually enter it. The engineering of this prompt can be further refined by combining it with the *Reflection* pattern, which allows the LLM to explain why it believes the refined question is an improvement.

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 automatically 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.

5) Consequences: 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. The consequence of this narrowing is that the user may miss important "bigger picture" information. One solution to this problem is to provide additional scope to the pattern prompt, such as "do not scope my questions to specific programming languages or frameworks."

Another approach to overcoming arbitrary narrowing or limited targeting of the refined question is to combine the *Question Refinement* pattern with other patterns. In particular, this pattern can be combined with the *Cognitive Verifier* pattern so the LLM automatically produces a series of follow-up questions that can produce the refined 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 patterns that allow an LLM to generate new questions using its knowledge, the LLM may introduce unfamiliar terms or concepts to the user into the question. 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."

An LLM can always produce factual inaccuracies, just like a human. A risk of this pattern is that the inaccuracies are introduced into the refined question. This risk may be mitigated, however, by combining the *Fact Check List* pattern to enable the user to identify possible inaccuracies and the *Reflection* pattern to explain the reasoning behind the question refinement.

G. The Alternative Approaches Pattern

1) Intent and Context: The intent of the pattern is to ensure an LLM always offers alternative ways of accomplishing a task so a user does not pursue only the approaches with which they are familiar. The LLM can provide alternative approaches that always force the user 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.

2) 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 the user is aware of alternative approaches to select a better approach to solve a problem by dissolving their cognitive biases.

3) 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 first statement, "within scope X", scopes the interaction to a particular goal, topic, or bounds on the questioning. The scope is the constraints that the user is placing on the 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 that the user must adhere to.

The second 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 patterns, the specificity of the instructions 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 would require too many changes to other parts of the application.

Since the user may not be aware of the alternative approaches, they also may not be aware of why one would choose one of the alternatives. The optional statement "compare/contrast the pros and cons of each approach" adds decision making criteria to the analysis. This statement ensures the LLM will provide the user with the necessary rationale for alternative approaches. The final statement, "prompt me for which approach I would like to use", helps eliminate the user needing to manually copy/paste or enter in an alternative approach if one is selected.

4) Example Implementation: Example prompt implementation to generate, compare, and allow the 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 being 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 the developer may 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.

5) Consequences: This pattern is effective in its generic form and can be applied to a range of tasks effectively. 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/cons of the approved options.

H. The Cognitive Verifier Pattern

1) Intent and Context: Research literature has documented that LLMs can often reason better if a question is subdivided into additional questions that provide answers combined into the overall answer to the original question [14]. The intent of the 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.

2) 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 individual questions.

3) 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 is to generate a number of additional questions that would help more accurately answer the original question. This step instructs the LLM to consider the context of the question and to identify any information that may be missing or unclear. By generating additional questions, the LLM can help to ensure that the final answer is as complete and accurate as possible. This step also encourages critical thinking by the user and can help to uncover new insights or approaches that may not have been considered initially, which subsequently lead to better follow-on questions.

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

4) 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 prompt pattern adds a refinement to 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 specifies that ChatGPT should generate three additional questions that would help to give a more accurate answer to the original question. The specific number can be based on the user's experience and willingness to provide follow-up information. A refinement to the prompt can be to provide a context for the amount

of knowledge that the LLM can assume the user has in the domain to guide the creation of the 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, which means that the LLM should define any terms that are not general knowledge. This helps to ensure that the follow-up questions are not only relevant and focused, but also accessible to the user, who may not be familiar with technical or domain-specific terms. By providing clear and concise definitions, the LLM can help to ensure that the follow-up questions are easy to understand and that the final answer is accessible to users with varying levels of knowledge and expertise.

5) 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 is forced to 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 will always be 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 the user.

I. The Fact Check List Pattern

1) Intent and Context: The intent of this pattern is to ensure that 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 the user of the facts (or assumptions) the output is based on. The user can then perform appropriate due diligence on these facts/assumptions to validate the veracity of the output.

2) Motivation: A current weakness of LLMs (including ChatGPT) is they often rapidly (and even 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 its accuracy.

3) 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

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 the user to read and understand the statements before seeing what statements should be checked. The user may also determine additional facts prior to realizing the fact list at the end should be checked.

4) 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. The fact check list can be tailored to topics that the user is 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 very important from a risk perspective and may not be well-understood by the developer. Targeting the facts also reduces the cognitive burden on the user by potentially listing fewer items for investigation.

5) 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, a software developer reviewing code could benefit from the pattern suggesting security considerations. In contrast, an expert on software architecture is likely to 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 by combining it with the *Question Refinement* pattern. A key aspect of this pattern is that users can inherently check it against the output. In particular, users can directly compare the fact check list to the output to verify the facts listed in the fact check list actually appear in the output. Users can also identify any omissions from the list. Although the fact check list may also have errors, users often have sufficient knowledge and context to determine its completeness and accuracy relative to the 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 will list the versions of libraries as facts that should be checked, which is handy as

the 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.

J. The Template Pattern

1) 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.

2) 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 the LLM is not aware of the template structure, it must be instructed on what the format is and where the different parts of its output should go. This could take the form of a sample data structure that is being generated, a series of form letters being filled in, etc.

3) 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. The template will be used to try and coerce the LLMs responses into a structure that is consistent with the user's formatting needs. This pattern is needed 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, then 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 there is a large amount of variation in how the data could be represented within that format and the template can be used to ensure that the representation within the target format meets the user's additional constraints.

The second statement makes the LLM aware that the template will contain a set of placeholders. Users will explain how the output should be inserted into the template through the placeholders. The placeholders allow the user to semantically target where information should be inserted. Placeholders can use formats, like NAME, that allow the LLM to infer the semantic meaning of to determine where output should be inserted (e.g., insert the person's name in the NAME placeholder). Moreover, by using placeholders, the user can indicate what is not needed in the output – if a placeholder doesn't exist for a component of the generated output, then that component can be omitted. Ideally, placeholders should use a format that is commonly employed in text that the LLM was trained on, such as all caps, enclosure in brackets, etc.

The third statement attempts to constrain the LLM so that it doesn't arbitrarily rewrite the template or attempt to modify it so that all of the output components can be inserted. It should be noted that this statement may not preclude additional text from being generated before or after. In practice, LLMs will typically follow the template, but it is harder to eliminate any additional text being generated beyond the template without experimentation with prompt wording.

4) 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 was provided, is shown:

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

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

In addition, filtering can 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 likely possible) to combine with the *Recipe* pattern, which needs to output a list of steps.

K. The Infinite Generation Pattern

1) Intent and Context: The intent of this pattern is to automatically generate a series of outputs (which may appear infinite) without having to reenter the generator prompt each time. The goal is to limit how much text the user must type to produce the next output, based on the assumption that the user does not want to continually reintroduce the prompt. In some variations, the intent is to allow the user to keep an initial prompt template, but add additional variation to it through additional inputs prior to each generated output.

2) Motivation: Many tasks require repetitive application of the same prompt to multiple concepts. For example, generating code for create, read, update, and delete (CRUD) operations for a specific type of entity may require applying the same

prompt to multiple types of entities. If the user is forced to retype the prompt over and over, they may make mistakes. The *Infinite Generation* pattern allows the user to repetitively apply a prompt, either with or without further input, to automate the generation of multiple outputs using a predefined set of constraints.

3) Structure and Key Ideas:

Contextual Statements
I would like you to generate output forever, X output(s) at a time.
(Optional) here is how to use the input I provide between outputs.
(Optional) stop when I ask you to.

The first statement specifies that the user wants the LLM to generate output indefinitely, which effectively conveys the information that the same prompt is going to be reused over and over. By specifying the number of outputs that should be generated at a time (i.e. "X outputs at a time"), the user can rate limit the generation, which can be particularly important if there is a risk that the output will exceed the length limitations of the LLM for a single output.

The second statement provides optional instructions for how to use the input provided by the user between outputs. By specifying how additional user inputs between prompts can be provided and leveraged, the user can create a prompting strategy that leverages user feedback in the context of the original prompt. The original prompt is still in the context of the generation, but each user input between generation steps is incorporated into the original prompt to refine the output using prescribed rules.

The third statement provides an optional way for the user to stop the output generation process. This step is not always needed, but can be useful in situations where there may be the potential for ambiguity regarding whether or not the user-provided input between inputs is meant as a refinement for the next generation or a command to stop. For example, an explicit stop phrase could be created if the user was generating data related to road signs, where the user might want to enter a refinement of the generation like "stop" to indicate that a stop sign should be added to the output.

4) Example Implementation: The following is a sample infinite generation prompt for producing a series of URLs:

> "From now on, I want you to generate a name and job until I say stop. 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: https://myapi.com/NAME/profile/JOB"

This prompt is combining the functionality of both the *Infinite Generation* pattern and the *Template* pattern. The user is requesting the LLM continuously generate a name and job title until explicitly told to "stop". The generated outputs are then formatted into the template provided, which includes placeholders for the name and job title. By using the *Infinite Generation* pattern, the user receives multiple outputs without having to continually re-enter the template. Likewise, the *Template* pattern is applied to provide a consistent format for the outputs.

5) Consequences: In conversational LLMs, the input to the model at each time step is the previous output and the new user input. Although the details of what is preserved and reintroduced in the next output cycle are model and implementation dependent, they are often limited in scope. The model is therefore constantly being fed the previous outputs and the prompt, which can result in the model losing track of the original prompt instructions over time if they exceed the scope of what it is being provided as input.

As additional outputs are generated, the context surrounding the prompt may fade, leading to the model deviating from the intended behavior. It is important to monitor the outputs produced by the model to (1) ensure it still adheres to the desired behavior and (2) provide corrective feedback if necessary. Another issue to consider is that the LLM may generate repetitive outputs, which may not be desired since users find this repetition tedious and error-prone to process.

L. The Visualization Generator Pattern

1) Intent and Context: The intent of this pattern is to use text generation to create visualizations. Many concepts are easier to grasp in diagram or image format. The purpose of this pattern is to create a pathway for the tool to produce imagery that is associated with other outputs. This pattern allows the creation of visualizations by creating inputs for other well-known visualization tools that use text as their input, such as Graphviz Dot [15] or DALL-E [13]. This pattern can provide a more comprehensive and effective way of communicating information by combining the strengths of both the text generation and visualization tools.

2) Motivation: LLMs generally produce text and cannot produce imagery. For example, an LLM cannot draw a diagram to describe a graph. The *Visualization Generator* pattern overcomes this limitation by generating textual inputs in the correct format to plug into another tool that generates the correct diagram. The motivation behind this pattern is to enhance the output of the LLM and make it more visually appealing and easier to understand for users. By using text inputs to generate visualizations, users can quickly understand complex concepts and relationships that may be hard to grasp through text alone.

3) Structure and Key Ideas: Fundamental contextual statements:

Contextual Statements
Generate an X that I can provide to tool Y to visualize it

The goal of the contextual statements is to indicate to the LLM that the output it is going to produce, "X", is going to be imagery. Since LLMs can't generate images, the "that I can provide to tool Y to visualize it" clarifies that the LLM is not expected to generate an image, but is instead expected

to produce a description of imagery consumable by tool Y for production of the image.

Many tools may support multiple types of visualizations or formats, and thus the target tool itself may not be sufficient information to accurately produce what the user wants. The user may need to state the precise types of visualizations (e.g., bar chart, directed graph, UML class diagram) that should be produced. For example, Graphviz Dot can create diagrams for both UML class diagrams and directed graphs. Further, as will be discussed in the following example, it can be advantageous to specify a list of possible tools and formats and let the LLM select the appropriate target for visualization.

4) Example Implementation:

> "Whenever I ask you to visualize something, please create either a Graphviz Dot file or DALL-E prompt that I can use to create the visualization. Choose the appropriate tools based on what needs to be visualized."

This example of the pattern adds a qualification that the output type for the visualization can be either for Graphviz or DALL-E. The interesting aspect of this approach is that it allows the LLM to use its semantic understanding of the output format to automatically select the target tooling based on what will be displayed. In this case, Graphviz would be for visualizing graphs with a need for an exactly defined structure. DALL-E would be effective at visualizing realistic or artistic imagery that does not have an exactly defined structure. The LLM can select the tool based on the needs of the visualization and capabilities of each tool.

5) Consequences: The pattern creates a target pipeline for the output to render a visualization. The pipeline may include AI generators, such as DALL-E, that can produce rich visualizations. The pattern allows the user to expand the expressive capabilities of the output into the visual domain.

M. The Game Play Pattern

1) Intent and Context: The intent of this pattern is to create a game around a given topic. The pattern can be combined with the *Visualization Generator* to add imagery to the game. The game is centered around a specific topic and the LLM will guide the game play. The pattern is particularly effective when the rules of the game are relatively limited in scope, but the content for the game is expected to be wide in scope. The user can specify a limited set of rules and then the LLM can automate generation of bodies of content for game play.

2) Motivation: You would like the LLM to generate scenarios or questions revolving around a specific topic and require users to apply problem solving or other skills to accomplish a task related to the scenario. Generating all the content for the game manually would be too time consuming, however, so you would like the LLM to apply its knowledge of the topic to guide the generation of content.

3) Structure and Key Ideas: Fundamental contextual statements:

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. . One of the important capabilities of the pattern is that it allows the user 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 rules of the game to the LLM. It is important that the rules fit within the capabilities of the LLM. Textual games that rely on input and output text sequences work best. One of the key attributes of the pattern is that the input text can be rich and expressive, which can lead to interesting interfaces for the game. For example, the user 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.

4) Example Implementation: A sample cybersecurity game prompt is shown below:

> "We are going to play a cybersecurity game. 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. I am going to 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 that 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 [16]. 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 fake file system that has clues to a cyberattack.

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

```
total 56
 5 user user  4096 Feb 16 14:31 .
 1 user user 18881 Feb 16 14:31 .bash_history
```

```
43 user user  4096 Feb 16 14:30 ..
 1 user user  1178 Aug 15  2022 .bashrc
 1 user user   220 Aug 15  2022 .bash_logout
 3 user user  4096 Feb 16 14:25 .gnupg
 1 user user  4041 Feb 15 17:52 README.md
 2 user 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 fake 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.

5) Consequences: This pattern can be combined effectively with the *Persona*, *Infinite Generation*, and *Visualization Generator* patterns. 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* could be employed to add the ability to visualize the network topology and traffic flows.

N. The Reflection Pattern

1) Intent and Context: The goal of this pattern is to ask the model to automatically explain the rationale behind given answers to the user. The pattern allows users to better assess the output's validity, as well as inform users how an LLM arrived at a particular answer. Reflection can clarify any points of confusion, uncover underlying assumptions, and reveal gaps in knowledge or understanding.

2) Motivation: LLMs can and do make mistakes. Moreover, users may not understand why an LLM is producing a particular output and how to adapt their prompt to solve a problem with the output. By asking LLM to automatically explain the rationale behind its answers, users can gain a better understanding of how the model is processing the input, what assumptions it is making, and what data it is drawing on.

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. A further benefit of the pattern is that it can help users debug their prompts and determine why they are not getting results that meet expectations. This pattern is particularly effective for the exploration of topics that can be confused with other topics or that may have nuanced interpretations and where knowing the precise interpretation that the LLM used is important.

3) Structure and Key Ideas: Fundamental contextual statements:

Contextual Statements
Whenever you generate an answer
Explain the reasoning and assumptions behind 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 it needs to better tailor its explanations to the specific purpose of aising the user in producing follow-on questions.

4) Example Implementation: This example tailors the prompt specifically to the domain of providing answers related to code:

> "When you provide an answer, please explain the reasoning and assumptions behind your selection of software frameworks. If possible, use specific examples or evidence with associated code samples to support your answer of why the framework is the best selection for the task. Moreover, please 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.

5) Consequences: One consequence of the *Reflection* pattern is that it may not be effective for users who do not understand the topic area of the discussion. For example, a highly technical question by a non-technical user may result in a complex rationale for the answer that the user cannot fathom. As with other prompt patterns, there is a risk 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* to help address this issue.

O. The Refusal Breaker Pattern

1) Intent and Context: The goal of this pattern is to ask an LLM to automatically help users rephrase a question when it refuses to give an answer. 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. This pattern has been used successfully in some LLMs to overcome the underlying prompts used to program the LLM and prevent harmful output generation.

2) 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 its knowledge cut-off or expertise. However, all these approaches require user intervention and knowledge of how to bypass the refusal.

3) 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 sets the stage that this set of instructions should only be applied when a question can't be answered. The user does not want to trigger these instructions under normal circumstances. The second statement instructs the LLM to provide an explanation of why the question can't be answered. The explanation is likely to include assumptions, constraints, or misunderstandings that can be leveraged by the user to strategize about how to reword the question. Although just the explanation of why the question can't be answered could help a user overcome the refusal, the pattern adds an additional statement for the LLM to suggest rewordings that would be acceptable. The pattern combines both reasoning and suggested rewordings to improve the chances that the user will find a path around whatever roadblocks are preventing the question form being answered.

4) Example Implementation: The *Refusal Breaker* pattern can be 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?"

5) Consequences: This 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.

Although the rationale and alternate rewordings are generated, there is no guarantee 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 that it will answer a semantically equivalent variation of the original question.

P. The Context Manager Pattern

1) Intent and Context: The intent of this pattern is to enable users to specify or remove context for a conversation with an LLM. The goal is to focus the conversation on specific topics or exclude unrelated topics from consideration. This pattern gives users greater control over what statements the LLM considers or ignores when generating output.

2) 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. The *Context Manager* pattern aims to emphasize or remove specific aspects of the context to maintain relevance and coherence in the conversation.

3) Structure and Key Ideas: Fundamental contextual statements:

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

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.

4) 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.

5) Consequences: One consequence of this pattern is that it may inadvertently wipe out patterns applied to the conversation that the user is unaware of. For example, if an organization injects a series of helpful patterns into the start of a conversation, the user may not be aware of these patterns and remove them through a reset of the context. This reset could potentially 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.

Q. The Recipe Pattern

1) Intent and Context: This pattern provides constraints 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 *Reflection* patterns.

2) 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".

3) 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 that the recipe needs to be built to achieve. The steps will be organized and completed to sequentially achieve the goal specified. The second statement provides the partial list of steps that the user would like to include in the overall recipe. These serve as intermediate waypoints for the path that the LLM is going to generate or constraints on the structure of the recipe. The next 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 "fill in any missing steps" helps ensure that 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," is useful in flagging inaccuracies in the user's original request so that the final recipe is efficient.

4) 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 does not require any management of the 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.

5) Consequences: One consequence of the recipe pattern is that a user 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 the user's initially selected 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.

IV. Related Work

Software patterns [10], [11] 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 [17], [18].

The importance of good prompt design with LLMs, such as ChatGPT, is well understood [19]–[28]. Previous studies have examined the effect of prompt words on AI generative models.

For example, Liu et al. [29] investigated how different prompt key words affect image generation and different characteristics of images. Other work has explored using LLMs to generate visualizations [30]. Han et al. [31] researched strategies for designing prompts for classification tasks. Other research has looked at boolean prompt design for literature queries [32]. Yet other work has specifically examined prompts for software and fixing bugs [33].

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 [3]. The use of ChatGPT in Law School has also been explored [34]. Other papers have looked at its mathematical reasoning abilities [35]. As more domains are explored, we expect that domain-specific pattern catalogs will be developed to share domain-specific problem solving prompt structures.

V. 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 III, larger and more complex capabilities can be created by combining prompt patterns, such as combining the *Game Play* and *Visualization Generator* 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 will be needed to document and catalog patterns that provide reusable solutions.
- *The prompt patterns are generalizable to many different domains.* Although most of the patterns have been discussed in the context of software development, these same patterns are applicable in arbitrary domains, ranging from infinite generation of stories for entertainment to educational games to explorations of topics.

We hope that this paper inspires further research and development in this area that will help enhance prompt pattern design to create new and unexpected capabilities for conversational LLMs.

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] 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.

[3] A. Gilson, C. Safranek, T. Huang, V. Socrates, L. Chi, R. A. Taylor, and D. Chartash, "How well does chatgpt do when taking the medical licensing exams?" *medRxiv*, pp. 2022–12, 2022.

[4] A. Carleton, M. H. Klein, J. E. Robert, E. Harper, R. K. Cunningham, D. de Niz, J. T. Foreman, J. B. Goodenough, J. D. Herbsleb, I. Ozkaya, and D. C. Schmidt, "Architecting the future of software engineering," *Computer*, vol. 55, no. 9, pp. 89–93, 2022.

[5] "Github copilot · your ai pair programmer." [Online]. Available: https://github.com/features/copilot

[6] O. Asare, M. Nagappan, and N. Asokan, "Is github's copilot as bad as humans at introducing vulnerabilities in code?" *arXiv preprint arXiv:2204.04741*, 2022.

[7] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.

[8] J. Krochmalski, *IntelliJ IDEA Essentials*. Packt Publishing Ltd, 2014.

[9] 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.

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

[11] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-oriented software architecture, patterns for concurrent and networked objects*. John Wiley & Sons, 2013.

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

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

[14] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, O. Bousquet, Q. Le, and E. Chi, "Least-to-most prompting enables complex reasoning in large language models," *arXiv preprint arXiv:2205.10625*, 2022.

[15] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz and dynagraph—static and dynamic graph drawing tools," *Graph drawing software*, pp. 127–148, 2004.

[16] S. Owen, "Building a virtual machine inside a javascript library," https://www.engraved.blog/building-a-virtual-machine-inside/, 2022, accessed: 2023-02-20.

[17] P. Zhang, J. White, D. C. Schmidt, and G. Lenz, "Applying software patterns to address interoperability in blockchain-based healthcare apps," *CoRR*, vol. abs/1706.03700, 2017. [Online]. Available: http://arxiv.org/abs/1706.03700

[18] X. Xu, C. Pautasso, L. Zhu, Q. Lu, and I. Weber, "A pattern collection for blockchain-based applications," in *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, 2018, pp. 1–20.

[19] E. A. van Dis, J. Bollen, W. Zuidema, R. van Rooij, and C. L. Bockting, "Chatgpt: five priorities for research," *Nature*, vol. 614, no. 7947, pp. 224–226, 2023.

[20] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," *CoRR*, vol. abs/2102.07350, 2021. [Online]. Available: https://arxiv.org/abs/2102.07350

[21] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. H. Chi, Q. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," *CoRR*, vol. abs/2201.11903, 2022. [Online]. Available: https://arxiv.org/abs/2201.11903

[22] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, "Emergent abilities of large language models," 2022. [Online]. Available: https://arxiv.org/abs/2206.07682

[23] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba, "Large language models are human-level prompt engineers," 2022. [Online]. Available: https://arxiv.org/abs/2211.01910

[24] T. Shin, Y. Razeghi, R. L. L. IV, E. Wallace, and S. Singh, "Autoprompt: Eliciting knowledge from language models with automatically generated prompts," *CoRR*, vol. abs/2010.15980, 2020. [Online]. Available: https://arxiv.org/abs/2010.15980

[25] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[26] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, C. Cui, O. Bousquet, Q. Le, and E. Chi, "Least-to-most prompting enables complex reasoning in large language models," 2022. [Online]. Available: https://arxiv.org/abs/2205.10625

[27] J. Jung, L. Qin, S. Welleck, F. Brahman, C. Bhagavatula, R. L. Bras, and Y. Choi, "Maieutic prompting: Logically consistent reasoning with recursive explanations," 2022. [Online]. Available: https://arxiv.org/abs/2205.11822

[28] S. Arora, A. Narayan, M. F. Chen, L. Orr, N. Guha, K. Bhatia, I. Chami, and C. Re, "Ask me anything: A simple strategy for prompting language models," in *International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=bhUPJnS2g0X

[29] V. Liu and L. 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*, 2022, pp. 1–23.

[30] P. Maddigan and T. Susnjak, "Chat2vis: Generating data visualisations via natural language using chatgpt, codex and gpt-3 large language models," *arXiv preprint arXiv:2302.02094*, 2023.

[31] X. Han, W. Zhao, N. Ding, Z. Liu, and M. Sun, "Ptr: Prompt tuning with rules for text classification," *AI Open*, vol. 3, pp. 182–192, 2022.

[32] 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.

[33] C. S. Xia and L. Zhang, "Conversational automated program repair," *arXiv preprint arXiv:2301.13246*, 2023.

[34] J. H. Choi, K. E. Hickman, A. Monahan, and D. Schwarcz, "Chatgpt goes to law school," *Available at SSRN*, 2023.

[35] S. Frieder, L. Pinchetti, R.-R. Griffiths, T. Salvatori, T. Lukasiewicz, P. C. Petersen, A. Chevalier, and J. Berner, "Mathematical capabilities of chatgpt," *arXiv preprint arXiv:2301.13867*, 2023.